HP Vertica Programmer's Guide

HP Vertica Analytics Platform

Software Version: 7.0.x

Document Release Date:
12/18/2013

Contents

HP Vertica Client Libraries

The HP Vertica client driver libraries provide interfaces for connecting your client applications (or third-party applications such as Cognos and MicroStrategy) to your HP Vertica database. The drivers simplify exchanging data for loading, report generation, and other common database tasks.

There are three separate client drivers:

- Open Database Connectivity (ODBC)—the most commonly-used interface for third-party applications and clients written in C, Python, PHP, Perl, and most other languages.
- Java Database Connectivity (JDBC)—used by clients written in the Java programming language.
- ActiveX Data Objects for .NET (ADO.NET)—used by clients developed using Microsoft's .NET Framework and written in C#, Visual Basic .NET, and other .NET languages.

Client Driver Standards

The HP Vertica client drivers are compatible with the following driver standards:

- The ODBC driver complies with version 3.5.1 of the ODBC standard.
- HP Vertica's JDBC driver is a type 4 driver that complies with the JDBC 3.0 standard. It is compiled using JDK version 1.5, and is compatible with client applications compiled using JDK versions 1.5 and 1.6.
- ADO.NET drivers conform to .NET framework 3.0 specifications.

The drivers do not support some of the optional features in the standards. See ODBC Feature Support and JDBC Feature Support and Using ADO.NET for details.

Client Driver and Server Version Compatibility

Usually, each version of the HP Vertica server is compatible with the previous version of the client drivers. This compatibility lets you upgrade your HP Vertica server without having to immediately upgrade your client software. However, some new features of the new server version may not be available through the old drivers.

The following table summarizes the compatibility of each recent version of the client drivers with the HP Vertica server versions.

Client Driver Version	Compatible Server Versions
5.1.x	5.1.x, 6.0.x, 6.1.x
6.0.x	6.0.x, 6.1
6.1.x	6.1.x, 7.0,x
7.0.x	7.0.x

Version 4.1 to 5.1 Client Driver Transition

The client driver libraries were completely rewritten for HP Vertica 5.1 to improve standards compatibility and support more platforms. As a result, some of the classes, functions, properties, and other elements of the driver APIs have been renamed or deprecated in favor of standard ones. See Updating ODBC Client Code From Previous Driver Versions, Updating ADO.NET Client Code From Previous Driver Versions, and Updating Client Code From 4.1 or Earlier JDBC Driver Versions for details on updating your pre-5.1 client code to work with the new client libraries.

HP Vertica ODBC/JDBC Client Installers

The ODBC/JDBC client drivers are a separate installation from the ADO.NET drivers. (ADO.NET support is not available in Community Edition.) As noted in the compatibility table, the 6.x ODBC/JDBC client drivers do not support access to a non HP Vertica 6.x database and above. For example, you cannot use the new 6.x ODBC/JDBC client drivers to access an HP Vertica 5.x database. If you plan on having a mixed HP Vertica environment supporting both 5.x and 6.x HP Vertica database, consider keeping the 5.x drivers installed.

ODBC/JDBC Multiple Version Installations

The following ODBC/JDBC drivers are supported on a single machine:

- 4.x and 5.x ODBC/JDBC drivers can be installed on the same machine.
- 4.x and 6.x ODBC/JDBC drivers can be installed on the same machine.

It is not possible to have both 5.x and 6.x ODBC drivers on a single machine. If you install the 6.x version, it automatically overlays the existing 5.x installation, and any DSN defined against a 5.x HP Vertica database is not supported.

HP Vertica ADO.NET Client Installers

Prior to version 6.x, ADO.Net drivers must be uninstalled prior to installing a later version of the driver. The 6.x ADO.Net drivers require the HP Vertica database to be 6.0.0 or above. The ADO.NET 6.x driver only supports access to an HP Vertica 6.x server. The ADO.NET 4.x plug-in does not work with an HP Vertica 6.x server. If you plan on also using the ODBC bridge and you need to access both HP Vertica 5.x and 6.x databases, consider keeping the 5.x versions of the ODBC/JDBC drivers for the reasons stated previously.

Installing the HP Vertica Client Drivers

You must install the HP Vertica client drivers to access HP Vertica from your client application. The drivers create and maintain connections to the database and provide APIs that your applications use to access your data. The client drivers support connections using JDBC, ODBC, and ADO.NET.

Client Driver Standards

The client drivers support the following standards:

- ODBC drivers conform to ODBC 3.5.1 specifications.
- JDBC drivers conform to JDK 5 specifications.
- ADO.NET drivers conform to .NET framework 3.0 specifications.

The remainder of this section explain the requirements for the HP Vertica client drivers, and the procedure for downloading, installing, and configuring them.

Driver Prerequisites

The following topics explain the system requirements for the client drivers. You need to ensure that your client system meets these requirements before installing and using the client drivers.

ODBC Prerequisites

There are several requirements your client systems must meet before you can install the HP Vertica ODBC drivers.

Operating System

The HP Vertica ODBC driver requires a supported platform. The list of currently-supported platforms can be found on the myVertica portal.

ODBC Driver Manager

The HP Vertica ODBC driver requires that the client system have a supported driver manager. See the myVertica portal for a list of supported driver managers.

UTF-8, UTF-16 and UTF-32 Support

The HP Vertica ODBC driver is a universal driver that supports UTF-8, UTF-16, and UTF-32 encoding. The default setting depends on the client platform. (see Additional ODBC Driver Configuration Settings for more information).

When using the driver with the DataDirect Connect driver manager, DataDirect Connect adapts to the ODBC driver's text encoding settings. You should configure the ODBC driver to use the encoding method that your application requires. This allows strings to be passed between the driver and the application without intermediate conversion.

See Also

- Installing the Client Drivers
- Programming ODBC Client Applications
- Creating an ODBC Data Source Name (DSN)

ADO.NET Prerequisites

The HP Vertica driver for ADO.NET requires the following software and hardware components:

Operating System

The HP Vertica ADO.NET driver requires a supported Windows operating system. The list of supported platforms can be found in the Supported Platforms document at http://www.vertica.com/documentation.

Memory

HP Vertica suggests a minimum of 512MB of RAM.

.NET Framework

The requirements for the .NET framework for ADO.NET in HP Vertica can be found in the Supported Platforms document at http://www.vertica.com/documentation.

See Also

- Programming ADO.NET Applications

Python Prerequisites

Python is a free, agile, object-oriented, cross-platform programming language designed to emphasize rapid development and code readability. Python has been released under several different open source licenses.

HP Vertica's ODBC driver is tested with Python version 2.4. It should work with versions 2.4 through 2.7. It may also work with Python version 3.0, but this is untested.

Python Driver

HP Vertica requires the pyodbc driver module. See your system's Python documentation for installation and configuration information.

Supported Operating Systems

The HP Vertica ODBC driver requires one of the operating systems listed in ODBC Prerequisites.

For usage and examples, see Using Python.

Perl Prerequisites

Perl is a free, stable, open source, cross-platform programming language licensed under its Artistic License, or the GNU General Public License (GPL).

Your Perl scripts access HP Vertica through its ODBC driver, using the Perl Database Interface (DBI) module with the ODBC Database Driver (DBD::ODBC). The HP Vertica ODBC driver is known to be compatible with these versions of Perl:

- 5.8
- 5.10

Later Perl versions may also work.

Perl Drivers

The following Perl driver modules have been tested with the HP Vertica ODBC driver:

- The DBI driver module, version 1.609
- The DBD::ODBC driver module, version 1.22

Other versions may also work.

Supported Client Systems

The HP Vertica ODBC driver requires one of the operating systems and driver managers listed in ODBC Prerequisites.

PHP Prerequisites

PHP is a widely-used general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. PHP is licensed under the PHP License, an open-source BSD-style license certified by the Open Source Initiative.

PHP Modules

The following PHP modules are required:

- php
- php-odbc
- php-pdo
- UnixODBC (if you are using the Unix ODBC driver)
- libiodbc (if you are using the iODBC driver)

Supported Client Systems

The HP Vertica ODBC driver requires one of the operating systems and driver managers listed in ODBC Prerequisites.

Installing the Client Drivers

How you install client drivers depends on the client's operating system:

- For Linux and UNIX clients, you must first install a Linux driver manager. After you have installed the driver manager, there are two different ways to install the client drivers:
  - On Red Hat Enterprise Linux 5, 64-bit and SUSE Linux Enterprise Server 10/11 64-bit, you can use the HP Vertica client RPM package to install the ODBC and JDBC drivers as well as the **vsql** client.
  - On other Linux platforms and UNIX-like platforms you can download the ODBC and JDBC drivers and install them individually.

  Note: The ODBC and JDBC client drivers are installed by the server `.rpm` files. If you have installed HP Vertica Analytics Platform on your Linux system for development or testing purposes, you do not need to download and install the client drivers on it—you just need to configure the drivers. To use ODBC, you need to create a DSN (see Creating an ODBC DSN for Linux, Solaris, AIX, and HP-UX). To use JDBC, you need to add the JDBC client driver to the Java CLASSPATH (see Modifying the Java CLASSPATH).

- On Windows clients, download the 32-bit or 64-bit client installer. The installer provides the ODBC and and JDBC drivers.
- There is an additional Windows installer for the ADO.NET client driver. 32-bit and 64-bit versions of the installer are available. ADO.NET is only available for the Enterprise edition of HP Vertica.

The remainder of this section describes how to install client drivers on different operating systems.

Installing Driver Managers Linux and Other UNIX-like Platforms

If your client platform does not already have a ODBC driver manager, you need to install one before you can use the HP Vertica ODBC client driver. The driver manager provides an interface between your client operating system and the ODBC drivers. See Supported Platforms for a list of driver managers that are supported on each of the client platforms.

HP Vertica provides reference implementations of supported driver managers in binary format. These binaries are built by HP Vertica and used when testing the product. The driver managers are not modified by HP Vertica and are built with the default settings.

The driver managers can be downloaded from the Download tab of the myVertica portal.

HP Vertica provides these binaries as a convenience for users in the event that the supported versions of the driver managers cannot be easily obtained from the original developers. There is no benefit to using the HP Vertica provided binaries if you installed the same or a compatible version of the supported driver-manager as part of your operating system.

The binaries provided by HP Vertica must be installed and configured manually. For example, the .so files must be in a directory that is in the operating system’s "lib" search path, for example, /usr/lib, or other location specified by the shell's library search path variable (LD_LIBRARY_PATH). HP Vertica does not provide instructions for installing and configuring these third party binaries. See the respective websites for the driver managers for installation and configuration information:

- UnixODBC: http://www.unixodbc.org/
- iODBC: http://www.iodbc.org

Installing ODBC Drivers on Linux, Solaris, AIX, and HP-UX

Note: For additional details about supported platforms, see Supported Platforms.

Read Driver Prerequisites before you proceed.

For Red Hat Enterprise Linux and SUSE Linux Enterprise Server, you can download and install a client RPM that installs both the ODBC and JDBC driver as well as the **vsql** client. See Installing the Client RPM on Red Hat and SUSE.

Note: The ODBC and JDBC client drivers are installed by the server `.rpm` files. If you have installed HP Vertica Analytics Platform on your Linux system for development or testing purposes, you do not need to download and install the client drivers on it—you just need to configure the drivers. To use ODBC, you need to create a DSN (see Creating an ODBC DSN for Linux, Solaris, AIX, and HP-UX). To use JDBC, you need to add the JDBC client driver to the Java CLASSPATH (see Modifying the Java CLASSPATH).

The ODBC driver installation packages are broken down by client platform on the myVertica portal. The package's filename is named based on its operating system and architecture (for example, `vertica_7.0..xx_odbc_x86_64_linux.tar.gz`)

Installation Procedure

1. Open a Web browser and log in to myVertica portal.
2. Click the Download tab and locate and download the driver package that corresponds to your client system.
3. If you did not directly download to the client system, transfer the downloaded file to it.
4. Log in to the client system as root.
5. If the directory `/opt/vertica/` does not exist, create it:

   ```
   # mkdir -p /opt/vertica/
   ```

6. Copy the downloaded file to the `/opt/vertica/` directory. For example:

   ```
   # cp vertica_7.0..xx_odbc_x86_64_linux.tar.gz /opt/vertica/
   ```

7. Change to the /opt/vertica/ directory:

   ```
   # cd /opt/vertica/
   ```

8. Uncompress the file you downloaded. For example:

   $ **`tar vzxf vertica_7.0..xx_odbc_x86_64_linux.tar.gz`**

   Two folders are created: one for the include file, and one for the library files. The path of the library file depends on the processor architecture: `lib` for 32-bit libraries, and `lib64` for 64-bit libraries. So, a 64-bit driver client download creates the directories:

   - `/opt/vertica/include`, which contains the header file
   - `/opt/vertica/lib64`, which contains the library file

Post Driver Installation Configuration

You must configure the ODBC driver before you can use it. There are two required configuration files:

- The `odbc.ini` configuration file defines the Data Source Names (DSNs) that tell the ODBC how to access your HP Vertica databases. See Creating an ODBC Data Source Name for instructions to create this file.

- The `vertica.ini` configuration file defines some HP Vertica-specific settings required by the drivers. See Additional ODBC Driver Configuration Settings for instructions to create this file.

Note: If you are upgrading your ODBC driver, you must either update your DSNs to point to the newly-installed driver or create new DSNs. If your `odbc.ini` file references drivers defined in an `odbcinst.ini` file, you just need to update the `odbcinst.ini` file. See Creating an ODBC Data Source Name (DSN) for details.

Installing the Client RPM on Red Hat and SUSE

For Red Hat Enterprise Linux and SUSE Linux Enterprise Server, you can download and install a client driver RPM that installs both the ODBC and JDBC driver libraries and the **vsql** client.

To install the client driver RPM package:

1. Open a Web browser and log in to the myVertica portal.

2. Click the Download tab and download the client RPM file that matches your client platform's architecture.

   Note: The 64-bit client driver RPM installs both the 64-bit and 32-bit ODBC driver libraries, so you do not need to install both on your 64-bit client system.

3. If you did not directly download the RPM on the client system, transfer the file to the client.

4. Log in to the client system as root.

5. Install the RPM package you downloaded:

   ```
   # rpm -Uvh package_name.rpm
   ```

Note: You receive one or more conflict error messages if there are existing HP Vertica client driver files on your system. This can happen if you are trying to install the client driver package on a system that has the server package installed, since the server package also includes the client drivers. In this case, you don't need to install the client drivers, and can instead use the drivers installed by the server package. If the conflict arises from an old driver installation or from a server installation for an older version, you can use the rpm command's `--force` switch to force it to overwrite the existing files with the files in the client driver package.

Once you have installed the client package, you need to create a DSN (see Creating an ODBC DSN for Linux, Solaris, AIX, and HP-UX) and set some additional configuration parameters (see

Additional ODBC Driver Configuration Settings) to use ODBC. To use JDBC, you need to modify your class path (see Modifying the Java CLASSPATH) before you can use JDBC.

You may also want to add the vsql client to your PATH environment variable so that you do not need to enter its full path to run it. You add it to your path by adding the following to the `.profile` file in your home directory or the global `/etc/profile` file:

```
export PATH=$PATH:/opt/vertica/bin
```

Installing JDBC Driver on Linux, Solaris, AIX, and HPUX

Note: The ODBC and JDBC client drivers are installed by the server `.rpm` files. If you have installed HP Vertica Analytics Platform on your Linux system for development or testing purposes, you do not need to download and install the client drivers on it—you just need to configure the drivers. To use ODBC, you need to create a DSN (see Creating an ODBC DSN for Linux, Solaris, AIX, and HP-UX). To use JDBC, you need to add the JDBC client driver to the Java CLASSPATH (see Modifying the Java CLASSPATH).

Note: For additional details about supported platforms, see Supported Platforms.

The JDBC driver is available for download from myVertica portal. There is a single `.jar` file that works on all platforms and architectures. To download and install the file:

1. Open a Web browser and log in to myVertica portal.

2. Click the Download tab and locate and download the JDBC driver.

3. You need to copy the `.jar` file you downloaded file to a directory in your Java CLASSPATH on every client system with which you want to access HP Vertica. You can either:

   - Copy the `.jar` file to its own directory (such as `/opt/vertica/java/lib`) and then add that directory to your CLASSPATH (recommended). See Modifying the Java CLASSPATH for details.

   - Copy the `.jar` file to directory that is already in your CLASSPATH (for example, a directory where you have placed other `.jar` files on which your application depends).

Note: In the directory where you copied the `.jar` file, you should create a symbolic link named `vertica_jdk_5.jar` to the `.jar` file. You can reference this symbolic link anywhere you need to use the name of the JDBC library without having to worry any future upgrade invalidating the file name. This symbolic link is automatically created on server installs. On clients, you need to create and manually maintain this symbolic link yourself if you installed the driver manually. The Installing the Client RPM on Red Hat and SUSE create this link when they install the JDBC library.

Installing ODBC/JDBC Client Drivers and vsql Client on Windows

This topic details how to download and install the HP Vertica ODBC/JDBC client drivers and vsql client for Windows systems. The installer can be run as a regular windows installer or silently. The Windows client drivers (ODBC, JDBC, etc.) are all installed using a single installer. There are 32-bit and 64-bit installers available. The 32-bit installer provides a 32-bit driver. The 64-bit installer provides both 32-bit and 64-bit drivers.

Read Driver Prerequisites before you proceed.

Note: Note: If you are uninstalling a previous release of the HP Vertica ODBC/JDBC drivers, delete any DSNs associated with those drivers before you uninstall. Windows requires that the driver files be present when a DSN is removed. If you uninstall the driver first, then the DSN cannot be removed. For releases after 5.1 you do not need to uninstall the drivers, as the installation program upgrades the existing 5.1+ drivers in place.

To Download the Windows client-drivers:

1. Open a Web browser and log in to myVertica portal.
2. Click the Download tab and select the ODBC/JDBC Windows installer that you want to install (32-bit or 64-bit) and follow the on-screen prompts to download the installer.

To Install the Windows Client-Drivers and vsql client:

1. As a Windows Administrator, double-click the installer to start the install process.
2. The introduction screen appears. Click Next to begin the installation.
3. Read the license agreement and check the appropriate radio box. Click Next to continue.
4. Optionally change the installation directory and click Next. The default directory is C:\Program Files (x86)\Vertica Systems\.
5. Select the components to install and click Next. By default all components are selected.
6. Click Install to install the options you selected.
7. Click Finish.

To Silent-Install the Windows Client-Drivers and vsql client:

1. As a Windows Administrator, open a command-line session and change directory to the folder that contains the installer.
2. Run the command vertica_client_drivers_[VERSION].exe /S /v/qn
3. The drivers are silently installed in C:\Program Files\Vertica Systems\. If you install the 32-bit drivers on a 64-bit system, then those drivers are installed to C:\Program Files (x86)\Vertica Systems\. The driver appears in the list of installed programs and is now available in the ODBC control panel.

After You install:

- The client drivers are available in the ODBC and JDBC folders of the installation directory.
- There is no shortcut for the vsql client. vsql is added to the windows PATH environment variable. Start a command window and type `vsql -?` at the command prompt to start vsql and show the help list. See vsql Notes for Windows Users for important details about using vsql in a Windows console.

You must perform an additional step for some of the client drivers before you use them:

- For ODBC, create a new Data Source Name (DSN).
- For JDBC, Modifying the Java CLASSPATH.

Modifying the Java CLASSPATH

The CLASSPATH environment variable contains the list of directories where the Java run time looks for library class files. For your Java client code to access HP Vertica, you need to add the directory where the HP Vertica JDBC `.jar` file is located.

Note: In your CLASSPATH, use the symbolic link `vertica_jdk_5.jar` that points to the JDBC library `.jar` file, rather than the `.jar` file itself. Using the symbolic link ensures that any updates to the JDBC library `.jar` file (which will use a different filename) will not invalidate your CLASSPATH setting, since the symbolic link's filename will remain the same. You just need to update the symbolic link to point at the new `.jar` file.

Linux, Solaris, AIX, HP-UX, and OS X

If you are using the Bash shell, use the `export` command to define the CLASSPATH variable:

```
# export CLASSPATH=/opt/vertica/java/lib/vertica_jdk_5.jar
```

If environment variable CLASSPATH is already defined, use the following command to prevent it from being overwritten:

```
# export CLASSPATH=$CLASSPATH:/opt/vertica/java/lib/vertica_jdk_5.jar
```

If you are using a shell other than Bash, consult its documentation to learn how to set environment variables.

You need to either set the CLASSPATH environment variable for every login session, or insert the command to set the variable into one of your startup scripts (such as `~/.profile` or `/etc/profile`).

Windows

Provide the class paths to the `.jar`, `.zip`, or `.class` files.

```
C:> SET CLASSPATH=classpath1;classpath2...
```

For example:

```
C:> SET CLASSPATH=C:\java\MyClasses\vertica_jdk_5.jar
```

As with the Linux/UNIX settings, this setting only lasts for the current session. To set the CLASSPATH permanently, set an environment variable:

1. On the Windows Control Panel, click **System**.
2. Click **Advanced** or **Advanced Systems Settings**.
3. Click **Environment Variables**.
4. Under User variables, click **New**.
5. In the Variable name box, type **`CLASSPATH`**.
6. In the Variable value box, type the path to the HP Vertica JDBC `.jar` file on your system (for example, `C:\Program Files (x86)\Vertica\JDBC\vertica_jdk_5.jar`)

Specifying the Library Directory in the Java Command

There is an alternative way to tell the Java run time where to find the HP Vertica JDBC driver other than changing the CLASSPATH environment variable: explicitly add the directory containing the `.jar` file to the java command line using either the `-cp` or `-classpath` argument. For example, on Linux, start your client application using:

```
# java -classpath /opt/vertica/java/lib/vertica_jdk_5.jar myapplication.class
```

Your Java IDE may also let you add directories to your CLASSPATH, or let you import the HP Vertica JDBC driver into your project. See your IDE documentation for details.

Installing the JDBC Driver on Macintosh OS X

To install the HP Vertica JDBC driver on your Macintosh OS X client system, download the cross-platform JDBC driver `.jar` file to your system and ensure OS X's Java installation can find it.

Downloading the JDBC Driver

To download the HP Vertica JDBC driver on Macintosh OS X:

1. On your Macintosh client system, open a browser and log into the myVertica portal.
2. Navigate to the Downloads page, scroll to the Client Software download section, and click the download link for the JDBC driver.
3. Accept the license agreement and wait for the download to complete.

Ensuring Java Can Find the JDBC Driver

In order for your Java client application to use the HP Vertica JDBC driver, the Java interpreter needs to be able to find its library file. Choose one of these methods to tell the Java interpreter where to look for the library:

- Copy the JDBC `.jar` file you downloaded to either the system-wide Java Extensions folder (`/Library/Java/Extensions`) or your user Java Extensions folder (`/Users/`*`username`*`/Library/Java/Extensions`).
- Add the directory containing the JDBC `.jar` file to the CLASSPATH environment variable (see Modifying the Java CLASSPATH).
- Specify the directory containing the JDBC `.jar` using the `-cp` argument in the Java command line you use to start your Java command line.

Installing the ODBC Driver on Macintosh OS X

The HP Vertica ODBC driver for Macintosh OS X is packaged as a gzipped tar archive (`.tar.gz`). This driver works with both 32-bit and 64-bit applications. You need to extract the library files from this archive onto your system.

Download the Driver

Follow these steps to download the HP Vertica ODBC driver for Macintosh:

1. On your Macintosh client system, open a browser and log into the the myVertica portal.
2. Navigate to the Downloads tab, scroll to the Client Software section, then click the download link for the Macintosh OS X ODBC driver package.
3. Accept the license agreement, and wait for the download to complete.

Decide Where to Install the Driver

Where you decide to install the driver depends on which users on the client Macintosh need to use the HP Vertica ODBC driver:

- If multiple users need to use the driver, install it in `/Library/ODBC/vertica`. You need to be logged into an administrator account to install the driver in this location.
- If just a single user needs to use the driver (or you do not have administrative privileges on the client OS X system), install the driver here: `/Users/`*`username`*`/Library/ODBC/vertica`.

Unpack the Driver

To unpack the driver:

1. Log into the client Macintosh either with an administrator account (if installing the driver for system-wide use) or as the user who needs to use the HP Vertica ODBC driver.
2. Open a Terminal window (in the Finder, click Applications > Utilities > Terminal).
3. Enter one of the following commands to create the target directory for the driver files:
   - To install system-wide: `mkdir -p /Library/ODBC/vertica`
   - To install for the current user: `mkdir -p ~/Library/ODBC/vertica`
4. Change to the directory you just created:
   - For system-wide installs: `cd /Library/ODBC/vertica`
   - For current user installs: `cd ~/Library/ODBC/vertica`
5. Unpack the .tar.gz file containing the ODBC driver using the command:

   ```
   tar -xzf ~/Downloads/vertica_7.0.x_odbc_mac_tar.gz
   ```

   Note: If you downloaded the driver `.tar.gz` file to a directory other than your Downloads directory, or you downloaded it using another user account, change the path in the above tar command to the path of the downloaded file.

After installing the ODBC driver, must create a DSN to be able to connect to your HP Vertica database. See Creating an ODBC DSN for Macintosh OS X Clients.

Using Legacy Drivers

The HP Vertica server supports connections from the previous version of the client drivers. For example, the HP Vertica version 5.1 server works with the 4.1 client drivers, since they were the drivers distributed with the previous version of the server. This backwards compatibility lets you upgrade your HP Vertica database first, then later upgrade your clients.

If you have not yet updated your code to work with the new version of the HP Vertica client drivers, you can continue to use the older drivers until you do. If you need to install your client application on a new client system, you can download and install the older drivers. See myVertica portal to download the installers; find installation documentation at http://www.vertica.com/documentation.

For detailed information on which the compatibility of different versions of the HP Vertica server and HP Vertica client, see Client Driver and Server Version Compatibility.

Note: The support for a previous version of the drivers is usually eliminated in the next release of HP Vertica. For example, the HP Vertica version 5.1 server does not support the version 4.0 drivers. You should update your client application to work with the new client drivers as soon as possible.

Creating an ODBC Data Source Name (DSN)

A Data Source Name (DSN) is the logical name that is used by Open Database Connectivity (ODBC) to refer to the driver and other information that is required to access data from a data source. Whether you are developing your own ODBC client code or you are using a third-party tool that needs to access HP Vertica using ODBC, you need to configure and test a DSN. The method you use depends upon the client operating system you are using.

Creating an ODBC DSN for Linux, Solaris, AIX, and HP-UX

DSNs are defined on Linux, Solaris, and other UNIX-like platforms in a text file. Your client's driver manager reads this file to determine how to connect to your HP Vertica database. The driver manager usually looks for the DSN definitions in two places:

- `/etc/odbc.ini`
- `~/.odbc.ini` (a file named `.odbc.ini` in the user's home directory)

The structure of these files is the same, only their location differs. If both files are present, the `~/.odbc.ini` file usually overrides the system-wide `/etc/odbc.ini` file.

Note: See your ODBC driver manager's documentation for details on where these files should be located and any other requirements.

odbc.ini File Structure

The `odbc.ini` is a text file that contains two types of lines:

- Section definitions, which are text strings enclosed in square brackets.
- Parameter definitions, which contain a parameter name, an equal sign, and then the parameter's value.

The first section of the file is always named [ODBC Data Sources], and contains a list of all the DSNs that the `odbc.ini` file defines. The parameters in this section are the names of the DSNs, which appear as section definitions later in the file. The value is a text description of the DSN, and has no function. For example, an `odbc.ini` file that defines a single DSN named HP VerticaDSN could have this ODBC Data Sources section:

```
[ODBC Data Sources]HPVerticaDSN = "vmartdb"
```

After the ODBC data sources section are sections that define each DSN. The name of a DSN section must match one of the names defined in the ODBC Data Sources section.

Configuring the odbc.ini file:

To create or edit the DSN definition file:

1. Using the text editor of your choice, open `odbc.ini` or `~/.odbc.ini`.
2. Create an ODBC Data Sources section and define a parameter whose name is the name of the DSN you want to create and whose value is a description of the DSN. For example, to create a DSN named VMart, you would enter:

   ```
   [ODBC Data Sources]VMart = "VMart database on HP Vertica"
   ```

3. Create a section whose name matches the DSN name you defined in step 2. In this section, you add parameters that define the DSN's settings. The most commonly-defined parameters are:

   - **Description** – Additional information about the data source.
   - **Driver** – The location and designation of the HP Vertica ODBC driver, or the name of a driver defined in the `odbcinst.ini` file (see below). For future compatibility, you should use the name of the symbolic link in the library directory (`/opt/vertica/lib` on 32-bit clients, and `/opt/vertica/lib64` on 64-bit clients), rather than the library file. For example, the symbolic link for the 64-bit ODBC driver library is:

     ```
     /opt/vertica/lib64/libverticaodbc.so
     ```

     The symbolic link always points to the most up-to-date version of the HP Vertica client ODBC library. Using the link ensures that you do not need to update all of your DSNs when you update your client drivers.
   - **Database** – The name of the database running on the server. This example uses vmartdb for the vmartdb.
   - **ServerName** — The name of the server where HP Vertica is installed. Use localhost if HP Vertica is installed on the same machine.
   - **UID** – Either the database superuser (same name as database administrator account) or a user that the superuser has created and granted privileges. This example uses the user name dbadmin.
   - **PWD** – The password for the specified user name. This example leaves the password field blank.
   - **Port** – The port number on which HP Vertica listens for ODBC connections. For example, 5433.

- **ConnSettings** – Can contain SQL commands separated by a semicolon. These commands can be run immediately after connecting to the server.
- **SSLKeyFile** – The file path and name of the client's private key. This file can reside anywhere on the system.
- **SSLCertFile** – The file path and name of the client's public certificate. This file can reside anywhere on the system.
- **Locale** – The default locale used for the session. By default, the locale for the database is en_US@collation=binary (English as in the United States of America). Specify the locale as an **ICU** Locale. See the ICU User Guide (http://userguide.icu-project.org/locale) for a complete list of parameters that can be used to specify a locale.

For example:

```
[VMart]Description = Vmart Database
Driver = /opt/vertica/lib64/libverticaodbc.so
Database = vmartdb
Servername = host01
UID = dbadmin
PWD =
Port = 5433
ConnSettings =
SSLKeyFile = /home/dbadmin/client.key
SSLCertFile = /home/dbadmin/client.crt
Locale = en_GB
```

See DSN Parameters for a complete list of parameters including HP Vertica-specific ones.

Using an odbcinst.ini File

Instead of giving the path of the ODBC driver library in your DSN definitions, you can use the name of a driver defined in the `odbcinst.ini` file. This is a useful method if you have many DSNs and often need to update them to point to new driver libraries. It also allows you to set some additional ODBC parameters, such as the threading model.

Just as in the `odbc.ini` file, `odbcinst.ini` has sections. Each section defines an ODBC driver that can be referenced in the `odbc.ini` files.

In a section, you can define the following parameters:

- **Description —** Additional information about the data source.
- **Driver —** The location and designation of the HP Vertica ODBC driver. For example: `/opt/vertica/lib64/libverticaodbc.so`

For example:

```
[HPVertica]Description = HP Vertica ODBC Driver
```

```
Driver = /opt/vertica/lib64/libverticaodbc.so
```

Then in your `odbc.ini` file, you use the name of the section you created in the `odbcinst.ini` file that describes the driver you want to use. For example:

```
[VMart]Description = HP Vertica Vmart database
Driver = HPVertica
```

If you are using the unixODBC driver manager, you should also add an ODBC section to override its standard threading settings. By default, unixODBC will serialize all SQL calls through ODBC, which prevents multiple parallel loads. To change this default behavior, add the following to your `odbcinst.ini` file:

```
[ODBC]Threading = 1
```

Configuring Additional ODBC Settings

On Linux and UNIX systems, you need to configure some additional driver settings before you can use your DSN. See Additional ODBC Driver Configuration Settings for details.

Testing a DSN Using Isql

The unixODBC driver manager includes a utility named isql, which is a simple ODBC command-line client. It lets you to connect to a DSN to send commands and receive results, similarly to vsql.

To use isql to test a DSN connection:

1. Run the following command:

   ```
   $ isql -v DSNnameSQL>
   ```

   Where *DSNname* is the name of the DSN you created.

   A connection message and a SQL prompt display. If they do not, you could have a configuration problem or you could be using the wrong user name or password.

2. Try a simple SQL statement. For example:

   ```
   SQL> SELECT table_name FROM tables;
   ```

   The isql tool returns the results of your SQL statement.

Note: If you have not set the ErrorMessagesPath in the additional driver configuration settings, any errors during testing will trigger a missing error message file ("The error message NoSQLGetPrivateProfileString could not be found in the en-US locale"). See Additional ODBC

Driver Configuration Settings for more information.

Creating an ODBC DSN for Windows Clients

Creating a DSN for Microsoft Windows clients consists of:

- Setting Up a DSN
- Testing the DSN Using Excel 2007

Setting Up a DSN

A Data Source Name (DSN) is the logical name that is used by Open Database Connectivity (ODBC) to refer to the drive and other information that is required to access data. The name is used by Internet Information Services (IIS) for a connection to an ODBC data source.

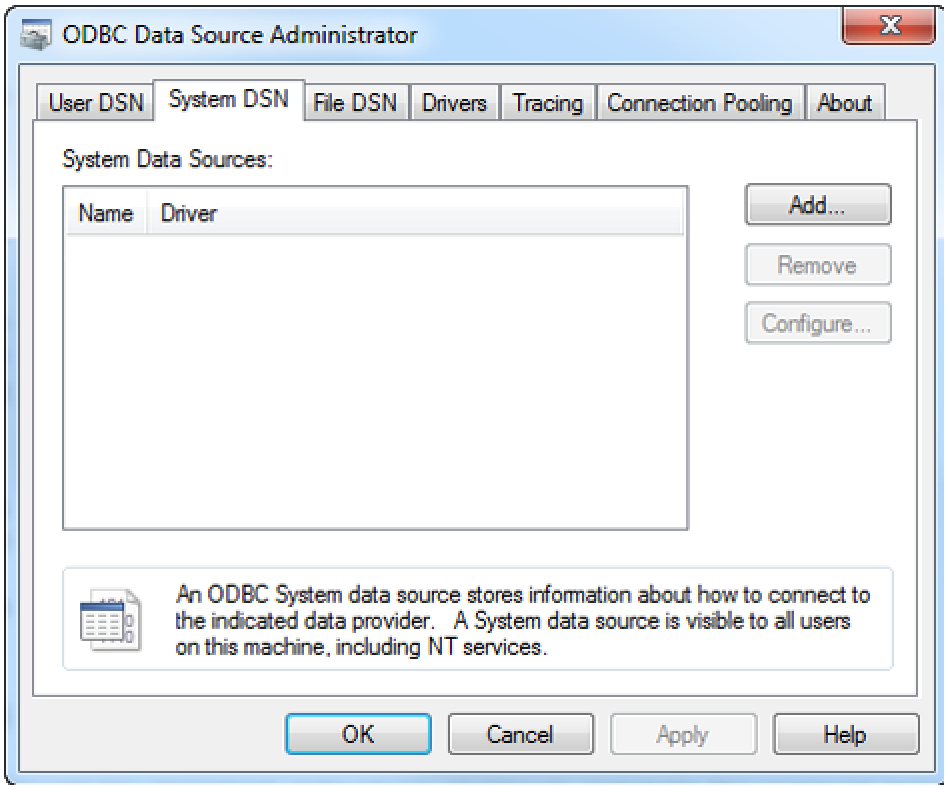
This section describes how to use the HP Vertica ODBC Driver to set up an ODBC DSN. This topic assumes that the driver is already installed, as described in Installing Client Drivers on Windows.

To set up a DSN:

1. Open the ODBC Administrator (For example, Start > Control Panel > Administrative Tools > Data Sources (ODBC)).

   Note: The method you use to open the ODBC Administrator depends on your version of Windows. Differences between Windows versions and Start Menu customizations could require a different action to open the ODBC Administrator

2. If you want all users on your client system to be able to access to the DSN for the HP Vertica database, click the **System DSN** tab.

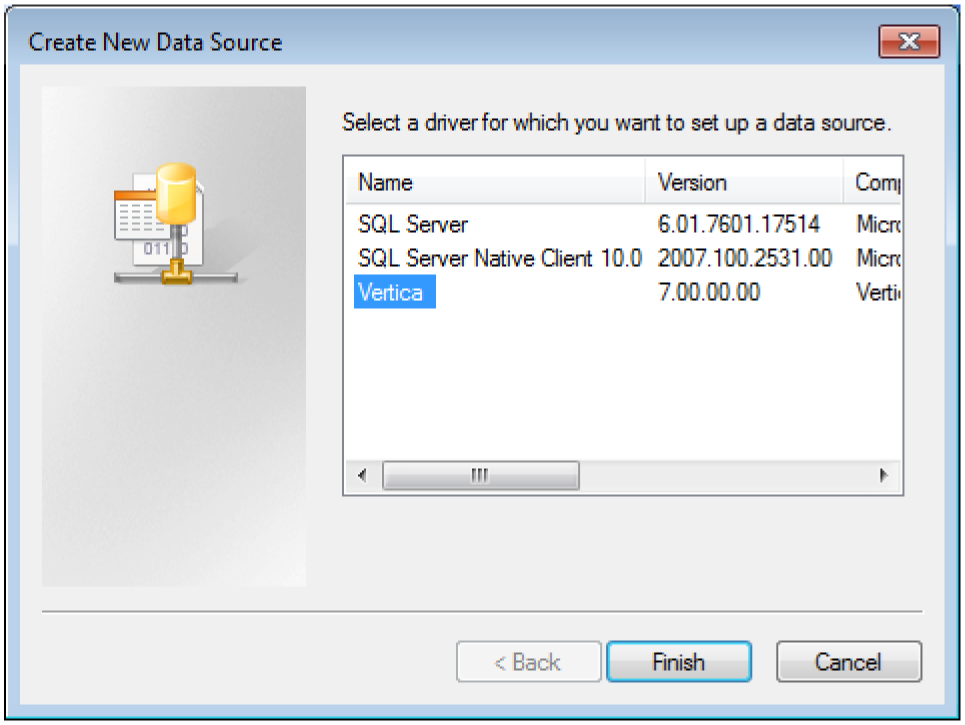


An ODBC System data source stores information about how to connect to the indicated data provider. A System data source is visible to all users on this machine, including NT services.

Otherwise, click the **User DSN** tab to create a DSN that is only usable by your Windows user account.

3. Click **Add** to create a new DSN to connect to the HP Vertica database.

4. Scroll through the list of drivers in the Create a New Data Source dialog to locate the HP Vertica driver. Select the driver, and then click **Finish**.

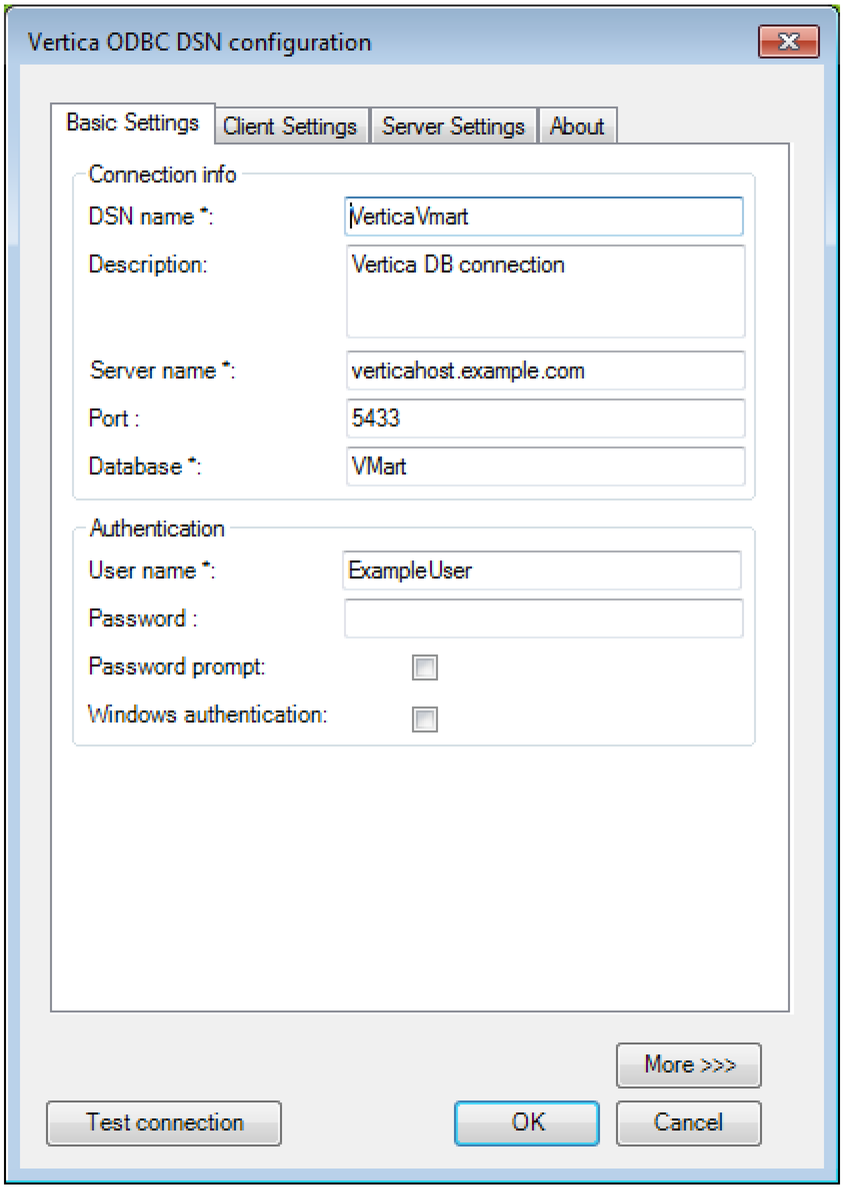


Select a driver for which you want to set up a data source.

Name	Version	Com
SQL Server	6.01.7601.17514	Micr
SQL Server Native Client 10.0	2007.100.2531.00	Micr
Vertica	7.00.00.00	Verti

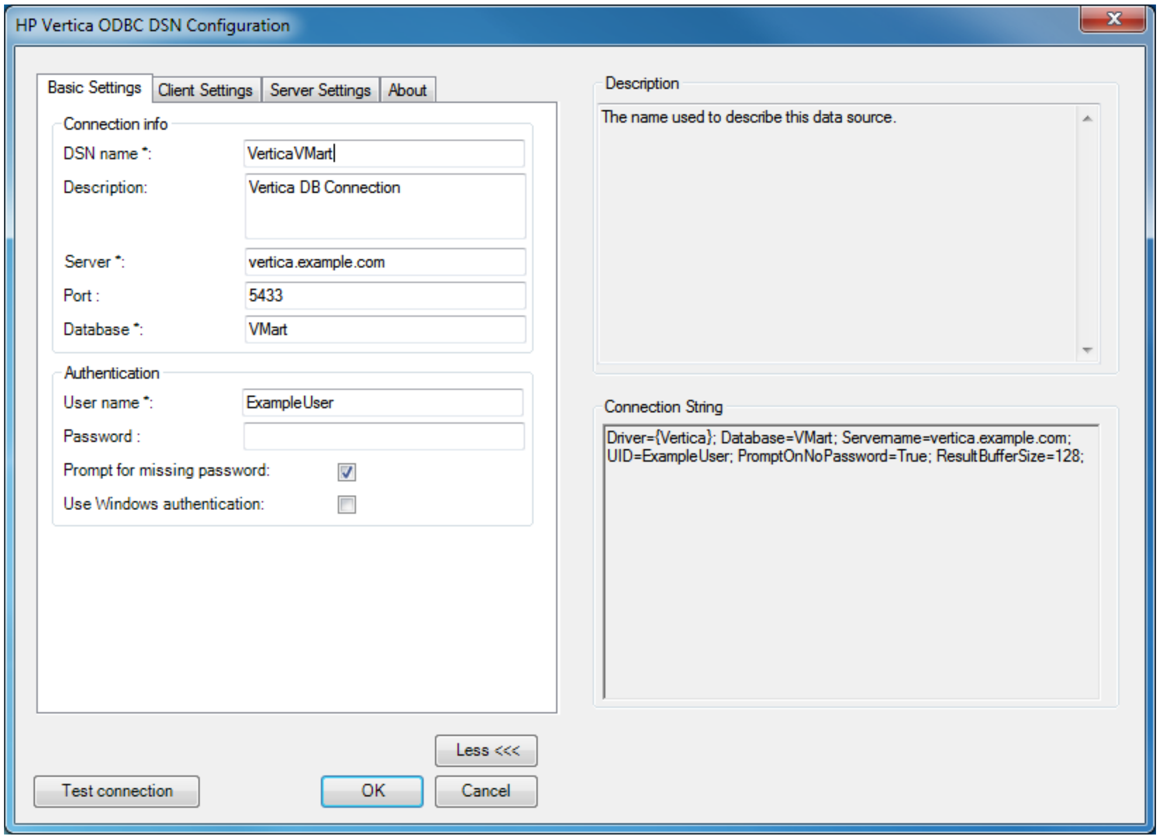
Note: If you have installed more than one version of the HP Vertica client drivers on your Windows client system, you may see multiple versions of the driver in this list. Choose the version that you know is compatible with your client application and Vertica Analytics Platform server. If you are unsure, use the latest version of the driver.

The HP Vertica ODBC DSN configuration dialog appears.



DSN name *:	VerticaVmart
Description:	Vertica DB connection
Server name *:	verticahost.example.com
Port :	5433
Database *:	VMart
User name *:	ExampleUser
Password :	
Password prompt:	☐
Windows authentication:	☐

5. Click the **More >>>** button to view a description of the field you are editing as well as the connection string defined by the DSN.



Connection info

DSN name *:	VerticaVMart
Description:	Vertica DB Connection
Server *:	vertica.example.com
Port :	5433
Database *:	VMart

Authentication

User name *:	ExampleUser
Password :	
Prompt for missing password:	✓
Use Windows authentication:	

Description

The name used to describe this data source.

Connection String

```
Driver={Vertica}; Database=VMart; Servername=vertica.example.com;
UID=ExampleUser; PromptOnNoPassword=True; ResultBufferSize=128;
```

6. Enter the information for your DSN. The following fields are required:

   - **DSN Name** — The name for the DSN. Clients will use this name to identify the DSN to which they want to connect.
   - **Server Name**— The hostname or IP address of any active node within an HP Vertica database.
   - **Database** —The name of the HP Vertica database.
   - **User Name** — The name of the user account to use when connecting to the database. This account name is used to log into the database if the application does not supply its own user name when connecting to the DSN.

   The rest of the fields are optional. See DSN Parameters for detailed information about the DSN parameters you can define.

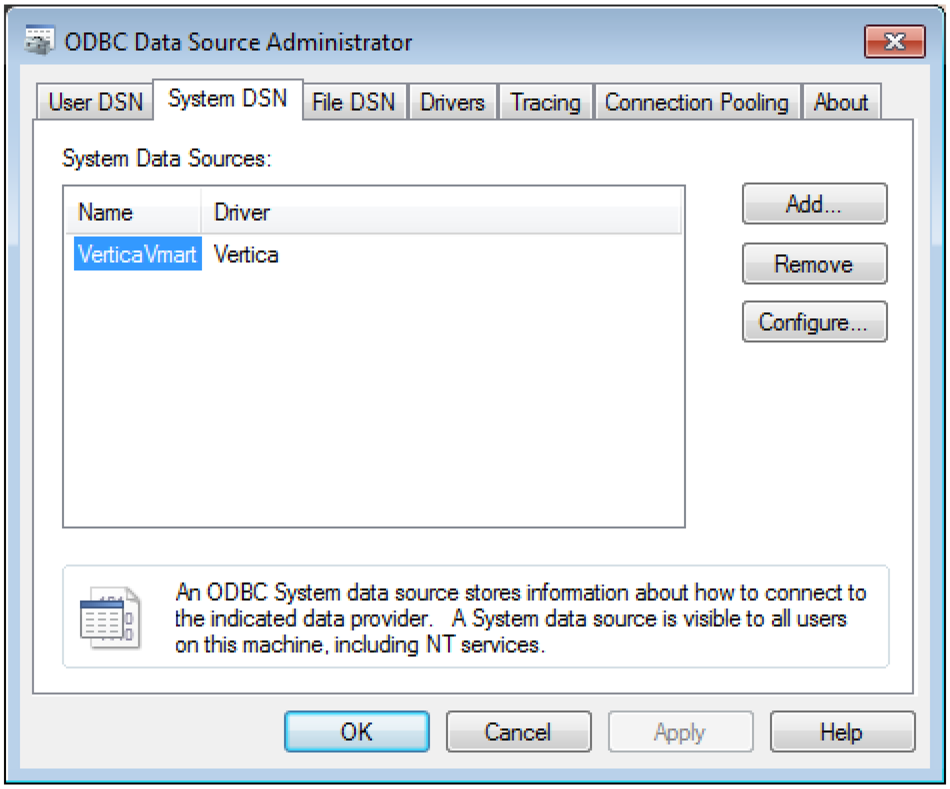
7. If you want to test your connection:

   a. Enter at least a valid **DSN name**, **Server name**, **Database**, and either **User name** or select **Windows authentication**.

   b. If you have no selected **Windows authentication**, either enter a password in the **Password** box or select **Password prompt** to have the driver prompt you for a password when connecting.

   c. Click **Test Connection**.

8. When you have finished editing and testing the DSN, click **OK**. The Vertica ODBC DSN configuration window closes, and your new DSN is listed in the ODBC Data Source Administrator window.



Name	Driver
VerticaVmart	Vertica

An ODBC System data source stores information about how to connect to the indicated data provider. A System data source is visible to all users on this machine, including NT services.

9. Click **OK** to close the ODBC Data Source Administrator.

After creating the DSN, you can test it using Microsoft Excel 2007.

Setting up a 32-Bit DSN on 64-Bit Versions of Windows

On 64-bit versions of Windows, the default ODBC Data Source Administrator creates and edits DSNs that are associated with the 64-bit HP Vertica ODBC library. Attempting to use these 64-bit DSNs with a 32-bit client application results in an architecture mismatch error. You need to create a specific 32-bit DSN for 32-bit clients by running the 32-bit ODBC Administrator usually located at:

```
c:\Windows\SysWOW64\odbcad32.exe
```

This administrator window edits a set of DSNs that are associated with the 32-bit ODBC library. You can then use your 32-bit client applications with the DSNs you create with this version of the ODBC administrator.

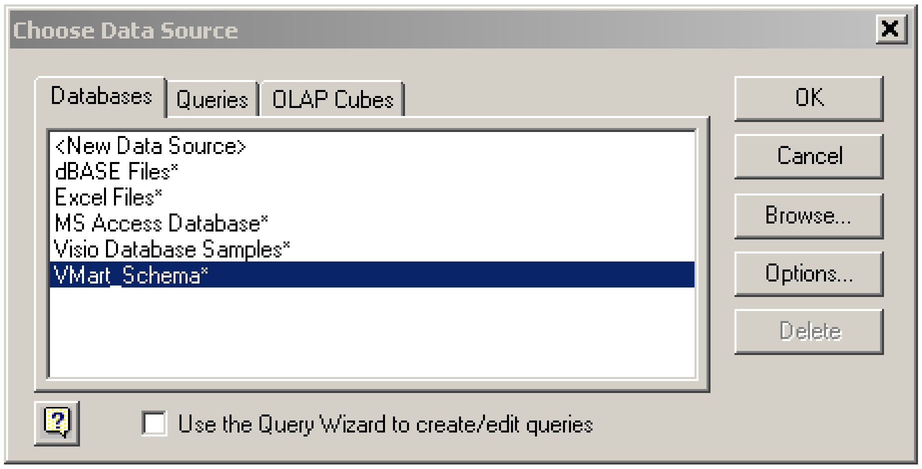
Testing a DSN Using Excel 2007

This section uses Microsoft Excel 2007 to verify that an application can connect to an ODBC data source. You can accomplish the same thing with any ODBC application.

1. Open Excel.

2. From the menu, select **Data > Get External Data > From Other Sources > From Microsoft Query**.

3. Select VMart_Schema*, make sure the "Use the Query Wizard" check box is deselected and click **OK**.



<New Data Source>
dBASE Files*
Excel Files*
MS Access Database*
Visio Database Samples*
VMart_Schema*

☐ Use the Query Wizard to create/edit queries

4. When the Add Tables window loads, click **Close**.

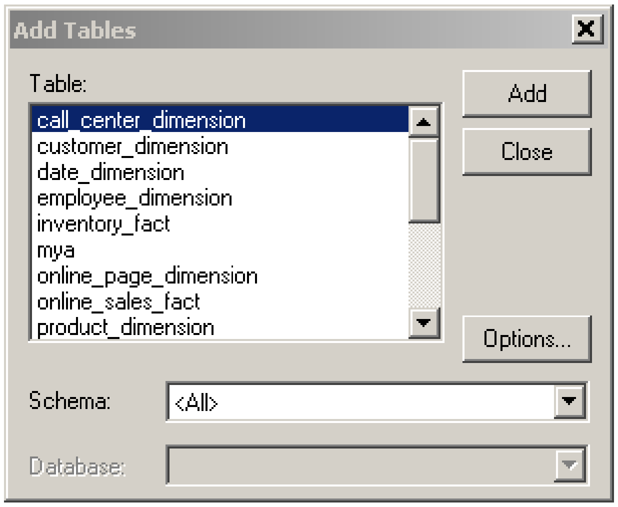
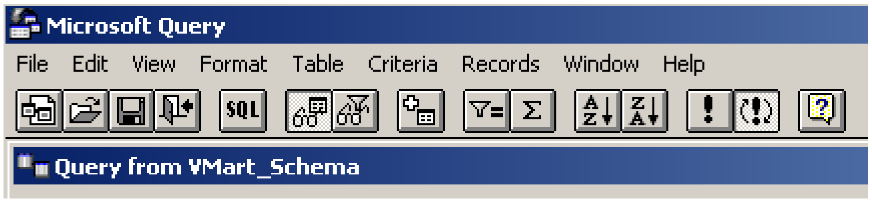


Table:
call_center_dimension
customer_dimension
date_dimension
employee_dimension
inventory_fact
mya
online_page_dimension
online_sales_fact
product_dimension

Schema: <All>

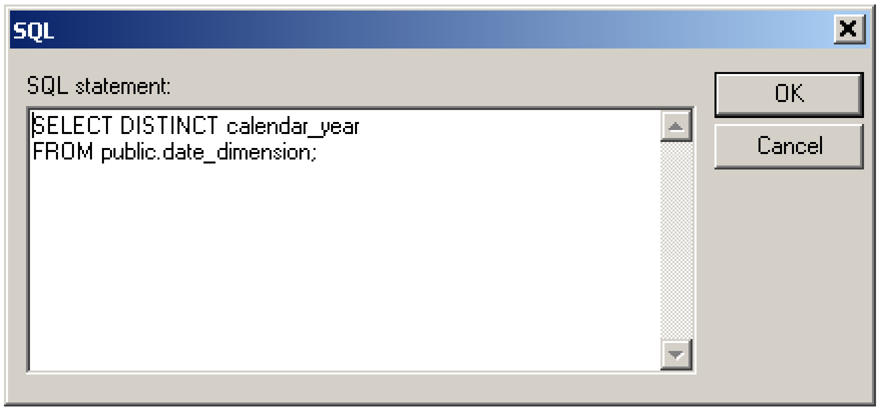
Database:

5. The Microsoft Query window opens; click the **SQL** button.



6. In the SQL window write any simple query to test your connection. This example uses the following query:

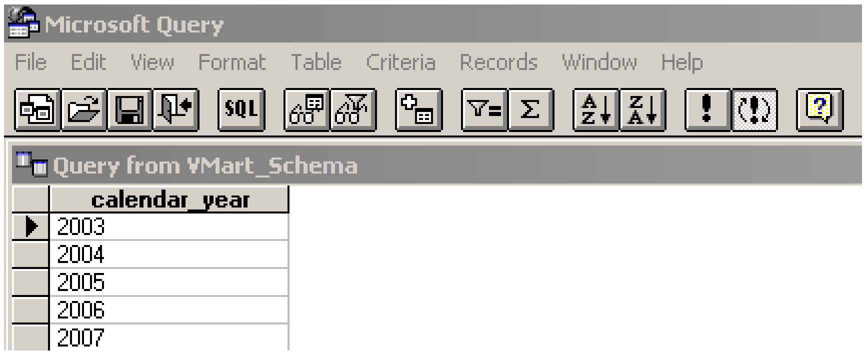
```
SELECT DISTINCT calendar_year FROM date_dimension;
```



```
SELECT DISTINCT calendar_year
FROM public.date_dimension;
```

7. If you see the caution, "SQL Query can't be represented graphically. Continue anyway?" click **OK**.

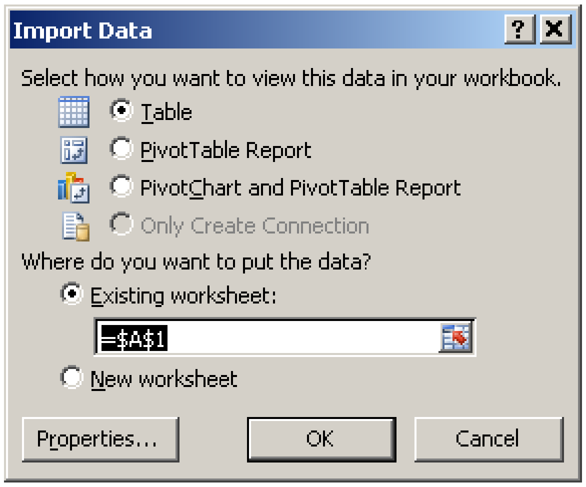
8. The data values 2003, 2004, 2005, 2006, 2007 indicate that you successfully connected to and ran a query through ODBC.



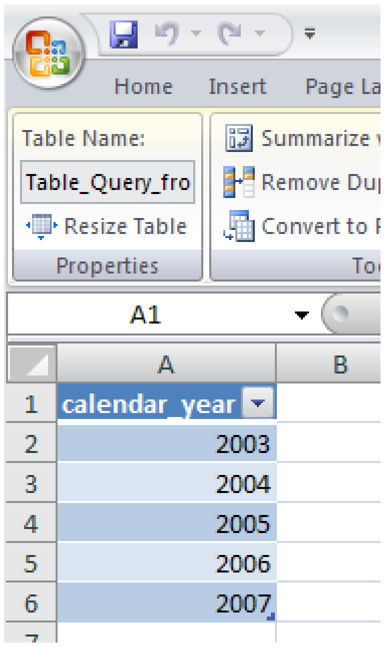
calendar_year
2003
2004
2005
2006
2007

9. Click **File > Return Data to Microsoft Office Excel**

10. In the Import Data dialog, click **OK**.


Import Data
Select how you want to view this data in your workbook.
[✓ Table]
PivotTable Report
PivotChart and PivotTable Report
Only Create Connection
Where do you want to put the data?
[✓ Existing worksheet:]
=A1
New worksheet
Properties...
OK
Cancel

The data is now available for use in an Excel worksheet.



	A	B
1	calendar_year	
2	2003	
3	2004	
4	2005	
5	2006	
6	2007	

Creating an ODBC DSN for Macintosh OS X Clients

DSNs are defined on OS X in a text file named `odbc.ini`. The ODBC driver manager in OS X reads this file to determine how to connect to your HP Vertica database. The driver manager looks for the `odbc.ini` file in two locations:

- System DSNs (those available for all users on the OS X client) are defined in `/Library/ODBC/odbc.ini`
- User DSNs (defined for just a single user) are defined in `/Users/`*`username`*`/ODBC/odbc.ini`

The structure of these files is the same, only their location differs. If both files are present, the entries in the user DSN usually override those in the system DSN.

odbc.ini File Structure

The `odbc.ini` is a text file that contains two types of lines:

- Section definitions, which are text strings enclosed in square brackets.
- Parameter definitions, which contain a parameter name, an equal sign, and then the value for the parameter.

The first section of the file is always named [ODBC Data Sources], and contains a list of all the DSNs that the `odbc.ini` file defines. The parameters in this section are the names of the DSNs, which appear as section definitions later in the file. The value is a text description of the DSN, and has no function. For example, an `odbc.ini` file that defines a single DSN named HP VerticaDSN could have this ODBC Data Sources section:

```
[ODBC Data Sources]HPVerticaDSN = VMart Database
```

Following the ODBC Data Sources section are sections that define each DSN. The name of a DSN section must match one of the parameter names defined in the ODBC Data Sources section.

Configuring the odbc.ini file:

To create or edit the DSN definition file:

1. Using the text editor of your choice, open `/Library/ODBC/odbc.ini` (to create a system DSN entry) or `/Users/`*`username`*`/ODBC/odbc.ini` to create a user DSN entry).

   Note: You must be a system administrator in order to define a system DSN.

2. Create an ODBC Data Sources section and create an entry for the DSN you want to create (for example, HPVerticaDSN). This entry establishes the name by which the new data source is referred. The value you assign to the DSN entry is just a comment that describes the DSN. For example:

   ```
   [ODBC Data Sources]HPVerticaDSN = "vmartdb connection"
   ```

3. Create a section for the DSN, and add values that define the settings needed to connect to your database. The following are the most commonly-defined parameters:

   - **Description** – Additional information about the data source.
   - **Driver** – The path to the HP Vertica ODBC driver library, or the name of a driver defined in the `odbcinst.ini` file (see below). The location of the driver depends on whether it was installed system-wide use, or just for an individual user:

If you installed the ODBC driver for system-wide use, its path should be `/Library/ODBC/vertica/lib/libverticaodbc.dylib`

If you installed the ODBC driver for just the current user, its path should be `/Users/`***`username`***`/Library/ODBC/vertica/lib/libverticaodbc.dylib`

The file name `libverticaodbc.dylib` is a symbolic link that always points to the most up-to-date version of the HP Vertica client ODBC library. Using this link ensures that you do not need to update all of your DSNs when you update your client drivers.

- **Database** – The name of the database running on the server.
- **ServerName** — The name of the server where HP Vertica is installed.
- **UID** – The HP Vertica user account to use when connecting.
- **PWD** – The password for the user given in the UID.
- **Port** – The port number on which HP Vertica listens for connections. This is usually 5433.
- **Locale** – The default locale used for the session. By default, the locale for the database is en_US@collation=binary (English as in the United States of America). Specify the locale as an **ICU** Locale. See the ICU User Guide for a complete list of parameters that can be used to specify a locale.

For example:

```
[VMart]Description = Vmart Database
Driver = /Library/ODBC/vertica/lib/libverticaodbc.dylib
Database = vmartdb
Servername = host01
UID = dbadmin
PWD = password
Port = 5433
Locale = en_GB
```

See DSN Parameters for a complete list of parameters including HP Vertica-specific ones.

Note: Instead of editing the `odbc.ini` and `odbcinst.ini` files in a text editor, you can install and use Apple's ODBC Administrator Tool.

Using an odbcinst.ini File

Instead of giving the path of the ODBC driver library in your DSN definitions, you can use the name of a driver defined in the `odbcinst.ini` file. This is a useful method if you have many DSNs, and often need to update them to point to new driver libraries. It also allows you to set some additional ODBC parameters.

Just as in the `odbc.ini` file, `odbcinst.ini` has sections. Each section defines an ODBC driver that can be referenced in the `odbc.ini` files.

In a section, you can define the following parameters:

- **Description —** Additional information about the database driver.
- **Driver —** The location of the HP Vertica ODBC driver library. For example: `/Library/ODBC/vertica/lib/libverticaodbc.dylib`

For example:

```
[HPVerticaDriver]Description = HP Vertica ODBC Driver
Driver = /Library/ODBC/vertica/lib/libverticaodbc.dylib
```

Then in your `odbc.ini` file, you use the name of the section you created in the `odbcinst.ini` file that describes the driver you want to use. For example:

```
[VMart]Description = HP Vertica Vmart database
Driver = HPVerticaDriver
```

Configuring Additional ODBC Settings

In addition to configuring the `odbc.ini` file, you need to configure some additional driver settings before you can use your DSN. See Additional ODBC Driver Configuration Settings for details.

DSN Parameters

The following tables list the connection properties you can set in the DSNs for use with HP Vertica's ODBC driver.

Required Connection Parameters

These connection parameters are the minimum requires to create a functioning DSN.

Parameters	Description	Default Value	Standard/HP Vertica
Driver	The file path and name of the driver used.	none	Standard
Database	The name of the database running on the server.	none	Standard

Parameters	Description	Default Value	Standard/HP Vertica
Servername	The host name or IP address of any active node within an HP Vertica database; for example, host01. If you use a host name whose DNS entry resolves to multiple IP addresses, the client attempts to connect to the first IP address. If the first address is unreachable, it attempts to connect to the second, then the third and so on until it either connects successfully or runs out of IP addresses. See ODBC Connection Failover for details. **Note:** You can also use the aliases "server" and "host" for this parameter.	none	Standard
UID	Either the database superuser (same name as the database administrator account) or a user that the superuser has created and granted privileges.	none	Standard

Optional Parameters

These are basic parameters that are not necessarily required.

Parameters	Description	Default Value	Standard/HP Vertica
Port	The port number on which HP Vertica listens for ODBC connections.	5433	Standard
PWD	The password for the specified user name. You may insert an empty string to leave this parameter blank.	none (login only succeeds if the user does not have a password set).	Standard

Advanced Settings

Parameters	Description	Default	Standard/HP Vertica
AutoCommit	A Boolean value that controls whether the driver automatically commits transactions after executing a **DML** statement.	`true`	Standard

Parameters	Description	Default	Standard/HP Vertica
BackupServerNode	A string containing a the host name or IP address (optionally followed by a colon and a port number) of database hosts to which the client libraries attempt to connect if the host in specified in the ServerName is unreachable. Multiple servers can be specified in a comma-separated list. When multiple host addresses are given, the client tries each in turn until one of the hosts responds or it reaches the end of the list of backup nodes. **Note:** You should limit the number of hosts you specify in this parameter. If the all hosts in the database are unreachable (due to a network issue, for example), a long list of backup servers will result in a significant delay for users before the client application returns a failure message.	none	HP Vertica

Parameters	Description	Default	Standard/HP Vertica
ConnectionLoadBalance	A Boolean that indicates whether the client is willing to accept having its connection redirected to a host in the database other than ServerNode to help spread the overhead of client connections across all hosts in the database cluster. This setting only has an effect if the server has its load balancing policy set to something other than "none." If the server does have a load balance policy set, the first node the client connects to will choose a node to handle the client connection. If this selected node is different than the node the client is connected to, the client disconnects and reconnects to the targeted node. See About Native Connection Load Balancing in the Administrator's Guide for details.	`false`	HP Vertica

Parameters	Description	Default	Standard/HP Vertica
ConnSettings	A string containing SQL commands that the driver should execute immediately after connecting to the server. This parameter is often used to configure the connection in some manner, such as setting a schema search path. **Note:** In the connection string ';' is a reserved symbol. If you need to set multiple parameters as part of ConnSettings parameter use '%3B' in place of ';'. Also use '+' for spaces.	none	HP Vertica
DirectBatchInsert	A Boolean that controls where data inserted through the connection is stored. When set to true, HP Vertica directly inserts data into **ROS** containers. Otherwise, it stores data using AUTO mode. When loading data using AUTO mode, HP Vertica inserts the data first into the **WOS**. If the WOS is full, then HP Vertica inserts the data directly into **ROS**. See the COPY statement for more details.	`false`	HP Vertica

Parameters	Description	Default	Standard/HP Vertica
DriverStringConversions	Controls whether the ODBC driver performs type conversions on strings sent between the ODBC driver and the database. Possible values are: • NONE—no conversion in either direction. this results in the highest performance. • INPUT—strings sent from the client to the server are converted, but strings sent from the server to the client are not. • OUTPUT—strings sent by the server to the client are converted, but strings sent from the client to the server are not. • BOTH—strings are converted in both directions.	`OUTPUT`	HP Vertica
Locale	The locale used for the session. Specify the locale as an **ICU** Locale. See the ICU User Guide (http://userguide.icu-project.org/locale) for a complete list of parameters that can be used to specify a locale.	`en_US@collation=binary` (English as in the United States of America)	HP Vertica

Parameters	Description	Default	Standard/HP Vertica
PromptOnNoPassword	Controls whether users are prompted to enter a password if none is supplied by the connection string or DSN used to connect to HP Vertica. See Prompting Windows Users for Passwords. **Note:** This setting only has an effect on Windows platforms. It is ignored by the drivers on other platforms.	`false`	HP Vertica
ReadOnly	A true/false value that controls whether the connection can only read data from HP Vertica.	`false`	HP Vertica
ResultBufferSize	Size of memory buffer for the large result sets in streaming mode. **Note:** This parameter was previously called MaxMemoryCache	`131072 (128KB)`	HP Vertica
TransactionIsolation	Sets the transaction isolation for the connection. Valid values are: • Read Committed • Serializable • Server Default See Changing Transaction Isolation Levels in the Administrator's Guide for an explanation of transaction isolation.	`Server Default`	HP Vertica

Identification

Parameters	Description	Default	Standard/HP Vertica
Description	An optional description for the DSN entry. Insert an empty string to leave the description empty.	none	Standard
Label / SessionLabel	Sets a label for the connection on the server. This value appears in the session_id column of the V_MONITOR.SESSIONS system table. **Note:** Label and SessionLabel are synonyms. They can be used interchangeably.	none	HP Vertica

Encryption

Parameters	Description	Default	Standard/HP Vertica
SSLMode	Controls whether the connection to the database uses SSL encryption. Valid values are: • require—Requires the server to use SSL. If the server cannot provide an encrypted channel, the connection fails. • prefer—Prefers the server to use SSL. If the server does not offer an encrypted channel, the client requests one. Note that the first connection attempt to the database tries to use SSL. If that fails, a second connection is attempted over a clear channel. • allow—Makes a connection to the server whether the server uses SSL or not. Note that the first connection attempt to the database is attempted over a clear channel. If that fails, a second connection is attempted over SSL. • disable—Never connects to the server using SSL. This setting is typically used for troubleshooting	`prefer`	HP Vertica

Third-Party Compatibility

Parameters	Description	Default	Standard/HP Vertica
ColumnsAsChar	How character column types are reported when the driver is in Unicode mode. When set to false, the ODBC driver reports the data type of character columns as WCHAR. If you set ColumnsAsChar to true, the driver identifies character column as CHAR. This setting is normally used for compatibility with some third-party clients such as Informatica.	false	HP Vertica
ThreePartNaming	A Boolean that controls how the database metadata APIs reports the catalog name. When set to true, the database name is returned as the catalog name in the database metadata. When set to false, NULL is returned as the catalog name. Enable this option of your client software expects to be able to get the catalog name from the database metadata and use it as part of a three-part name reference.	`false` on Unix clients `true` on Windows clients	HP Vertica

Kerberos Connection Parameters

Use the following parameters for client authentication using Kerberos.

Parameters	Description	Default	Standard/HP Vertica
KerberosServiceName	Provides the service name portion of the HP Vertica Kerberos principal; for example: `vertica/host@EXAMPLE.COM`	vertica	HP Vertica
KerberosHostname	Provides the instance or host name portion of the HP Vertica Kerberos principal; for example: `vertica/host@EXAMPLE.COM`	Value specified in the servername connection string property	HP Vertica

See Also

- Additional ODBC Driver Configuration Settings

Setting DSN Parameters

The parameters in the following tables are common for all user and system DSN entries. The examples provided are for Windows clients.

To edit DSN parameters:

- On UNIX and Linux client platforms, you can edit the `odbc.ini` file. (See Creating an ODBC DSN for Linux, Solaris, AIX, and HP-UX.) The location of this file is specific to the driver manager.

- On Windows client platforms, you can edit some DSN parameters using the HP Vertica ODBC client driver interface. See Creating an ODBC DSN for Windows Clients.

  You can also edit the DSN parameters directly by opening the DSN entry in the Windows registry (for example, at `HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\`*`DSNname`*). Directly editing the registry can be risky, so you should only use this method for parameters cannot be set through the ODBC driver's user interface, or via your client code.

- You can set parameters in the connection string when opening a connection using the `SQLDriverConnect()` function:

  ```
  sqlRet = SQLDriverConnect(sql_hDBC, 0, (SQLCHAR*)"DSN=DSNName;Locale=en_GB", SQL_NTS,
  szDNS, 1024,&nSize, SQL_DRIVER_NOPROMPT);
  ```

  Note: In the connection string ';' is a reserved symbol. If you need to set multiple parameters as part of ConnSettings parameter use '%3B' in place of ';'. Also use '+' instead of spaces.

  For Example:

  ```
  sqlRet = SQLDriverConnect(sql_hDBC, 0, (SQLCHAR*)"DSN=HP VerticaSQL;ConnSettings=set+sea
  rch_path+to+a,b,c%3Bset+locale=ch;SSLMode=prefer", SQL_NTS,
  szDNS, 1024,&nSize, SQL_DRIVER_NOPROMPT);
  ```

- Your client code can retrieve DSN parameter values after a connection has been made to HP Vertica using the `SQLGetConnectAttr()` and `SQLGetStmtAttr()` API calls. Some parameters can be set and using `SQLSetConnectAttr()` and `SQLSetStmtAttr()`.

  For details of the list of HP Vertica-specific parameters see HP Vertica-specific ODBC Header File.

Upgrading the HP Vertica Client Drivers

The HP Vertica client driver are usually updated for each new release of the HP Vertica server. The client driver installation packages include the version number of the corresponding HP Vertica server release. Usually, the drivers are forward-compatible with the next release, so your client applications are still be able to connect using the older drivers after you upgrade to the next version of HP Vertica Analytics Platform server. See Client Driver and Server Version Compatibility for details on which client driver versions work withe each version of HP Vertica server.

You should upgrade your clients as soon as possible after upgrading your server, to take advantage of new features and to maintain maximum compatibility with the server.

To upgrade your drivers, follow the same procedure you used to install them in the first place. The new installation will overwrite the old. See the specific instructions for installing the drivers on your client platform for any special instructions regarding upgrades.

Note: Installing new ODBC drivers does not alter existing DSN settings. You may need to change the driver settings in either the DSN or in the `odbcinst.ini` file, if your client system uses one. See Creating an ODBC Data Source Name for details.

Additional ODBC Driver Configuration Settings

On Linux and UNIX platforms, in addition to the DSN settings, you need to provide some additional settings to configure the HP Vertica ODBC client driver. These settings control the following:

- The text encoding used by the driver manager (for example, UTF-8 or UTF-16).
- The location of the directory containing the HP Vertica ODBC driver's error message files.
- Whether and how the ODBC driver logs messages.

Note: Most of the additional driver configuration settings are automatically set on Windows platforms. The Windows ODBC driver's DSN Configuration dialog lets you control whether the ODBC driver logs messages. On Linux/UNIX systems, you must supply the additional configuration settings before the ODBC drivers can function properly.

The topics in this section describe these settings in greater detail.

Location of the Additional Driver Settings

Where the additional driver settings are stored depends on your client platform:

- On Linux and UNIX platforms, the settings are contained in a text file named `vertica.ini` (although you can choose a different filename). You tell the HP Vertica ODBC driver where to find this file using an environment variable named VERTICAINI.
- On Windows platforms, the additional settings are set using the ODBC Data Source Configuration window. The values for the settings are stored in the Windows registry under the path HKEY_LOCAL_MACHINE\SOFTWARE\Vertica\Driver.

Creating a vertica.ini File

There is no standard location for the `vertica.ini` file—you can store the file anywhere that it is convenient for you on your client system. One possible location is in the `/etc` directory if you have multiple users on your client system that need to access it, or have a `vertica.ini` file in each user's home directory so users can alter their own settings. Wherever you store it, be sure users have read access to the file.

The format of the `vertica.ini` file is similar to the `odbc.ini` file, with a section followed by parameter definitions. Unlike the `odbc.ini` file, `vertica.ini` contains a single section named Driver:

```
[Driver]
```

Following the section definition, you add setting definitions, one per line. A setting definition consists of the setting name, followed by an equal sign (=), followed by the value. The value does not need quotes. For example, to set the ODBCInstLib setting, you add a line like this:

```
ODBCInstLib=/usr/lib64/libodbcinst.so
```

See Additional Parameter Settings for a list of the additional settings.

Required Settings

You must configure two settings in order for the ODBC driver to work correctly:

- ErrorMessagesPath
- ODBCInstLib (unless the driver manager's installation library is in a directory listed in the LD_LIBRARY_PATH or LIB_PATH environment variables).

Also, if your driver manager does not use UTF-8 encoding, you need to set DriverManagerEncoding to the proper encoding.

Setting the VERTICAINI Environment Variable

You must set an environment variable named VERTICAINI to the absolute path of the `vertica.ini` file. The HP Vertica ODBC driver uses this variable to find the settings.

Where you set this variable depends on whether users on your client system need to have separate `vertica.ini` files. If you want to have a single, system-wide `vertica.ini` file, you can add a command to set VERTICAINI in `/etc/profile` or some other system-wide environment file. For example:

```
export VERTICAINI=/etc/vertica.ini
```

If users need individual `vertica.ini` files, set VERTICAINI in their `~/.profile` or similar configuration file. For example:

```
export VERTICAINI=~/.vertica.ini
```

On Macintosh OS X client systems, you can set the VERTICAINI environment variable in each user's `~/.MacOSX/environment.plist` file. See the Environment Variables entry in the Apple's Developer's Library for more information.

Example vertica.ini File

The following example `vertica.ini` file configures the ODBC driver to:

- use the 64-bit UnixODBC driver manager.
- get its error messages from the standard HP Vertica 64-bit ODBC driver installation directory.
- log all warnings and more severe messages to log files stored in the temporary directory.

```
[Driver]
DriverManagerEncoding=UTF-16
```

```
ODBCInstLib=/usr/lib64/libodbcinst.so
ErrorMessagesPath=/opt/vertica/lib64
LogLevel=4
LogPath=/tmp
```

Additional Parameter Settings

The following parameters can be set for the HP Vertica client drivers.

Logging Settings

These parameters control how messages between the client and server are logged. None of these settings are required. If they are not set, then the client library does not log any messages. They apply to both ADO.NET and ODBC.

- **LogLevel**—The severity of messages that are logged between the client and the server. The valid values are:
  - 0—No logging
  - 1—Fatal errors
  - 2—Errors
  - 3—Warnings
  - 4—Info
  - 5—Debug
  - 6—Trace (all messages)

  The value you specify for this setting sets the minimum severity for a message to be logged. For example, setting LogLevel to 3 means that the client driver logs all warnings, errors, and fatal errors.
- **LogPath**—The absolute path of a directory to store log files . For example: `/var/log/verticaodbc`
- **LogNamespace**—Limits logging to messages generated by certain objects in the client driver.

Note: These settings are also available for the HP Vertica JDBC driver through connection properties. See Connection Properties for details.

ODBC-specific Settings

The following settings are used only by the HP Vertica ODBC client driver.

- **DriverManagerEncoding**—The UTF encoding standard that the driver manager uses. This setting needs to match the encoding the driver manager expects. The available values for this setting are:
  - UTF-8
  - UTF-16 (usually used by unixODBC)
  - UTF-32 (usually used by iODBC)

  See the documentation for your driver manager to find the correct value for this setting.

  Note: While both UTF-16 and UTF-8 are valid settings for DataDirect, Vertica recommends that you set the DataDirect driver manager encoding to UTF-16.

  If you do not set this parameter, the ODBC driver defaults to the value shown in the following table. If your driver manager uses a different encoding, you must set this value for the ODBC driver to be able to work.

Client Platform	Default Encoding
AIX 32-bit	UTF-16
AIX 64-bit	UTF-32
HPUX 32-bit	UTF-32
HPUX 64-bit	UTF-32
Linux x86 32-bit	UTF-32
Linux x86 64-bit	UTF-32
Linux Itanium 64-bit	UTF-32
OS X	UTF-32
Solaris x86 32-bit	UTF-32
Solaris x86 64-bit	UTF-32
Solaris SPARC 32-bit	UTF-32
Solaris SPARC 64-bit	UTF-32
Windows 32-bit	UTF-16
Windows 64-bit	UTF-16

- **ErrorMessagesPath**—The absolute path to the parent directory that contains the HP Vertica client driver's localized error message files. These files are usually stored in the same directory as the HP Vertica ODBC driver files.

  > **Note:** This setting is required. If you do not set it, then any error the ODBC driver encounters will result in an error message about a missing `ODBCMessages.xml` file.

- **ODBCInstLib**—The absolute path to the file containing the ODBC installer library (ODBCInst). This setting is required if the directory containing this library is not set in the LD_LIBRARY_PATH or LIB_PATH environment variables. The library files for the major driver manager are:

  - UnixODBC: libodbcinst.so
  - iODBC: libiodbcinst.so (libiodbcinst.2.dylib on OS X)
  - DataDirect: libodbcinst.so

> **Note:** On AIX platforms, you need give the path to the library archive, followed by the name of the library enclosed in parenthesis. For example:
> `ODBCInstLib=/usr/lib64/libodbcinst.a(libodbcinst.so.1)`

ADO.NET-specific Settings

This setting applies only to the ADO.NET client driver:

C#PreloadLogging—Tells the HP Vertica ADO.NET driver to begin logging as soon as possible, before the driver has fully loaded itself. Normally, logging only starts after the driver has fully loaded. Valid values for this setting are:

- 0—Do not start logging before the driver has loaded.
- 1—Start logging as soon as possible.

Programming ODBC Client Applications

HP Vertica provides an Open Database Connectivity (ODBC) driver that allows applications to connect to the HP Vertica database. This driver can be used by custom-written client applications that use the ODBC API to interact with HP Vertica. ODBC is also used by many third-party applications to connect to HP Vertica, including business intelligence applications and extract, transform, and load (ETL) applications.

This section details the process for configuring the HP Vertica ODBC driver. It also explains how to use the ODBC API to connect to HP Vertica in your own client applications.

This section assumes that you have already installed the ODBC libraries on your client system. If you have not, see Client Driver Install Procedures.

ODBC Architecture

The ODBC architecture has four layers:

- **Client Application**

  Is an application that opens a data source through a Data Source Name (DSN). It then sends requests to the data source, and receives the results of those requests. Requests are made in the form of calls to ODBC functions.

- **Driver Manager**

  Is a library on the client system that acts as an intermediary between a client application and one or more drivers. The driver manager:

  - Resolves the DSN provided by the client application.
  - Loads the driver required to access the specific database defined within the DSN.
  - Processes ODBC function calls from the client or passing them to the driver.
  - Retrieves results from the driver.
  - Unloads drivers when they are no longer needed.

  On Windows and Mac client systems, the driver manager is provided by the operating system. On Linux and UNIX systems, you usually need to install a driver manager. See ODBC Prerequisites for a list of driver managers that can be used with HP Vertica on your client platform.

- **Driver**

  A library on the client system that provides access to a specific database. It translates requests into the format expected by the database, and translates results back into the format required by the client application.

- **Database**

  The database processes requests initiated at the client application and returns results.

ODBC Feature Support

The ODBC driver for HP Vertica supports the most of the features defined in the Microsoft ODBC 3.5 specifications. The following features are *not* supported:

- Updatable result sets
- Backwards scrolling cursors
- Cursor attributes
- More than one open statement per connection. For example you cannot execute a new statement while another statement has a result set open. If you need to execute multiple statements at once, open multiple database connections.
- Keysets
- Bookmarks

The HP Vertica ODBC driver accurately reports its capabilities. If you need to determine whether it complies with a specific feature, you should query the driver's capabilities directly using the `SQLGetInfo()` function.

Updating ODBC Client Code From Previous Driver Versions

In HP Vertica Version 5.1, the client drivers were rewritten to improve standards compliance and reliability. As a result, some HP Vertica-specific features and past incompatibilities have been eliminated. You must update any client code written for versions of the HP Vertica ODBC driver earlier than 5.1 to work with the newer drivers. The following topics give you an overview of the necessary code changes.

DSN Parameter Changes

A number of HP Vertica-specific DSN parameters have been eliminated or changed.

Removed DSN Parameters

The following parameters are no longer available in the 5.1 ODBC driver.

Parameter	Description
BatchInsertEnforceLength	Batch inserts now behave the same way all other inserts behave. If a piece of data is too wide for its column, the row will be rejected. To avoid having rows rejected, either truncate the data yourself or use the COPY statement directly, which defaults to truncating data.
BinaryDataTransfer	All data is now transferred using NATIVE VARCHAR.
BoolAsChar	Boolean columns can no longer be bound to SQL_CHAR values. Use SQL_BIT values instead.
Debug	Use the LogLevel parameter instead.
LRSPath and LRSStreaming	The ODBC driver now always streams data. It does not cache data on the local disk.
SupressWarnings	Removed to prevent clients from ignoring warnings.
WideCharSizeIn and WideCharSizeOut	These options are now set using the DriverManagerEncoding option in the `vertica.ini` file.

Changed DSN Parameters

The following DSN parameters have changed since the previous version of the ODBC driver.

Parameter	Description
MaxMemoryCache	This parameter is now named ResultBufferSize to make its name consistent across all HP Vertica client drivers.

New DSN Parameter

The following DSN parameters are new in the version 5.1 ODBC driver.

Parameter	Description
DriverStringConversions	Determines whether strings sent between the server and the ODBC client are converted.
ThreePartNaming	Controls whether the database metadata uses the database name as the catalog name. This parameter is only used for backwards compatibility with some client software that expects a non-null catalog name.

New DSN Parameter Alias

Parameter	Description
Password	An alias for PWD.
SessionLabel	This is a new alias for the existing Label parameter that lets you assign a label to an ODBC connection.
UserName	An alias for UID.

Function Changes

The following functions have changed their behavior in the version 5.1 drivers.

- `SQLGetInfo()` now returns the file name of the ODBC library file when queried for the SQL_DRIVER_NAME. Previous versions would return the brand name of the driver, which is not part of the ODBC specifications.
- In previous versions of the ODBC driver, passing the `SQLGetInfo` function SQL_PARAM_ARRAY_SELECTS returned SQL_PAS_BATCH, indicating that the driver returns a batch for each set of parameters in a SELECT statement. This return value was incorrect. The driver now correctly returns SQL_PAS_NO_SELECT, which indicates that the driver does not support SELECT queries using arrays of parameters.
- For better compatibility with the ODBC standards, the time data type in the ODBC driver no longer contain fractions of a second. In the new driver, functions that convert time values to strings (for example, a `SQLBindCol()` call to bind a SQL_TYPE_TIME to a SQL_C_CHAR value) no longer add fractional second values to the string. Earlier versions of the driver would return the fractions of a second that the HP Vertica database stores.
- `SQLBindParameter()` now requires that its column-size argument for variable-width columns (such as SQL_CHAR, SQL_VARCHAR, SQL_VARBINARY) be non-zero as specified in the ODBC standards. The implementation of this function in previous versions of the driver allowed a non-standard zero value to indicate the column width was a default width value.
- The driver is now more strict about converting from character data types to numerical data types. In previous drivers, any portion of the character string that could not be converted to a numeric value was ignored. The new driver returns an error if any portion of the string cannot be converted. For example, asking the previous driver to insert the character string "3 days" to an integer column resulted in the value 3 being stored in the column. Now, the driver returns an error when attempting to store this character value in an integer column.
- Functions that return result sets containing catalog metadata now return SQL_WVARCHAR columns instead of SQL_VARCHAR columns. This is standard behavior for ODBC 3.52 drivers that are Unicode capable.

Removed Functions

The LCOPY function has been removed from the version 5.1 driver. You should instead use the LOCAL option of the COPY SQL statement to copy data from the client system to the server. See Streaming Data From the Client Using COPY LOCAL.

Interval and TimeStamp Changes

When you call the `SQLBindParameter()` function to bind a SQL_C_INTERVAL value (for example, SQL_INTERVAL_STRUCT) to an INTERVAL column after calling `SQLPrepare()`, the interval leading precision is now reset to the default value of 2. Earlier versions of the driver did not reset the leading precision as called for in the ODBC standards. If you want your interval value to have a greater leading precision, your client application can take either of the following steps:

- Call `SQLBindParamter()` to bind the value to the INTERVAL column before calling `SQLPrepare()`. The interval precision is not reset unless you call `SQLBindParameter()` after you have called `SQLPrepare()`.
- Reset the leading precision by using the `SQLSetDescField()` function to set SQL_DESC_DATETIME_INTERVAL_PRECISION to whatever value you want.

`SQLDescribeCol()` now returns a new width for interval data types. For example, when passed SQL_INTERVAL_SECOND, the previous driver would return 21 bytes. The new driver returns 16.

In general, the new drivers are more strict regarding interval values, and throw errors when the old drivers silently truncated values. If your application throws an exception when dealing with an interval, your first debugging step is to make sure the values you are inserting are the correct width and type.

Note: The units interval style is not supported. Do not use the SET INTERVALSTYLE statement to change the interval style in your client applications.

`SQLBindParameter()` is more strict about the number of digits you supply in fractions of a second in a timestamp. For example, the following call generates an error:

```
SQL_TIMESTAMP_STRUCT ts;ts.fraction = 123456; //represents the fraction 0.000123456
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_TIMESTAMP, SQL_TYPE_TIMESTAMP, 0, 6, (S
QLPOINTER)&ts, 0, NULL);
SQLExecute(hstmt);
```

The error occurs because the fractional value represents more than six digits. Instead you would need to change the SQLBindParameter call to allow 9 digits in the fraction:

```
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_TIMESTAMP, SQL_TYPE_TIMESTAMP, 0, 9, (S
QLPOINTER)&ts, 0, NULL);
```

Note that the fraction will be truncated to 0.000123, since timestamps do not have nanosecond precision. But you still need to allow for all of the fractional digits to be inserted into the timestamp.

New Additional Driver Information

The new HP Vertica version 5.1 ODBC driver has some additional configuration settings not covered by the standard ODBC.INI file. use a configuration file named `vertica.ini` on Linux, AIX, Solaris, and HP-UX. It controls several features of the ODBC driver. For more information, see Additional ODBC Driver Configuration Settings.

HP Vertica-specific ODBC Header File

The HP Vertica ODBC driver provides a C header file named `verticaodbc.h` that defines several useful constants that you can use in your applications. These constants let you access and alter HP Vertica-specific settings.

This file's location depends on your client operating system:

- `/opt/vertica/include` on Linux and UNIX systems.
- `C:\Program Files (x86)\Vertica\ODBC\include` on Windows systems.

The constants defined in this file are listed below.

Parameter	**Description**	**Associated Function**
`SQL_ATTR_VERTICA_RESULT_BUFFER_SIZE`	Sets the size of the buffer used when retrieving results from the server.	`SQLSetConnectAttr()` `SQLGetConnectAttr()`

Parameter	Description	Associated Function
`SQL_ATTR_VERTICA_DIRECT_BATCH_INSERT`	Determines whether a batch is inserted directly into the ROS (1) or using AUTO mode (0). By default batches are inserted using AUTO. When loading data using AUTO mode, HP Vertica inserts the data first into the **WOS**. If the WOS is full, then HP Vertica inserts the data directly into **ROS**. See the COPY statement for more details.	`SQLSetConnectAttr()` `SQLSetStmtAttr()` `SQLGetConnectAttr()` `SQLGetStmtAttr()`
`SQL_ATTR_VERTICA_LOCALE`	Changes the locale from en_US@collation=binary to the ICU locale specified. See Setting the Locale for ODBC Sessions for an example of using this parameter.	`SQLSetConnectAttr()` `SQLGetConnectAttr()`

Connecting to HP Vertica

The first step in any ODBC application is to connect to the database. When you create the connection to a data source using ODBC, you use the name of the DSN that contains the details of the driver to use, the database host, and other basic information about connecting to the data source.

There are 4 steps your application needs to take to connect to a database:

1. Call `SQLAllocHandle()` to allocate a handle for the ODBC environment. This handle is used to create connection objects and to set application-wide settings.

2. Use the environment handle to set the version of ODBC that your application wants to use. This ensures that the data source knows which API your application will use to interact with it.

3. Allocate a database connection handle by calling `SQLAllocHandle()`. This handle represents a connection to a specific data source.

4. Use the `SQLConnect()` or `SQLDriverConnect()` functions to open the connection to the database.

   Note: If you specify a locale either in the connection string or in the DSN, the call to the connection function returns SQL_SUCCESS_WITH_INFO on a successful connection, with messages about the state of the locale.

When creating the connection to the database, use `SQLConnect()` when the only options you need to set at connection time is the username and password. Use `SQLDriverConnect()` when you want to change connection options, such as the locale.

The following example demonstrates connecting to a database using a DSN named ExampleDB. After it creates the connection successfully, this example simply closes it.

```
// Demonstrate connecting to Vertica using ODBC.
// Standard i/o library
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// SQL include files that define data types and ODBC API
// functions
#include <sql.h>
#include <sqlext.h>
#include <sqltypes.h>
int main()
{
    SQLRETURN ret;   // Stores return value from ODBC API calls
    SQLHENV hdlEnv;  // Handle for the SQL environment object
    // Allocate an a SQL environment object
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }

    // Set the ODBC version we are going to use to
    // 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
            (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(!SQL_SUCCEEDED(ret)) {
         printf("Could not set application version to ODBC 3.\n");
         exit(EXIT_FAILURE);
    } else {
         printf("Set application version to ODBC 3.\n");
    }
    // Allocate a database handle.
    SQLHDBC hdlDbc;
```

```
      ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
      if(!SQL_SUCCEEDED(ret)) {
           printf("Could not allocate database handle.\n");
           exit(EXIT_FAILURE);
      } else {
           printf("Allocated Database handle.\n");
      }
     // Connect to the database using
     // SQL Connect
     printf("Connecting to database.\n");
     const char *dsnName = "ExampleDB";
     const char* userID = "ExampleUser";
     const char* passwd = "password123";
     ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
         SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
         (SQLCHAR*)passwd, SQL_NTS);
     if(!SQL_SUCCEEDED(ret)) {
         printf("Could not connect to database.\n");
         exit(EXIT_FAILURE);
     } else {
         printf("Connected to database.\n");
     }
     // We're connected. You can do real
     // work here

     // When done, free all of the handles to close them
     // in an orderly fashion.
     printf("Disconnecting and freeing handles.\n");
     ret = SQLDisconnect( hdlDbc );
     if(!SQL_SUCCEEDED(ret)) {
         printf("Error disconnecting from database. Transaction still open?\n");
         exit(EXIT_FAILURE);
     }

     SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
     SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
     exit(EXIT_SUCCESS);
 }
```

Running the above code prints the following:

```
Allocated an environment handle.
Set application version to ODBC 3.
Allocated Database handle.
Connecting to database.
Connected to database.
Disconnecting and freeing handles.
```

See Setting the Locale for ODBC Sessions for an example of using SQLDriverConnect to connect to the database.

Notes

- If you use the DataDirect® driver manager, you should always use the SQL_DRIVER_NOPROMPT value for the SQLDriverConnect function's DriverCompletion parameter (the final parameter in the function call) when connecting to HP Vertica. HP Vertica's ODBC driver on Linux and UNIX platforms does not contain a UI, and therefore cannot prompt users for a password.

- On Windows client platforms, the ODBC driver can prompt users for connection information. See Prompting Windows Users for Missing Connection Parameters for more information.

- If your database is not in compliance with the terms your license agreement (for example, it is larger than the data allowance in your license), HP Vertica sends a message about the non-compliance to your client application in the return value of the `SQLConnect()` function. Your application should always test this return value to see if it is SQL_SUCCESS_WITH_INFO. If it is, your application should extract and display the message to the user.

Enabling Native Connection Load Balancing in ODBC

Native connection load balancing helps spread the overhead caused by client connections on the hosts in the HP Vertica database. Both the server and the client must enable native connection load balancing in order for it to have an effect. If both have enabled it, then when the client initially connects to a host in the database, the host picks a host to handle the client connection from a list of the currently up hosts in the database, and informs the client which host it has chosen. If the initially-contacted host did not choose itself to handle the connection, the client disconnects, then opens a second connection to the host selected by the first host. The connection process to this second host proceeds as usual—if SSL is enabled, then SSL negotiations begin, otherwise the client begins the authentication process. See About Native Connection Load Balancing in the Administrator's Guide for details.

To enable native load balancing on your client, set the ConnectionLoadBalance connection parameter to true either in the DSN entry or in the connection string. The following example demonstrates connecting to the database several times with native connection load balancing enabled, and fetching the name of the node handling the connection from the V_MONITOR.CURRENT_SESSION system table.

```
// Demonstrate enabling native load connection balancing.
// Standard i/o library
#include <stdlib.h>
#include <iostream>
#include <assert.h>
// Only needed for Windows clients
// #include <windows.h>
// SQL include files that define data types and ODBC API
// functions
#include <sql.h>
#include <sqlext.h>
```

```
#include <sqltypes.h>

using namespace std;
int main()
{
    SQLRETURN ret;   // Stores return value from ODBC API calls
    SQLHENV hdlEnv;  // Handle for the SQL environment object
    // Allocate an a SQL environment object
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    assert(SQL_SUCCEEDED(ret));

    // Set the ODBC version we are going to use to
    // 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
            (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    assert(SQL_SUCCEEDED(ret));

    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    assert(SQL_SUCCEEDED(ret));

    // Connect four times. If load balancing is on, client should
    // connect to different nodes.
    for (int x=1; x <= 4; x++) {

        // Connect to the database using SQLDriverConnect. Set
        // ConnectionLoadBalance to 1 (true) to enable load
        // balancing.
        cout << endl << "Connection attempt #" << x << "... ";
        const char *connStr = "DSN=VMart;ConnectionLoadBalance=1;"
            "UID=ExampleUser;PWD=password123";


        ret = SQLDriverConnect(hdlDbc, NULL, (SQLCHAR*)connStr, SQL_NTS,
               NULL, 0, NULL, SQL_DRIVER_NOPROMPT );
        if(!SQL_SUCCEEDED(ret)) {
            cout << "failed. Exiting." << endl;
            exit(EXIT_FAILURE);
        } else {
            cout << "succeeded" << endl;
        }
        // We're connected. Query the v_monitor.current_session table to
        // find the name of the node we've connected to.

        // Set up a statement handle
        SQLHSTMT hdlStmt;
        SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
        assert(SQL_SUCCEEDED(ret));

        ret = SQLExecDirect( hdlStmt, (SQLCHAR*)"SELECT node_name FROM "
            "V_MONITOR.CURRENT_SESSION;", SQL_NTS );

        if(SQL_SUCCEEDED(ret)) {
            // Bind varible to column in result set.
            SQLTCHAR node_name[256];
            ret = SQLBindCol(hdlStmt, 1, SQL_C_TCHAR, (SQLPOINTER)node_name,
                sizeof(node_name), NULL);
            while(SQL_SUCCEEDED(ret = SQLFetchScroll(hdlStmt, SQL_FETCH_NEXT,1))) {
                // Print the bound variables, which now contain the values from the
                // fetched row.
                cout << "Connected to node " << node_name << endl;
            }
        }
        // Free statement handle
        SQLFreeHandle(SQL_HANDLE_STMT,hdlStmt);
        cout << "Disconnecting." << endl;
        ret = SQLDisconnect( hdlDbc );
        assert(SQL_SUCCEEDED(ret));
    }
    // When done, free all of the handles to close them
    // in an orderly fashion.
    cout << endl << "Freeing handles..." << endl;
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    cout << "Done!" << endl;
    exit(EXIT_SUCCESS);
}
```

Running the above example produces output similar to the following:

```
Connection attempt #1... succeeded
Connected to node v_vmart_node0001
Disconnecting.

Connection attempt #2... succeeded
Connected to node v_vmart_node0002
Disconnecting.

Connection attempt #3... succeeded
Connected to node v_vmart_node0003
Disconnecting.

Connection attempt #4... succeeded
Connected to node v_vmart_node0001
Disconnecting.

Freeing handles...
Done!
```

ODBC Connection Failover

If a client application attempts to connect to a host in the Vertica Analytics Platform cluster that is down, the connection attempt fails when using the default connection configuration. This failure usually returns an error to the user. The user must either wait until the host recovers and retry the connection or manually edit the connection settings to choose another host.

Due to Vertica Analytics Platform's distributed architecture, you usually do not care which database host handles a client application's connection. You can use the client driver's connection failover feature to prevent the user from getting connection errors when the host specified in the connection settings is unreachable. It gives you two ways to let the client driver automatically attempt to connect to a different host if the one specified in the connection parameters is unreachable:

- Configure your DNS server to return multiple IP addresses for a host name. When you use this host name in the connection settings, the client attempts to connect to the first IP address from the DNS lookup. If the host at that IP address is unreachable, the client tries to connect to the second IP, and so on until it either manages to connect to a host or it runs out of IP addresses.
- Supply a list of backup hosts for the client driver to try if the primary host you specify in the connection parameters is unreachable.

For both methods, the process of failover is transparent to the client application (other than specifying the list of backup hosts, if you choose to use the list method of failover). If the primary host is unreachable, the client driver automatically tries to connect to other hosts.

Failover only applies to the initial establishment of the client connection. If the connection breaks, the driver does not automatically try to reconnect to another host in the database.

Choosing a Failover Method

You usually choose to use one of the two failover methods. However, they do work together. If your DNS server returns multiple IP addresses and you supply a list of backup hosts, the client first tries all of the IPs returned by the DNS server, then the hosts in the backup list.

Note: If a host name in the backup host list resolves to multiple IP addresses, the client does not try all of them. It just tries the first IP address in the list.

The DNS method of failover centralizes the configuration client failover. As you add new nodes to your Vertica Analytics Platform cluster, you can choose to add them to the failover list by editing the DNS server settings. All client systems that use the DNS server to connect to Vertica Analytics Platform automatically use connection failover without having to change any settings. However, this method does require administrative access to the DNS server that all clients use to connect to the Vertica Analytics Platform cluster. This may not be possible in your organization.

Using the backup server list is easier than editing the DNS server settings. However, it decentralizes the failover feature. You may need to update the application settings on each client system if you make changes to your Vertica Analytics Platform cluster.

Using DNS Failover

To use DNS failover, you need to change your DNS server's settings to map a single host name to multiple IP addresses of hosts in your Vertica Analytics Platform cluster. You then have all client applications use this host name to connect to Vertica Analytics Platform.

You can choose to have your DNS server return as many IP addresses for the host name as you want. In smaller clusters, you may choose to have it return the IP addresses of all of the hosts in your cluster. However, for larger clusters, you should consider choosing a subset of the hosts to return. Otherwise there can be a long delay as the client driver tries unsuccessfully to connect to each host in a database that is down.

Using the Backup Host List

To enable backup list-based connection failover, your client application has to specify at least one IP address or host name of a host in the `BackupServerNode` parameter. The host name or IP can optionally be followed by a colon and a port number. If not supplied, the driver defaults to the standard HP Vertica port number (5433). To list multiple hosts, separate them by a comma.

The following example demonstrates setting the `BackupServerNode` connection parameter to specify additional hosts for the connection attempt. The connection string intentionally has a non-existent node, so that the initial connection fails. The client driver has to resort to trying the backup hosts to establish a connection to HP Vertica.

```
// Demonstrate using connection failover.
// Standard i/o library
#include <stdlib.h>
#include <iostream>
#include <assert.h>

// Only needed for Windows clients
// #include <windows.h>

// SQL include files that define data types and ODBC API
// functions
#include <sql.h>
#include <sqlext.h>
#include <sqltypes.h>

using namespace std;

int main()
{
    SQLRETURN ret;   // Stores return value from ODBC API calls
    SQLHENV hdlEnv;  // Handle for the SQL environment object
    // Allocate an a SQL environment object
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    assert(SQL_SUCCEEDED(ret));

    // Set the ODBC version we are going to use to
    // 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
            (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    assert(SQL_SUCCEEDED(ret));

    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    assert(SQL_SUCCEEDED(ret));

/* DSN for this connection specifies a bad node, and good backup nodes:
[VMartBadNode]
Description=VMart Vertica Database
Driver=/opt/vertica/lib64/libverticaodbc.so
Database=VMart
Servername=badnode.example.com
BackupServerNode=node02.example.com,node03.example.com
*/
```

```
    // Connect to the database using SQLConnect
    cout << "Connecting to database." << endl;
    const char *dsnName = "VMartBadNode"; // Name of the DSN
    const char* userID = "ExampleUser"; // Username
    const char* passwd = "password123"; // password
    ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if(!SQL_SUCCEEDED(ret))
{
        cout << "Could not connect to database." << endl;
        exit(EXIT_FAILURE);
    } else
{
        cout << "Connected to database." << endl;
    }
    // We're connected. Query the v_monitor.current_session table to
    // find the name of the node we've connected to.

    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
    assert(SQL_SUCCEEDED(ret));

    ret = SQLExecDirect( hdlStmt, (SQLCHAR*)"SELECT node_name FROM "
        "v_monitor.current_session;", SQL_NTS );

    if(SQL_SUCCEEDED(ret)) {
        // Bind varible to column in result set.
        SQLTCHAR node_name[256];
        ret = SQLBindCol(hdlStmt, 1, SQL_C_TCHAR, (SQLPOINTER)node_name,
            sizeof(node_name), NULL);
        while(SQL_SUCCEEDED(ret = SQLFetchScroll(hdlStmt, SQL_FETCH_NEXT,1)))
{
            // Print the bound variables, which now contain the values from the
            // fetched row.
            cout << "Connected to node " << node_name << endl;
        }
    }

    cout << "Disconnecting." << endl;
    ret = SQLDisconnect( hdlDbc );
    assert(SQL_SUCCEEDED(ret));

    // When done, free all of the handles to close them
    // in an orderly fashion.
    cout << endl << "Freeing handles..." << endl;
    SQLFreeHandle(SQL_HANDLE_STMT,hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    cout << "Done!" << endl;
    exit(EXIT_SUCCESS);
}
```

When run, the example's output on the system console is similar to the following:

```
Connecting to database.
Connected to database.
Connected to node v_vmart_node0002
Disconnecting.

Freeing handles...
Done!
```

Notice that the connection was made to the first node in the backup list (node 2).

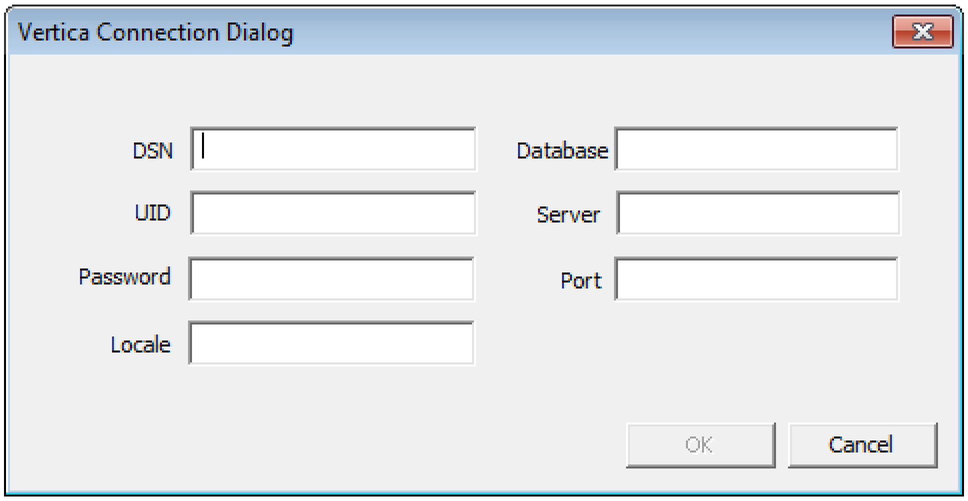
Note: When native connection load balancing is enabled, the additional servers specified in the BackupServerNode connection parameter are only used for the initial connection to an HP Vertica host. If host redirects the client to another host in the database cluster to handle its connection request, the second connection does not use the backup node list. This is rarely an issue, since native connection load balancing is aware of which nodes are currently up in the database. See Enabling Native Connection Load Balancing in ODBC

Prompting Windows Users for Missing Connection Parameters

The HP Vertica Windows ODBC driver can prompt the user for connection information if required information is missing. The driver displays the HP Vertica Connection Dialog if the client application calls `SQLDriverConnect` to connect to HP Vertica and either of the following is true:

- the DriverCompletion parameter is set to SQL_DRIVER_PROMPT.
- the DriverCompletion parameter is set to SQL_DRIVER_COMPLETE or SQL_DRIVER_COMPLETE_REQUIRED and the connection string or DSN being used to connect is missing the server, database, or port information.

If either of the above conditions are true, the driver displays an HP Vertica Connection Dialog to the user to prompt for connection information.



Any parameters supplied in the connection string or DSN filled in on the dialog.

Note: Your connection string at least needs to specify Vertica as the driver, otherwise Windows will not know to use the Vertica ODBC driver to try to open the connection.

The required fields on the connection dialog are Database, UID, Server, and Port. Once these are filled in, the form enables the **OK** button.

If the user clicks **Cancel** on the dialog, the `SQLDriverConnect` function call returns SQL_NO_DATA immediately, without attempting to connect to HP Vertica. If the user supplies incomplete or incorrect information for the connection, the connection function returns SQL_ERROR after the connection attempt fails.

Note: If the DriverCompletion parameter of the `SQLDriverConnect` function call is SQL_DRIVER_NOPROMPT, the ODBC driver immediately returns a SQL_ERROR indicating that it cannot connect because not enough information has been supplied and the driver is not allowed to prompt the user for the missing information.

Prompting Windows Users for Passwords

If the connection string or DSN supplied to the `SQLDriverConnect` function that client applications call to connect to HP Vertica lacks any of the required connection properties needed to connect, the HP Vertica's Windows ODBC driver opens a dialog box to prompt the user to enter the missing information (see Prompting Windows Users for Missing Connection Parameters). The user's password is not normally considered a required connection property, since HP Vertica user accounts may not have a password. If the password property is missing, the ODBC driver still tries to connect to HP Vertica without supplying a password.

You can use the PromptOnNoPassword DSN parameter to force ODBC driver to treat the password as a required connection property. This parameter is useful if you do not want to store passwords in DSN entries. Passwords saved in DSN entries are insecure, since they are stored as clear text in the Windows registry and therefore visible to other users on the same system.

There are two other factors which also decide whether the ODBC driver displays the the HP Vertica Connection Dialog. These are (in order of priority):

- The `SQLDriverConnect` function call's DriverCompletion parameter.
- Whether the DSN or connection string contain a password

The following table shows how the PromptOnNoPassword DSN parameter, the DriverCompletion parameter of the `SQLDriverConnect` function, and whether the DSN or connection string contains a password interact to control whether the HP Vertica Connection dialog appears.

PromptOnNoPassword Setting	DriverCompletion Value	DSN or Connection String Contains Password?	HP Vertica Connection Dialog Displays?	Notes
any value	SQL_DRIVER_ PROMPT	any case	Yes	This DriverCompletion value forces the dialog to always appear, even if all required connection properties are supplied.
any value	SQL_DRIVER_ NOPROMPT	any case	No	This DriverCompletion value always prevents the dialog from appearing.
any value	SQL_DRIVER_ COMPLETE	Yes	No	Connection dialog displays if another required connection property is missing.
true	SQL_DRIVER_ COMPLETE	No	Yes	
false (default)	SQL_DRIVER_ COMPLETE	No	No	Connection dialog displays if another required connection property is missing.

The following example code demonstrates using the PromptOnNoPassword DSN parameter along with a system DSN.

```
          wstring connectString = L"DSN=VerticaDSN;PromptOnNoPassword=1;";
retcode = SQLDriverConnect(
              hdbc,
              0,
              (SQLWCHAR*)connectString.c_str(),
              connectString.length(),
              OutConnStr,
              255,
              &OutConnStrLen,
              SQL_DRIVER_COMPLETE );
```

No Password Entry vs. Empty Passwords

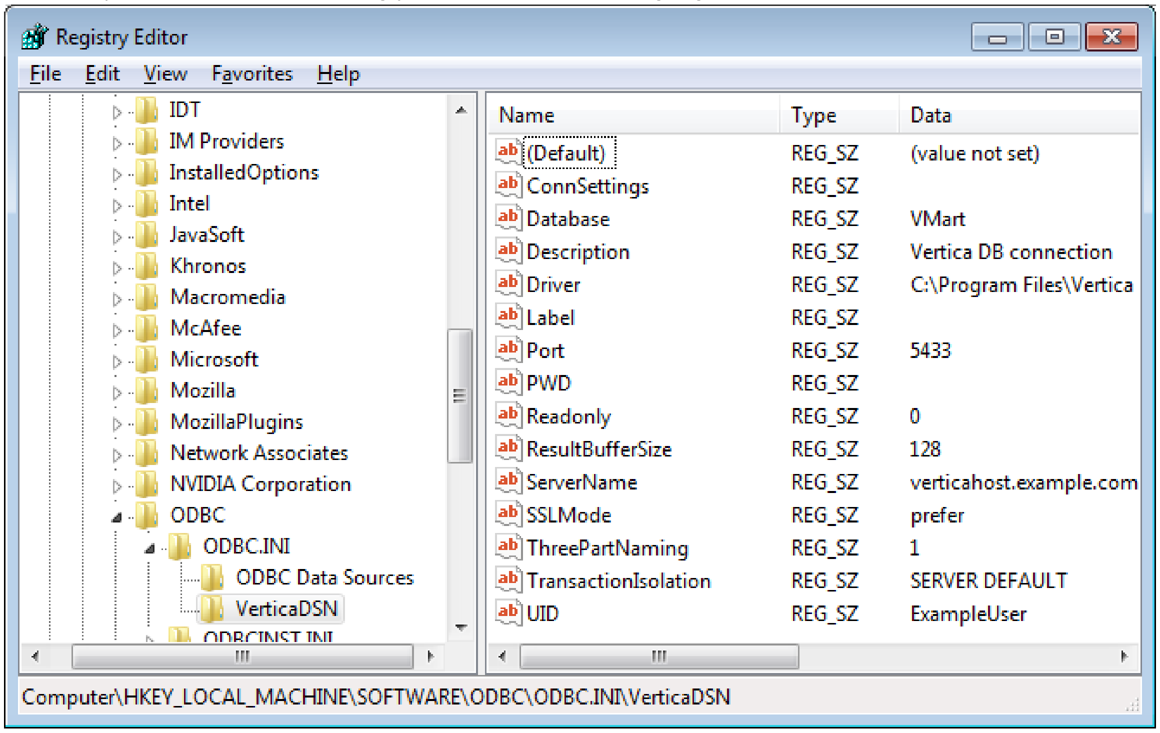
There is a difference between not having a password property in the connection string or DSN and having an empty password. The PromptOnNoPassword DSN parameter only has an effect if the connection string or DSN does not have a PWD property (which holds the user's password). If it does, even if it is empty, PromptOnNoPassword will not prompt the Windows ODBC driver to display the HP Vertica Connection Dialog.

This difference can cause confusion if you are using a DSN to provide the properties for your connection. Once you enter a password for a DSN connection in the Windows ODBC Manager and save it, Windows adds a PWD property to the DSN definition in the registry. If you later delete the password, the PWD property remains in the DSN definition—value is just set to an empty string. The PWD property is created even if you just use the Test button on the ODBC Manager dialog to test the DSN and later clear it before saving the DSN.

Once the password has been set, the only way to remove the PWD property from the DSN definition is to delete it using the Windows Registry Editor:

1. On the Windows Start menu, click Run.
2. In the Run dialog, type regedit, then click OK.
3. In the Registry Editor window, click Edit > Find (or press Ctrl+F).
4. In the Find window, enter the name of the DSN whose PWD property you want to delete and click OK.

5. If find operation did not locate a folder under the ODBC.INI folder, click Edit > Find Next (or press F3) until the folder matching your DSN's name is highlighted.



Name	Type	Data
(Default)	REG_SZ	(value not set)
ConnSettings	REG_SZ	
Database	REG_SZ	VMart
Description	REG_SZ	Vertica DB connection
Driver	REG_SZ	C:\Program Files\Vertica
Label	REG_SZ	
Port	REG_SZ	5433
PWD	REG_SZ	
Readonly	REG_SZ	0
ResultBufferSize	REG_SZ	128
ServerName	REG_SZ	verticahost.example.com
SSLMode	REG_SZ	prefer
ThreePartNaming	REG_SZ	1
TransactionIsolation	REG_SZ	SERVER DEFAULT
UID	REG_SZ	ExampleUser

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\VerticaDSN

6. Select the PWD entry and press Delete.

7. Click Yes to confirm deleting the value.

The DSN now does not have a PWD property and can trigger the connection dialog to appear when used along with PromptOnNoPassword=true and DriverConnect=SQL_DRIVER_COMPLETE.

Setting the Locale for ODBC Sessions

HP Vertica provides three ways to set the locale for an ODBC session:

- Specify the locale for all connections made using the DSN:
  - On Linux and other UNIX-like platforms: Creating an ODBC DSN for Linux, Solaris, AIX, and HP-UX
  - On Windows platforms, set the locale in the ODBC DSN configuration editor's Locale field on the Server Settings tab. See Creating an ODBC DSN for Windows Clients for detailed information.
- Set the Locale connection parameter in the connection string in `SQLDriverConnect()` function.

  For example:

```
SQLDriverConnect(conn, NULL, (SQLCHAR*)"DSN=Vertica;Locale=en_GB", SQL_NTS, szConnOut,
sizeof(szConnOut), &iAvailable, SQL_DRIVER_NOPROMPT)
```

- Use the `SQLSetConnectAttr()` method with the `SQL_ATTR_VERTICA_LOCALE` constant and specify the ICU string as the attribute value. See the example below.

Notes

- Having the client system use a non-Unicode locale (such as setting `LANG=C` on Linux platforms) and using a Unicode locale for the connection to HP Vertica can result in errors such as "(10170) String data right truncation on data from data source." If data received from HP Vertica isn't in UTF-8 format. The driver allocates string memory based on the system's locale setting, and non-UTF-8 data can trigger an overrun. You can avoid these errors by always using a Unicode locale on the client system.

  If you specify a locale either in the connection string or in the DSN, the call to the connection function returns SQL_SUCCESS_WITH_INFO on a successful connection, with messages about the state of the locale.

- ODBC applications can be in either ANSI or Unicode mode:

  - If Unicode, the encoding used by ODBC is UCS-2.
  - If ANSI, the data must be in single-byte ASCII, which is compatible with UTF-8 on the database server.

  The ODBC driver converts UCS-2 to UTF-8 when passing to the HP Vertica server and converts data sent by the HP Vertica server from UTF-8 to UCS-2.

- If the end-user application is not already in UCS-2, the application is responsible for converting the input data to UCS-2, or unexpected results could occur. For example:

  - On non-UCS-2 data passed to ODBC APIs, when it is interpreted as UCS-2, it could result in an invalid UCS-2 symbol being passed to the APIs, resulting in errors.
  - Or the symbol provided in the alternate encoding could be a valid UCS-2 symbol; in this case, incorrect data is inserted into the database.

  ODBC applications should set the correct server session locale using `SQLSetConnectAttr` (if different from database-wide setting) in order to set the proper collation and string functions behavior on server.

The following example code demonstrates setting the locale using both the connection string and through the `SQLSetConnectAttr()` function.

```
// Standard i/o library
#include <stdio.h>
#include <stdlib.h>
```

```
// Only needed for Windows clients
// #include <windows.h>
// SQL include files that define data types and ODBC API
// functions
#include <sql.h>
#include <sqlext.h>
#include <sqltypes.h>
// Vertica-specific definitions. This include file is located as
// /opt/vertica/include on database hosts.
#include <verticaodbc.h>
int main()
{
    SQLRETURN ret;   // Stores return value from ODBC API calls
    SQLHENV hdlEnv;  // Handle for the SQL environment object
    // Allocate an a SQL environment object
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }
    // Set the ODBC version we are going to use to 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not set application version to ODBC 3.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application version to ODBC 3.\n");
    }
    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate database handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated Database handle.\n");
    }
    // Connect to the database using SQLDriverConnect
    printf("Connecting to database.\n");
    // Set the locale to English in Great Britain.
    const char *connStr = "DSN=ExampleDB;locale=en_GB;"
        "UID=dbadmin;PWD=password123";
    ret = SQLDriverConnect(hdlDbc, NULL, (SQLCHAR*)connStr, SQL_NTS,
               NULL, 0, NULL, SQL_DRIVER_NOPROMPT );
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not connect to database.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Connected to database.\n");
    }
    // Get the Locale
    char locale[256];
    SQLGetConnectAttr(hdlDbc, SQL_ATTR_VERTICA_LOCALE, locale, sizeof(locale),
        0);
    printf("Locale is set to: %s\n", locale);
    // Set the locale to a new value
    const char* newLocale = "en_GB";
    SQLSetConnectAttr(hdlDbc, SQL_ATTR_VERTICA_LOCALE, (SQLCHAR*)newLocale,
        SQL_NTS);

    // Get the Locale again
    SQLGetConnectAttr(hdlDbc, SQL_ATTR_VERTICA_LOCALE, locale, sizeof(locale),
        0);
    printf("Locale is now set to: %s\n", locale);
    // When done, free all of the handles to close them
    // in an orderly fashion.
    printf("Disconnecting and freeing handles.\n");
    ret = SQLDisconnect( hdlDbc );
    if(!SQL_SUCCEEDED(ret)) {
        printf("Error disconnecting from database. Transaction still open?\n");
        exit(EXIT_FAILURE);
    }
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

AUTOCOMMIT and ODBC Transactions

The AUTOCOMMIT connection attribute controls whether INSERT, ALTER, COPY and other data-manipulation statements are automatically committed after they complete. By default, AUTOCOMMIT is enabled—all statements are committed after they execute. This is often not the best setting to use, since it is less efficient. Also, you often want to control whether a set of statements are committed as a whole, rather than have each individual statement committed. For example, you may only want to commit a series of inserts if all of the inserts succeed. With AUTOCOMMIT disabled, you can roll back the transaction if one of the statements fail.

If AUTOCOMMIT is on, the results of statements are committed immediately after they are executed. You cannot roll back a statement executed in AUTOCOMMIT mode.

For example, when AUTOCOMMIT is on, the following single INSERT statement is automatically committed:

```
ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"INSERT INTO customers VALUES(500,"
    "'Smith, Sam', '123-456-789');", SQL_NTS);
```

If AUTOCOMMIT is off, you need to manually commit the transaction after executing a statement. For example:

```
ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"INSERT INTO customers VALUES(500,"
    "'Smith, Sam', '123-456-789');", SQL_NTS);
// Other inserts and data manipulations
// Commit the statements(s)
ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
```

The inserted row is only committed when you call `SQLEndTran()`. You can roll back the INSERT and other statements at any point before committing the transaction.

Note: Prepared statements cache the AUTOCOMMIT setting when you create them using `SQLPrepare()`. Later changing the connection's AUTOCOMMIT setting has no effect on the AUTOCOMMIT settings of previously created prepared statements. See Using Prepared Statements for details.

The following example demonstrates turning off AUTOCOMMIT, executing an insert, then manually committing the transaction.

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
int main()
{
    // Set up the ODBC environment
    SQLRETURN ret;
    SQLHENV hdlEnv;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not set application version to ODBC3.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application to ODBC 3.\n");
    }
    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate database handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated Database handle.\n");
    }
    // Connect to the database
    printf("Connecting to database.\n");
    const char *dsnName = "ExampleDB";
    const char* userID = "dbadmin";
    const char* passwd = "password123";
    ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not connect to database.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Connected to database.\n");
    }
    // Get the AUTOCOMMIT state
    SQLINTEGER  autoCommitState;
    SQLGetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, &autoCommitState, 0, NULL);
    printf("Autocommit is set to: %d\n", autoCommitState);


    // Disable AUTOCOMMIT
    printf("Disabling autocommit.\n");
    ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF,
        SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not disable autocommit.\n");
        exit(EXIT_FAILURE);
    }

    // Get the AUTOCOMMIT state again
    SQLGetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, &autoCommitState, 0, NULL);
    printf("Autocommit is set to: %d\n", autoCommitState);

    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);


    // Create a table to hold the data
    SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
        SQL_NTS);
    SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
        "(CustID int, CustName varchar(100), Phone_Number char(15));",
        SQL_NTS);


    // Insert a single row.
    ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"INSERT INTO customers VALUES(500,"
        "'Smith, Sam', '123-456-789');", SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not perform single insert.\n");
    } else {
        printf("Performed single insert.\n");
    }


    // Need to commit the transaction before closing, since autocommit is
    // disabled. Otherwise SQLDisconnect returns an error.
    printf("Committing transaction.\n");
    ret =  SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Error committing transaction.\n");
        exit(EXIT_FAILURE);
    }

    // Clean up
    printf("Free handles.\n");
    ret = SQLDisconnect(hdlDbc);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Error disconnecting from database. Transaction still open?\n");
        exit(EXIT_FAILURE);
    }
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

Running the above code results in the following output:

```
Allocated an environment handle.
Set application to ODBC 3.
Allocated Database handle.
Connecting to database.
Connected to database.
Autocommit is set to: 1
Disabling autocommit.
Autocommit is set to: 0
Performed single insert.
Committing transaction.
Free handles.
```

Note: You can also disable AUTOCOMMIT in the ODBC connection string. See Setting DSN Parameters for more information.

Retrieving Data Through ODBC

To retrieve data through ODBC, you execute a query that returns a result set (SELECT, for example), then retrieve the results using one of two methods:

- Use the `SQLFetch()` function to retrieve a row of the result set, then access column values in the row by calling `SQLGetData()`.
- Use the `SQLBindColumn()` function to bind a variable or array to a column in the result set, then call `SQLExtendedFetch()` or `SQLFetchScroll()` to read a row of the result set and insert its values into the variable or array.

In both methods you loop through the result set until you either reach the end (signaled by the SQL_NO_DATA return status) or encounter an error.

Note: HP Vertica supports one cursor per connection. Attempting to use more than one cursor per connection will result in an error. For example, you receive an error if you execute a statement while another statement has a result set open.

The following code example demonstrates retrieving data from HP Vertica by:

1. Connecting to the database.
2. Executing a SELECT statement that returns the IDs and names of all tables.
3. Binds two variables to the two columns in the result set.
4. Loops through the result set, printing the ids and name values.

```
// Demonstrate running a query and getting results by querying the tables
// system table for a list of all tables in the current schema.
// Some standard headers
#include <stdlib.h>
#include <sstream>
#include <iostream>
#include <assert.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
// Use std namespace to make output easier
using namespace std;
// Helper function to print SQL error messages.
template <typename HandleT>
void reportError(int handleTypeEnum, HandleT hdl)
{
    // Get the status records.
    SQLSMALLINT   i, MsgLen;
    SQLRETURN ret2;
    SQLCHAR       SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH];
    SQLINTEGER    NativeError;
    i = 1;
    cout << endl;
    while ((ret2 = SQLGetDiagRec(handleTypeEnum, hdl, i, SqlState, &NativeError,
                                Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA) {
        cout << "error record #" << i++ << endl;
        cout << "sqlstate: " << SqlState << endl;
        cout << "detailed msg: " << Msg << endl;
        cout << "native error code: " << NativeError << endl;
    }
}
int main()
{
    // Set up the ODBC environment
    SQLRETURN ret;
    SQLHENV hdlEnv;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    assert(SQL_SUCCEEDED(ret));
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    assert(SQL_SUCCEEDED(ret));
    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    assert(SQL_SUCCEEDED(ret));
    // Connect to the database
    cout << "Connecting to database." << endl;
    const char* dsnName = "ExampleDB";
    const char* userID = "dbadmin";
    const char* passwd = "password123";
    ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        cout << "Could not connect to database" << endl;
        reportError<SQLHDBC>(SQL_HANDLE_DBC, hdlDbc);
        exit(EXIT_FAILURE);
    } else {
        cout << "Connected to database." << endl;
    }


    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
    assert(SQL_SUCCEEDED(ret));

    // Execute a query to get the names and IDs of all tables in the schema
    // search p[ath (usually public).
    ret = SQLExecDirect( hdlStmt, (SQLCHAR*)"SELECT table_id, table_name "
        "FROM tables ORDER BY table_name", SQL_NTS );

    if(!SQL_SUCCEEDED(ret)) {
        // Report error an go no further if statement failed.
        cout << "Error executing statement." << endl;
        reportError<SQLHDBC>(SQL_HANDLE_STMT, hdlStmt);
        exit(EXIT_FAILURE);
    } else {


        // Query succeeded, so bind two variables to the two colums in the
        // result set,
        cout << "Fetching results..." << endl;
        SQLBIGINT table_id;       // Holds the ID of the table.
        SQLTCHAR table_name[256]; // buffer to hold name of table
        ret = SQLBindCol(hdlStmt, 1, SQL_C_SBIGINT, (SQLPOINTER)&table_id,
            sizeof(table_id), NULL);
        ret = SQLBindCol(hdlStmt, 2, SQL_C_TCHAR, (SQLPOINTER)table_name,
            sizeof(table_name), NULL);

        // Loop through the results,
        while( SQL_SUCCEEDED(ret = SQLFetchScroll(hdlStmt, SQL_FETCH_NEXT,1))) {
            // Print the bound variables, which now contain the values from the
            // fetched row.
            cout << table_id << " | " << table_name << endl;
        }


        // See if loop exited for reasons other than running out of data
        if (ret != SQL_NO_DATA) {
            // Exited for a reason other than no more data... report the error.
            reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
        }
    }
    // Clean up by shutting down the connection
    cout << "Free handles." << endl;
    ret = SQLDisconnect( hdlDbc );
    if(!SQL_SUCCEEDED(ret)) {
        cout << "Error disconnecting. Transaction still open?" << endl;
        exit(EXIT_FAILURE);
    }
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

Running the example code in the vmart database produces output similar to this:

```
Connecting to database.
Connected to database.
Fetching results...
45035996273970908 | call_center_dimension
45035996273970836 | customer_dimension
45035996273972958 | customers
45035996273970848 | date_dimension
45035996273970856 | employee_dimension
45035996273970868 | inventory_fact
45035996273970904 | online_page_dimension
45035996273970912 | online_sales_fact
45035996273970840 | product_dimension
45035996273970844 | promotion_dimension
45035996273970860 | shipping_dimension
45035996273970876 | store_dimension
45035996273970894 | store_orders_fact
45035996273970880 | store_sales_fact
45035996273972806 | t
45035996273970852 | vendor_dimension
45035996273970864 | warehouse_dimension
Free handles.
```

Loading Data Through ODBC

A primary task for many client applications is loading data into the HP Vertica database. There are several different ways to insert data using ODBC, which are covered by the topics in this section.

Using a Single Row Insert

The easiest way to load data into HP Vertica is to run an INSERT SQL statement using the SQLExecuteDirect function. However this method is limited to inserting a single row of data.

```
ret = SQLExecDirect(hstmt, (SQLTCHAR*)"INSERT into Customers values"
      "(1,'abcda','efgh','1')", SQL_NTS);
```

Using Prepared Statements

HP Vertica supports using server-side prepared statements with both ODBC and JDBC. Prepared statements let you define a statement once, and then run it many times with different parameters. The statement you want to execute contains placeholders instead of parameters. When you execute the statement, you supply values for each placeholder.

Placeholders are represented by question marks (?) as in the following example query:

```
SELECT * FROM public.inventory_fact WHERE product_key = ?
```

Server-side prepared statements are useful for:

- Optimizing queries. HP Vertica only needs to parse the statement once.
- Preventing SQL injection attacks. A SQL injection attack occurs when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly run. Since a prepared statement is parsed separately from the input data, there is no chance the data can be accidentally executed by the database.
- Binding direct variables to return columns. By pointing to data structures, the code doesn't have to perform extra transformations.

The following example demonstrates a using a prepared statement for a single insert.

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
// Some constants for the size of the data to be inserted.
#define CUST_NAME_LEN 50
#define PHONE_NUM_LEN 15
#define NUM_ENTRIES 4
int main()
{
    // Set up the ODBC environment
    SQLRETURN ret;
    SQLHENV hdlEnv;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not set application version to ODBC3.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application to ODBC 3.\n");
    }
    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    // Connect to the database
    printf("Connecting to database.\n");
    const char *dsnName = "ExampleDB";
    const char* userID = "dbadmin";
    const char* passwd = "password123";
    ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not connect to database.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Connected to database.\n");
    }

    // Disable AUTOCOMMIT
    printf("Disabling autocommit.\n");
    ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF,
        SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not disable autocommit.\n");
        exit(EXIT_FAILURE);
    }


    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
    SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
        SQL_NTS);
    SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
        "(CustID int, CustName varchar(100), Phone_Number char(15));",
        SQL_NTS);

    // Set up a bunch of variables to be bound to the statement
    // parameters.

    // Create the prepared statement. This will insert data into the
    // table we created above.
    printf("Creating prepared statement\n");
    ret = SQLPrepare (hdlStmt, (SQLTCHAR*)"INSERT INTO customers (CustID, "
        "CustName,  Phone_Number) VALUES(?,?,?)", SQL_NTS) ;
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not create prepared statement\n");
        SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
        SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
        SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
        exit(EXIT_FAILURE);
    } else {
        printf("Created prepared statement.\n");
    }
    SQLINTEGER custID = 1234;
    SQLCHAR custName[100] = "Fein, Fredrick";
    SQLVARCHAR phoneNum[15] = "555-123-6789";
    SQLLEN strFieldLen = SQL_NTS;
    SQLLEN custIDLen = 0;
    // Bind the data arrays to the parameters in the prepared SQL
    // statement
    ret = SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
        0, 0, &custID, 0 , &custIDLen);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not bind custID array\n");
        SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
        SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
        SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
        exit(EXIT_FAILURE);
    } else {
        printf("Bound custID to prepared statement\n");
    }
    // Bind CustNames
    SQLBindParameter(hdlStmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
        50, 0, (SQLPOINTER)custName,  0, &strFieldLen);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not bind custNames\n");
        SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
        SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
        SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
        exit(EXIT_FAILURE);
    } else {
        printf("Bound custName to prepared statement\n");
    }
    // Bind phoneNums
    SQLBindParameter(hdlStmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
        15, 0, (SQLPOINTER)phoneNum, 0, &strFieldLen);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not bind phoneNums\n");
        SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
        SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
        SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
        exit(EXIT_FAILURE);
    } else {
        printf("Bound phoneNum to prepared statement\n");
    }
    // Execute the prepared statement.
    printf("Running prepared statement...");
    ret = SQLExecute(hdlStmt);
    if(!SQL_SUCCEEDED(ret)) {
        printf("not successful!\n");
    }  else {
        printf("successful.\n");
    }

    // Done with batches, commit the transaction
    printf("Committing transaction\n");
    ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not commit transaction\n");
    }  else {
        printf("Committed transaction\n");
    }

    // Clean up
    printf("Free handles.\n");
    ret = SQLDisconnect( hdlDbc );
    if(!SQL_SUCCEEDED(ret)) {
        printf("Error disconnecting. Transaction still open?\n");
        exit(EXIT_FAILURE);
    }
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

Using Batch Inserts

You use batch inserts to insert chunks of data into the database. By breaking the data into batches, you can monitor the progress of the load by receiving information about any rejected rows after each batch is loaded. To perform a batch load through ODBC, you typically use a prepared statement with the parameters bound to arrays that contain the data to be loaded. For each batch, you load a new set of data into the arrays then execute the prepared statement.

When you perform a batch load, HP Vertica uses a COPY statement to load the data. Each additional batch you load uses the same COPY statement. The statement remains open until you end the transaction, close the cursor for the statement, or execute a non-INSERT statement.

Using a single COPY statement for multiple batches improves batch loading efficiency by:

- reducing the overhead of inserting individual batches
- combining individual batches into larger ROS containers

Note: If the database connection has AUTOCOMMIT enabled, then the transaction is automatically committed after each batch insert statement which closes the COPY statement. Leaving AUTOCOMMIT enabled makes your batch load much less efficient, and can cause added overhead in your database as all of the smaller loads are consolidated.

Even though HP Vertica uses a single COPY statement to insert multiple batches within a transaction, you can locate which (if any) rows were rejected due to invalid row formats or data type issues after each batch is loaded. See Tracking Load Status (ODBC) for details.

Note: While you can find rejected rows during the batch load transaction, other types of errors (such as running out of disk space or a node shutdown that makes the database unsafe) are only reported when the COPY statement ends.

Since the batch loads share a COPY statement, errors in one batch can cause earlier batches in the same transaction to be rolled back.

Batch Insert Steps

The steps your application needs to take in order to perform an ODBC Batch Insert are:

1. Connect to the database.
2. Disable autocommit for the connection.
3. Create a prepared statement that inserts the data you want to load.
4. Bind the parameters of the prepared statement to arrays that will contain the data you want to load.
5. Populate the arrays with the data for your batches.
6. Execute the prepared statement.
7. Optionally, check the results of the batch load to find rejected rows.
8. Repeat the previous three steps until all of the data you want to load is loaded.
9. Commit the transaction.
10. Optionally, check the results of the entire batch transaction.

The following example code demonstrates a simplified version of the above steps.

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
int main()
{
    // Number of data rows to insert
    const int NUM_ENTRIES = 4;

    // Set up the ODBC environment
    SQLRETURN ret;
    SQLHENV hdlEnv;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not set application version to ODBC3.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application to ODBC 3.\n");
    }
    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate database handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated Database handle.\n");
    }
    // Connect to the database
    printf("Connecting to database.\n");
    const char *dsnName = "ExampleDB";
    const char* userID = "dbadmin";
    const char* passwd = "password123";
    ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not connect to database.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Connected to database.\n");
    }


    // Disable AUTOCOMMIT
    printf("Disabling autocommit.\n");
    ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF,
                            SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not disable autocommit.\n");
        exit(EXIT_FAILURE);
    }

    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);

    // Create a table to hold the data
    SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
        SQL_NTS);
    SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
        "(CustID int, CustName varchar(100), Phone_Number char(15));",
        SQL_NTS);

    // Create the prepared statement. This will insert data into the
    // table we created above.
    printf("Creating prepared statement\n");
```

```
ret = SQLPrepare (hdlStmt, (SQLTCHAR*)"INSERT INTO customers (CustID, "
    "CustName,  Phone_Number) VALUES(?,?,?)", SQL_NTS) ;
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not create prepared statement\n");
    exit(EXIT_FAILURE);
} else {
    printf("Created prepared statement.\n");
}
// This is the data to be inserted into the database.
SQLCHAR custNames[][50] = { "Allen, Anna", "Brown, Bill", "Chu, Cindy",
    "Dodd, Don" };
SQLINTEGER custIDs[] = { 100, 101, 102, 103};
SQLCHAR phoneNums[][15] = {"1-617-555-1234", "1-781-555-1212",
    "1-508-555-4321", "1-617-555-4444"};
// Bind the data arrays to the parameters in the prepared SQL
// statement. First is the custID.
ret = SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
    0, 0, (SQLPOINTER)custIDs, sizeof(SQLINTEGER) , NULL);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind custID array\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound CustIDs array to prepared statement\n");
}
// Bind CustNames
ret = SQLBindParameter(hdlStmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
    50, 0, (SQLPOINTER)custNames, 50, NULL);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind custNames\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound CustNames array to prepared statement\n");
}
// Bind phoneNums
ret = SQLBindParameter(hdlStmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
    15, 0, (SQLPOINTER)phoneNums, 15, NULL);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind phoneNums\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound phoneNums array to prepared statement\n");
}
// Tell the ODBC driver how many rows we have in the
// array.
ret = SQLSetStmtAttr( hdlStmt, SQL_ATTR_PARAMSET_SIZE,
    (SQLPOINTER)NUM_ENTRIES, 0 );
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not bind set parameter size\n");
    exit(EXIT_FAILURE);
} else {
    printf("Bound phoneNums array to prepared statement\n");
}

// Add multiple batches to the database. This just adds the same
// batch of data four times for simplicity's sake. Each call adds
// the 4 rows into the database.
for (int batchLoop=1; batchLoop<=5; batchLoop++) {
    // Execute the prepared statement, loading all of the data
```

```
        // in the arrays.
        printf("Adding Batch #%d...", batchLoop);
        ret = SQLExecute(hdlStmt);
        if(!SQL_SUCCEEDED(ret)) {
           printf("not successful!\n");
        }  else {
            printf("successful.\n");
        }
    }
    // Done with batches, commit the transaction
    printf("Committing transaction\n");
    ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not commit transaction\n");
    }  else {
        printf("Committed transaction\n");
    }

    // Clean up
    printf("Free handles.\n");
    ret = SQLDisconnect( hdlDbc );
    if(!SQL_SUCCEEDED(ret)) {
        printf("Error disconnecting. Transaction still open?\n");
        exit(EXIT_FAILURE);
    }
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

The result of running the above code is shown below.

```
Allocated an environment handle.
Set application to ODBC 3.
Allocated Database handle.
Connecting to database.
Connected to database.
Creating prepared statement
Created prepared statement.
Bound CustIDs array to prepared statement
Bound CustNames array to prepared statement
Bound phoneNums array to prepared statement
Adding Batch #1...successful.
Adding Batch #2...successful.
Adding Batch #3...successful.
Adding Batch #4...successful.
Adding Batch #5...successful.
Committing transaction
Committed transaction
Free handles.
```

The resulting table looks like this:

```
=> SELECT * FROM customers;
```

```
 CustID |  CustName   |  Phone_Number
--------+-------------+-----------------
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
    100 | Allen, Anna | 1-617-555-1234
    101 | Brown, Bill | 1-781-555-1212
    102 | Chu, Cindy  | 1-508-555-4321
    103 | Dodd, Don   | 1-617-555-4444
(20 rows)
```

Note: An input parameter bound with the SQL_C_NUMERIC data type uses the default numeric precision (37) and the default scale (0) instead of the precision and scale set by the SQL_NUMERIC_STRUCT input value. This behavior adheres to the ODBC standard. If you do not want to use the default precision and scale, use `SQLSetDescField()` or `SQLSetDescRec()` to change them in the statement's attributes.

Tracking Load Status (ODBC)

After loading a batch of data, your client application can get the number of rows that were processed and find out whether each row was accepted or rejected.

Finding the Number of Accepted Rows

To get the number of rows processed by a batch, you add an attribute named SQL_ATTR_PARAMS_PROCESSED_PTR to the statement object that points to a variable to receive the number rows:

```
    SQLULEN rowsProcessed;
    SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &rowsProcessed, 0);
```

When your application calls `SQLExecute()` to insert the batch, the HP Vertica ODBC driver saves the number of rows that it processed (which is not necessarily the number of rows that were successfully inserted) in the variable you specified in the SQL_ATTR_PARAMS_PROCESSED_PTR statement attribute.

Finding the Accepted and Rejected Rows

Your application can also set a statement attribute named SQL_ATTR_PARAM_STATUS_PTR that points to an array where the ODBC driver can store the result of inserting each row:

```
    SQLUSMALLINT   rowResults[ NUM_ENTRIES ];
    SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAM_STATUS_PTR, rowResults, 0);
```

This array must be at least as large as the number of rows being inserted in each batch.

When your application calls `SQLExecute` to insert a batch, the ODBC driver populates the array with values indicating whether each row was successfully inserted (SQL_PARAM_SUCCESS or SQL_PARAM_SUCCESS_WITH_INFO) or encountered an error (SQL_PARAM_ERROR).

The following example expands on the example shown in Using Batch Inserts to include reporting the number of rows processed and the status of each row inserted.

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
// Helper function to print SQL error messages.
template <typename HandleT>
void reportError(int handleTypeEnum, HandleT hdl)
{
    // Get the status records.
    SQLSMALLINT   i, MsgLen;
    SQLRETURN ret2;
    SQLCHAR       SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH];
    SQLINTEGER    NativeError;
    i = 1;
    printf("\n");
    while ((ret2 = SQLGetDiagRec(handleTypeEnum, hdl, i, SqlState, &NativeError,
        Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA) {
            printf("error record %d\n", i);
            printf("sqlstate: %s\n", SqlState);
            printf("detailed msg: %s\n", Msg);
            printf("native error code: %d\n\n", NativeError);
            i++;
    }
}
int main()
{
    // Number of data rows to insert
    const int NUM_ENTRIES = 4;


    SQLRETURN ret;
    SQLHENV hdlEnv;
```

```
ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate a handle.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Allocated an environment handle.\n");
}
ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
    (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not set application version to ODBC3.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Set application to ODBC 3.\n");
}
SQLHDBC hdlDbc;
ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not allocate database handle.\n");
    exit(EXIT_FAILURE);
} else {
    printf("Allocated Database handle.\n");
}
// Connect to the database
printf("Connecting to database.\n");
const char *dsnName = "ExampleDB";
const char* userID = "dbadmin";
const char* passwd = "password123";
ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
    SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
    (SQLCHAR*)passwd, SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    printf("Could not connect to database.\n");
    reportError<SQLHDBC>(SQL_HANDLE_DBC, hdlDbc);
    exit(EXIT_FAILURE);
} else {
    printf("Connected to database.\n");
}
// Set up a statement handle
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
    SQL_NTS);
// Create a table into which we can store data
printf("Creating table.\n");
ret = SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
    "(CustID int, CustName varchar(50), Phone_Number char(15));",
    SQL_NTS);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Created table.\n");
}
// Create the prepared statement. This will insert data into the
// table we created above.
printf("Creating prepared statement\n");
ret = SQLPrepare (hdlStmt, (SQLTCHAR*)"INSERT INTO customers (CustID, "
    "CustName,  Phone_Number) VALUES(?,?,?)", SQL_NTS) ;
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Created prepared statement.\n");
}
// This is the data to be inserted into the database.
char custNames[][50] = { "Allen, Anna", "Brown, Bill", "Chu, Cindy",
    "Dodd, Don" };
SQLINTEGER custIDs[] = { 100, 101, 102, 103};
char phoneNums[][15] = {"1-617-555-1234", "1-781-555-1212",
    "1-508-555-4321", "1-617-555-4444"};
// Bind the data arrays to the parameters in the prepared SQL
// statement
ret = SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
    0, 0, (SQLPOINTER)custIDs, sizeof(SQLINTEGER) , NULL);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Bound CustIDs array to prepared statement\n");
}
// Bind CustNames
SQLBindParameter(hdlStmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
    50, 0, (SQLPOINTER)custNames, 50, NULL);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Bound CustNames array to prepared statement\n");
}
// Bind phoneNums
SQLBindParameter(hdlStmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
    15, 0, (SQLPOINTER)phoneNums, 15, NULL);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
    exit(EXIT_FAILURE);
} else {
    printf("Bound phoneNums array to prepared statement\n");
}
// Set up a variable to recieve number of parameters processed.
SQLULEN rowsProcessed;
// Set a statement attribute to point to the variable
SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &rowsProcessed, 0);
// Set up an array to hold the result of each row insert
SQLUSMALLINT   rowResults[ NUM_ENTRIES ];
// Set a statement attribute to point to the array
SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAM_STATUS_PTR, rowResults, 0);
// Tell the ODBC driver how many rows we have in the
// array.
SQLSetStmtAttr(hdlStmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)NUM_ENTRIES, 0);
// Add multiple batches to the database. This just adds the same
// batch of data over and over again for simplicity's sake.
for (int batchLoop=1; batchLoop<=5; batchLoop++) {
    // Execute the prepared statement, loading all of the data
    // in the arrays.
    printf("Adding Batch #%d...", batchLoop);
    ret = SQLExecute(hdlStmt);
    if(!SQL_SUCCEEDED(ret)) {
        reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
        exit(EXIT_FAILURE);
    }
    // Number of rows processed is in rowsProcessed
    printf("Params processed: %d\n", rowsProcessed);
    printf("Results of inserting each row:\n");
    int i;
    for (i = 0; i<NUM_ENTRIES; i++) {
        SQLUSMALLINT result = rowResults[i];
        switch(rowResults[i]) {
            case SQL_PARAM_SUCCESS:
            case SQL_PARAM_SUCCESS_WITH_INFO:
                printf("  Row %d inserted successsfully\n", i+1);
                break;
            case SQL_PARAM_ERROR:
                printf("  Row %d was not inserted due to an error.", i+1);
                break;
            default:
                printf("  Row %d had some issue with it: %d\n", i+1, result);
        }
    }
}
// Done with batches, commit the transaction
printf("Commit Transaction\n");
ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
if(!SQL_SUCCEEDED(ret)) {
    reportError<SQLHDBC>( SQL_HANDLE_STMT, hdlStmt );
}


// Clean up
printf("Free handles.\n");
ret = SQLDisconnect( hdlDbc );
if(!SQL_SUCCEEDED(ret)) {
    printf("Error disconnecting. Transaction still open?\n");
    exit(EXIT_FAILURE);
}
SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
exit(EXIT_SUCCESS);
}
```

Running the example code produces the following output:

```
Allocated an environment handle.Set application to ODBC 3.
Allocated Database handle.
Connecting to database.
Connected to database.
Creating table.
Created table.
Creating prepared statement
Created prepared statement.
Bound CustIDs array to prepared statement
Bound CustNames array to prepared statement
Bound phoneNums array to prepared statement
Adding Batch #1...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Adding Batch #2...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Adding Batch #3...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Adding Batch #4...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Adding Batch #5...Params processed: 4
Results of inserting each row:
  Row 1 inserted successfully
  Row 2 inserted successfully
  Row 3 inserted successfully
  Row 4 inserted successfully
Commit Transaction
Free handles.
```

Error Handling During Batch Loads

When loading individual batches, you can find information on how many rows were accepted and what rows were rejected (see Tracking Load Status (ODBC) for details). Other errors, such as disk space errors, do not occur while inserting individual batches. This behavior is caused by having a single COPY statement perform the loading of multiple consecutive batches. Using the single COPY statement makes the batch load process perform much faster. It is only when the COPY statement closes that the batched data is committed and HP Vertica reports other types of errors.

Your bulk loading application should check for errors when the COPY statement closes. Normally, you force the COPY statement to close by calling the `SQLEndTran()` function to end the transaction. You can also force the COPY statement to close by closing the cursor using the `SQLCloseCursor()` function, or by setting the database connection's AutoCommit property to true before inserting the last batch in the load.

Note: The COPY statement also closes if you execute any non-insert statement. However having to deal with errors from the COPY statement in what might be an otherwise-unrelated query is not intuitive, and can lead to confusion and a harder to maintain application. You should explicitly end the COPY statement at the end of your batch load and handle any errors at that time.

Loading Batches in Parallel

To load batches in parallel, you need to establish a thread for each parallel batch you want to load. Then for each thread, set the batch size, prepare the insert, and execute the batch insert. The following code samples illustrate this.

```
#define THREAD_COUNT 10#define ROWS_PER_THREAD 100000
#define BATCH_SIZE 10000
void *BatchInsert(void *arg){
    SQLRETURN rc = SQL_SUCCESS;
    int i, j;
    SQLINTEGER *intValArray = NULL;
    SQLINTEGER lRows=BATCH_SIZE;
    // connect to db, allocate statement, set auto-commit off – skipped
    intValArray = (SQLINTEGER*) malloc(sizeof(*intValArray) * BATCH_SIZE);
    rc = SQLSetStmtAttr( hStmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)lRows, 0 );

// prepare insert
    rc = SQLPrepare (hStmt, (SQLTCHAR*)"insert into mt_test values(?)", SQL_NTS) ;
    rc = SQLBindParameter(hStmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0, (SQL
POINTER)intValArray, sizeof(*intValArray), NULL);
    for (i = 0; i < ROWS_PER_THREAD; i) {
        for (j = 0; j < BATCH_SIZE; j++) {
            intValArray[j] = (SQLINTEGER) ++i;
        }
        rc = SQLExecute(hStmt);
    }
    rc = SQLEndTran (SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
}
int runMT(int argc, char **argv) {
    pthread_t t[THREAD_COUNT];
    void *trc;
    for (int i=0;i<THREAD_COUNT;++i){
        pthread_create(&t[i], NULL, BatchInsert, argv[0]);
    }
    for (int i=0;i<THREAD_COUNT;++i){
        pthread_join(t[i], &trc);
    }
    free(trc);
    return 0;
}
```

Using the COPY Statement

The COPY statement lets you bulk load data from a file on stored on a database node into the HP Vertica database. This method is the most efficient way to load data into HP Vertica because the file resides on the database server. One drawback is that only a database superuser can use COPY, since it requires privilege in order to access the filesystem of the database node.

One drawback of using COPY instead of performing batch loads is that you can only get results of the load (the number of accepted and rejected rows) when the COPY statement has finished. With batch loads, you can monitor the progress as batches are inserted. The ability to monitor the progress of a load can be a useful feature if you want to stop loading if a large portion of the data is being rejected.

A primary concern when bulk loading data using COPY is deciding whether the data should be loaded directly into **ROS** using the DIRECT option, or by using the AUTO method (loading into **WOS** until it fills, then loading into ROS). You should load directly into the ROS when your transaction will load a large (more than 100MB of data or so) amount of data.

Note: The exceptions/rejections files are created on the client machine when the exceptions and rejected data modifiers are specified on the COPY command. Specify a local path and filename for these modifiers when executing a COPY query from the driver.

The following example loads data into the **WOS (Write Optimized Store)** until it fills, then stores additional data directly in **ROS (Read Optimized Store)**.

```
ret=SQLExecDirect(hdlStmt, (SQLCHAR*)"COPY customers "
    "FROM '/data/customers.txt' AUTO",SQL_NTS);
```

The following example loads data into the **ROS (Read Optimized Store**.

```
ret=SQLExecDirect(hdlStmt, (SQLCHAR*)"COPY customers "
    "FROM '/data/customers.txt' DIRECT",SQL_NTS);
```

See the COPY statement in the SQL Reference Manual for more information about its syntax and use.

The following example demonstrates using the COPY command.

```
// Some standard headers
#include <stdio.h>
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
// Helper function to determine if an ODBC function call returned
// successfully.
bool notSuccess(SQLRETURN ret) {
    return (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO);
}
int main()
{
    // Set up the ODBC environment
    SQLRETURN ret;
    SQLHENV hdlEnv;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(notSuccess(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(notSuccess(ret)) {
        printf("Could not set application version to ODBC3.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application to ODBC 3.\n");
    }
    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    // Connect to the database
    printf("Connecting to database.\n");
    const char *dsnName = "ExampleDB";

    // Note: User MUST be a database superuser to be able to access files on the
    // filesystem of the node.
    const char* userID = "dbadmin";
    const char* passwd = "password123";
    ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if(notSuccess(ret)) {
        printf("Could not connect to database.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Connected to database.\n");
    }

    // Disable AUTOCOMMIT
    printf("Disabling autocommit.\n");
    ret = SQLSetConnectAttr(hdlDbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, SQL_NTS);
    if(notSuccess(ret)) {
        printf("Could not disable autocommit.\n");
        exit(EXIT_FAILURE);
    }


    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);
    // Create table to hold the data
    SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
        SQL_NTS);
    SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers"
        "(Last_Name char(50) NOT NULL, First_Name char(50),Email char(50), "
        "Phone_Number char(15));",
        SQL_NTS);

    // Run the copy command to load data into ROS.
    ret=SQLExecDirect(hdlStmt, (SQLCHAR*)"COPY customers "
        "FROM '/data/customers.txt' DIRECT",
        SQL_NTS);
    if(notSuccess(ret)) {
        printf("Data was not successfully loaded.\n");
        exit(EXIT_FAILURE);
    } else {
        // Get number of rows added.
        SQLLEN numRows;
        ret=SQLRowCount(hdlStmt, &numRows);
        printf("Successfully inserted %d rows.\n", numRows);

    }

    // Done with batches, commit the transaction
    printf("Committing transaction\n");
    ret = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
    if(notSuccess(ret)) {
        printf("Could not commit transaction\n");
    }  else {
        printf("Committed transaction\n");
    }

    // Clean up
    printf("Free handles.\n");
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

The example prints the following when run:

```
Allocated an environment handle.
Set application to ODBC 3.
Connecting to database.
Connected to database.
Disabling autocommit.
Successfully inserted 10001 rows.
Committing transaction
Committed transaction
Free handles.
```

Streaming Data From the Client Using COPY LOCAL

The LOCAL option of the SQL COPY statement lets you stream data from a file on a client system to your HP Vertica database. This statement works through the ODBC driver, making the task of transferring data files from the client to the server much easier.

The LOCAL option of COPY works transparently through the ODBC driver. Just have your client application execute a statement containing a COPY LOCAL statement, and the ODBC driver will read and stream the data file from the client to the server. For example:

```
// Some standard headers
#include <stdio.h>
```

```
#include <stdlib.h>
// Only needed for Windows clients
// #include <windows.h>
// Standard ODBC headers
#include <sql.h>
#include <sqltypes.h>
#include <sqlext.h>
int main()
{
    // Set up the ODBC environment
    SQLRETURN ret;
    SQLHENV hdlEnv;
    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not allocate a handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Allocated an environment handle.\n");
    }
    // Tell ODBC that the application uses ODBC 3.
    ret = SQLSetEnvAttr(hdlEnv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not set application version to ODBC3.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application to ODBC 3.\n");
    }
    // Allocate a database handle.
    SQLHDBC hdlDbc;
    ret = SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not aalocate a database handle.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Set application to ODBC 3.\n");
    }
    // Connect to the database
    printf("Connecting to database.\n");
    const char *dsnName = "ExampleDB";
    const char* userID = "dbadmin";
    const char* passwd = "password123";
    ret = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,
        SQL_NTS,(SQLCHAR*)userID,SQL_NTS,
        (SQLCHAR*)passwd, SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Could not connect to database.\n");
        exit(EXIT_FAILURE);
    } else {
        printf("Connected to database.\n");
    }

    // Set up a statement handle
    SQLHSTMT hdlStmt;
    SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);


    // Create table to hold the data
    SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE IF EXISTS customers",
        SQL_NTS);
    SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers"
        "(Last_Name char(50) NOT NULL, First_Name char(50),Email char(50), "
        "Phone_Number char(15));",
        SQL_NTS);

    // Run the copy command to load data into ROS.
    ret=SQLExecDirect(hdlStmt, (SQLCHAR*)"COPY customers "
        "FROM LOCAL '/home/dbadmin/customers.txt' DIRECT",
        SQL_NTS);
    if(!SQL_SUCCEEDED(ret)) {
        printf("Data was not successfully loaded.\n");
        exit(EXIT_FAILURE);
    } else {
        // Get number of rows added.
        SQLLEN numRows;
        ret=SQLRowCount(hdlStmt, &numRows);
        printf("Successfully inserted %d rows.\n", numRows);
    }


    // COPY commits automatically, unless it is told not to, so
    // there is no need to commit the transaction.

    // Clean up
    printf("Free handles.\n");
    ret = SQLDisconnect( hdlDbc );
    if(!SQL_SUCCEEDED(ret)) {
        printf("Error disconnecting. Transaction still open?\n");
        exit(EXIT_FAILURE);
    }
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);
    exit(EXIT_SUCCESS);
}
```

This example is essentially the same as the example shown in Using the COPY Statement, except it uses the COPY statement's LOCAL option to load data from the client system rather than from the filesystem of the database node.

Note: On Windows clients, the path you supply for the COPY LOCAL file is limited to 216 characters due to limitations in the Windows API.

See Also

- COPY LOCAL

Programming JDBC Client Applications

The HP Vertica JDBC driver provides you with a standard JDBC API. If you have accessed other databases using JDBC, you should find accessing HP Vertica familiar. This section explains how to use the JDBC to connect your Java application to HP Vertica.

You must first install the JDBC client driver on all client systems that will be accessing the HP Vertica database. For installation instructions, see Installing the HP Vertica Client Drivers.

For more information about JDBC:

- Vertica Analytics Platform JDBC Driver JavaDoc (HP Vertica extensions)
- An Introduction to JDBC, Part 1

JDBC Feature Support

The HP Vertica JDBC driver complies with the JDBC 4.0 standards (although it does not implement all of the optional features in them). Your application can use the `DatabaseMetaData` class to determine if the driver supports a particular feature it wants to use. In addition, the driver implements the `Wrapper` interface, which lets your client code discover HP Vertica-specific extensions to the JDBC standard classes, such as `VerticaConnection` and `VerticaStatement` classes.

Some important facts to keep in mind when using the HP Vertica JDBC driver:

- Cursors are forward only and are not scrollable. Result sets cannot be updated.
- A connection supports executing a single statement at any time. If you want to execute multiple statements simultaneously, you must open multiple connections.
- Because HP Vertica does not have stored procedures, `CallableStatement` is not supported. The `DatabaseMetaData.getProcedures()` and `.getProcedureColumns()` methods return information about SQL functions (including **User Defined Functions**) instead of stored procedures.

Multiple SQL Statement Support

The HP Vertica JDBC driver can execute strings containing multiple statements. For example:

```
stmt.executeUpdate("CREATE TABLE t(a INT);INSERT INTO t VALUES(10);");
```

Only the `Statement` interface supports executing strings containing multiple SQL statements. You cannot use multiple statement strings with `PreparedStatement`. COPY statements that copy a file from a host file system work in a multiple statement string. However, client COPY statements (COPY FROM STDIN) do not work.

Multiple Batch Conversion to COPY Statements

The HP Vertica JDBC driver converts all batch inserts into HP Vertica COPY statements. If you turn off your JDBC connection's AutoCommit property, the JDBC driver uses a single COPY statement to load data from sequential batch inserts which can improve load performance by reducing overhead. See Batch Inserts Using JDBC Prepared Statements for details.

Multiple JDBC Version Support

The HP Vertica JDBC driver implements both JDBC 3.0 and JDBC 4.0 compliant interfaces. The interface that the driver returns to your application depends on the JVM version on which it is running. If your application is running on a 5.0 JVM, the driver supplies your application with JDBC 3.0 classes. If your application is running on a 6.0 or later JVM, the driver supplies it with JDBC 4.0 classes.

Updating Application Code From Previous Driver Versions

If you have a client application that was written using a previous version of the HP Vertica JDBC driver, you may need to alter its code to work with newer driver versions, or you may want to change your application to take advantage of new driver features. The topics in this section explain the changes that have been made to the JDBC driver from previous driver versions.

Updating Client Code From 4.1 or Earlier JDBC Driver Versions

The HP Vertica version 5.1 client drivers were rewritten to improve standards compliance and reliability. As a result, some HP Vertica-specific features and past incompatibilities were eliminated. If you client code worked with a pre-5.1 version of the HP Vertica JDBC driver, you must update it to work with the new driver versions. The following topics explain these updates.

Driver Package and Interface Name Changes

The name of the HP Vertica client driver package has changed. Previously, the HP Vertica JDBC driver classes were located in `com.vertica`. They are now in `com.vertica.jdbc`.

Loading the 4.1 Driver	Loading the 5.1 Driver
`Class.forName("com.vertica.Driver");`	`Class.forName("com.vertica.jdbc.Driver");`

Note: The HP Vertica version 7.0 JDBC driver introduces JDBC 4.0 compatibility which includes the JNDI service registration that eliminates the need to manually load the JDBC driver. See New Features in the HP Vertica Version 7.0 JDBC Driver for details.

Interface Name Changes

The following table lists the interfaces in the HP Vertica JDBC client library whose names have changed.

4.1 Interface Name	5.1 Interface Name
`PGConnection`	`VerticaConnection`
`PGStatement`	`VerticaStatement`

Normally, you reference these interfaces only when casting a `Connection` or `Statement` object to access HP Vertica-specific methods or properties. Many of the HP Vertica-specific methods and properties have been removed from the version 5.1 client drivers in favor of JDBC-standard methods and properties. You should not need to cast to these interfaces as often when using the new client drivers. See Converting From PGConnection to VerticaConnection and Converting From PGStatement to VerticaStatement for more information.

Removed Classes

The HP Vertica version 5.1 JDBC driver has removed several legacy classes.

Previously, many HP Vertica-specific methods would throw a `PSQLException` when encountering an error. This class has been removed. Instead, all methods now throw a standard `SQLException`.

Note: Note: The HP Vertica version 7.0 JDBC driver introduces a set of `SQLException` subclasses that more specifically describe error situations. See New Features in the HP Vertica Version 7.0 JDBC Driver for more information.

The `PGInterval` class has been replaced with a pair of HP Vertica interval classes: `VerticaDayTimeInterval` (which represents all ten types of day/time intervals) and `VerticaYearMonthInterval` (which represents all three types of year/month intervals). See Using Intervals with JDBC for details.

The `PGMoney` class has been removed from the driver.

Converting From PGConnection to VerticaConnection

The `VerticaConnection` interface replaces the `PGConnection` interface in the version 5.1 JDBC driver. The `PGConnection` interface contained HP Vertica-specific extensions to the standard JDBC `Connection` interface. `VerticaConnection` does not implement some of `PGConnection`'s methods to make it more compliant with the JDBC standards.

Property Setters and Getters

The `PGConnection` interface had several specific setters and getters for some connection properties. These have been removed from `VerticaConnection` and replaced with properties, as described in the following table:

4.1 Methods	5.1 Property Name	Description
`getBatchDirectInsert()setBatchDirectInsert()`	`DirectBatchInsert`	Controls whether data is inserted directly into **ROS**, or into **WOS**.
`getMaxLRSMemory()setMaxLRSMemory()`	`ResultBufferSize`	Controls the size of the memory buffer the client uses when retrieving a result set stream.

Instead of these non-standard methods, use the `VerticaConnection.getProperty()` and `VerticaConnection.setProperty()` methods to access these properties, or set them when creating the database connection. See Connection Properties and Setting and Getting Connection Property Values for more information.

Deprecated Methods

The following methods of the `PGConnection` class were deprecated in the version 4.1 drivers, and are not implemented by `VerticaConnection`:

Removed Methods	Reason
`getBatchInsertEnforceLength()setBatchInsertEnforceLength()`	In the HP Vertica version 5.1 drivers, batch inserts always enforce column-width limitations. This makes batch inserts consistent with non-batch inserts, which always enforce width limitations. If you do not want to batch load errors to occur due to data being too wide for the column, you can either: • Truncate the data to the column's width before attempting to insert it • Use the `VerticaCopyStream` class to execute a COPY FROM STDIN statement, without an ENFORCELENGTH parameter.
`getBinaryBatchInsert()setBinaryBatchInsert()`	The new JDBC driver now uses a single batch insert protocol rather than having separate binary and text modes to insert data.
`getLocale()setLocale()`	These methods are not implemented by `VerticaConnection`. Instead, you set the connection's locale using the SET LOCALE SQL statement. See Setting the Locale for JDBC Sessions for details.
`getStreamingLRS()setStreamingLRS()`	The new JDBC driver always uses streaming result sets rather than optionally caching the results on the client.

Removed Methods	Reason

`getEncoding()setEncoding()`
`getFastPathAPI()`
`getLargeObjectAPI()`
`getManagedBatchInsert()`
`setManagedBatchInsert()`
`getObject()`
`getPGType()`
`getPrepareThreshold()`
`setPrepareThreshold()`
`getSQLType()`
`getUse35CopyFormat()`
`setUse35CopyFormat()`
`getUse35CopyParameters()`
`setUser35CopyParameters()`
`addDataType()`
	All of these methods were previously deprecated and have been removed from the version 5.1 JDBC driver.

Savepoint Support

The version 5.1 JDBC driver implements the `Savepoint` interface, allowing you to use **savepoints** to segment your transactions and later roll back a portion of the transaction.

Updatable Result Set Changes

The `createStatement()` method in the 4.1 and earlier JDBC drivers accepted the `ResultSet.CONCUR_UPDATABLE` flag to create an updatable result set. However, the driver's support of this feature was limited. The 5.1 JDBC driver does not support updatable result sets. The `createStatement()` method now accepts just `ResultSet.TYPE_FORWARD_ONLY` and `ResultSet.CONCUR_READ_ONLY` flags. It throws an exception if you pass it other flags.

Converting From PGStatement to VerticaStatement

The `VerticaStatement` interface contains the HP Vertica-specific extensions of the standard JDBC `Statement` interface. In previous versions of the HP Vertica JDBC driver, this interface was named `PGStatement`. In addition to the name change, a some methods have been removed from this interface.

Deprecated Methods

`VerticaStatement` does not implement the following `PGStatement` methods because they are obsolete:

- `executeCopyIn()`
- `executeCopyOut()`
- `finishLoad()`
- `getLastOID()`

Bulk Loading Method Changes

`VerticaStatement` does not implement the following `PGStatement` bulk loading methods:

- `addStreamtoCopyIn()`
- `startCopyIn()`
- `finishCopyIn()`

Instead, the new `VerticaCopyStream` class implements a stream copying feature. See Streaming Data Via JDBC.

Connection Property Setters and Getters

In previous versions of the drivers, many of the connection property setters and getters on the `PGConnection` interface were also implemented in the `PGStatement` interface. These methods allowed you to change some of the connection parameters for the statement, letting you override their settings on the `PGConnection` object used to create the statement. `VerticaStatement` does not implement these setters and getters helping to prevent potential confusion and difficult-to-debug issues caused by having different statements on the same connection having their own connection properties.

`VerticaStatement` objects cache two properties independently: DirectBatchInsert and ResultBufferSize. Once a `VerticaStatement` object is instantiated, it stores the values of these properties that were set on the `VerticaConnection` object you used to create them. The object retains these values even if you later change the property on the `VerticaConnection` object.

For example, in the following code:

```
// Set the DirectBatchInsert property
((VerticaConnection) conn).setProperty("DirectBatchInsert", true );
// Create a statement object
Statement stmtA = conn.createStatement();
// Change the direct batch insert setting and create another
// statement.
((VerticaConnection) conn).setProperty("DirectBatchInsert", false );
Statement stmtB = conn.createStatement();
```

The `stmtA` object has its DirectBatchInsert property set to true, since that was the property's value on the `VerticaConnection` used to instantiate it. Since this property is specific to the statement, `stmtA`'s DirectBatchInsert property remains unchanged when the connection's DirectBatchInsert property changes to false later.

Note: The `VerticaStatement` interface does not cache any of the other connection properties. If you change another property (such as Locale) on a `VerticaConnection` object, the change affects all of the `VerticaStatement` objects instantiated using that connection object.

Multiple Statement Support

The previous JDBC driver's implementation of the `Statement` interface did not support executing SQL strings containing multiple statements. The new driver's implementation does support multiple statements. For example:

```
stmt.executeUpdate("CREATE TABLE t(a INT);INSERT INTO t VALUES(10);");
```

Only the `Statement` interface supports executing strings containing multiple SQL statements. You cannot use multiple statement strings with `PreparedStatement`. COPY statements that copy a file from a host file system work in a multiple statement string. However, client COPY statements (COPY FROM STDIN) do not work.

Connection Property Changes

JDBC connection properties let you control the behavior of your application's connection to the database. The HP Vertica version 5.1 JDBC driver has removed some old properties, renamed several others, and added some new properties.

New Connection Properties

The following properties have been added to the HP Vertica version 5.1 JDBC driver.

Property	Description
ConnSettings	Contains SQL commands to be executed when the database connection is established.
LogLevel	Sets the types of messages that the client writes to a log file.
LogNameSpace	Limits the messages written to the log to only those generated by classes in the given namespace.
LogPath	Sets the path where the log file is written.
TransactionIsolation	Previously could only be accessed using a getter and setter method on the `PGConnection` object. It can now be set using the same methods as other connection properties.
ThreePartNaming	Controls whether the database metadata uses the database name as the catalog name. This property is only used for backwards compatibility with some client software.

See Connection Properties for more information about these new properties.

Renamed Properties

Several properties have been renamed to make connection properties more consistent across the different client libraries:

4.1 Property Name	5.1 Name	Description
defaultAutoCommit	Autocommit	Sets whether statements automatically commit themselves.
SessionLabel	Label	Identifies the connection on the server.
maxLRSMemory	ResultBufferSize	Sets the buffer size the driver uses when retrieving a result set from the server. In addition, the default value has been changed from 8MB to 8KB.

Removed Connection Properties

The following table lists the connection properties have been removed from the HP Vertica version 5.1 JDBC driver:

Parameter	Description
batchInsertEnforceLength	Controlled whether inserting data that is too wide for its column would cause an error. Removed to make batch inserts consistent with other types of data inserts. Attempting to insert data that is too wide for a column always results in an error.
batchInsertRecordTerminator	Set the record terminator for a batch insert. This property was deprecated and was only available for backwards compatibility.
binaryBatchInsert	Controlled whether batched data inserts were transmitted as binary data or converted to string. Removed because the driver now uses NATIVE VARCHAR transfer for all data types.
BinaryDataTransfer	Controlled whether data was transmitted between the server and the client in binary format. Removed because the driver now uses NATIVE VARCHAR to transfer data
KeepAlive	Caused the network connection to send keepalive packets to ensure the connection remained open during idle periods. The new JDBC driver always sends keepalive packets.
Locale	Set the locale for the connection. Instead of a property, use a query to set the locale. You can include a SQL statement in the ConnSettings property to set the locale as soon as the JDBC driver connects to the database. See Setting the Locale for JDBC Sessions.

Parameter	Description
prepareThreshold	Controlled how many times a statement would be executed before the server would automatically convert it to a server-side prepared statement. Removed since this functionality was deprecated.
streamingLRS	Controlled whether results were streamed to the client as it requested data, or if all data in a result set was sent to the client, which cached it in a local file. Removed since the new driver always uses streaming mode.

New Features in the HP Vertica Version 7.0 JDBC Driver

The HP Vertica Version 7.0 JDBC driver introduces JDBC 4.0 compatibility. This driver is completely backwards compatible with the earlier driver versions, so you do not need to change the code of your Java client application or even recompile it. You may wish to update you client application to take advantage of some of the features offered by the new driver, which are explained below.

JNDI Service Registration

In prior versions of the HP Vertica JDBC driver, your client application needed to load the JDBC driver by calling `Class.forName("com.vertica.jdbc.Driver")` before using it. The new driver implements JNDI service registration, so the JVM automatically finds and loads the JDBC driver when your application tries to use it. Therefore, your application no longer needs to call `Class.forName` before using the JDBC driver. There is no harm in having you application still call `Class.forName` so you do not need to remove this call if it is already there.

Note: If you want your application to remain compatible with the JDBC 3.0 driver so it can run in a Java 5 JVM, you should still have it call `Class.forName` to load the driver. You can remove the call if your application will only be run on Java 6 or later JVMs which use the JDBC 4.0 driver.

Exception Class Improvements

In the JDBC 3.0 standard, most errors were signaled by throwing a single exception class (`SQLException`). This single class makes it difficult for a client application to analyze errors. JDBC 4.0 introduces a hierarchy of subclasses of `SQLException` which provide more specific information to your application about the type of error that has occurred. With these new classes, your client application can better respond to errors, potentially resolving them itself.

For example, exception subclasses that inherit from the `SQLTransientException` subclass of `SQLException` (such as `SQLTransientConnectionException`) are thrown by the driver when it encounters an error that may be caused by a temporary condition (such as a timeout or a network issue that might resolve itself). Your client application could catch this exception and automatically retry the operation instead of immediately reporting a failure to the user.

The `SQLException` class and its subclasses now make iterating over multiple exceptions easier by implementing the `Iterator` interface.

See Handling Errors for details about using these new `SQLException` subclasses.

Wrapper Interface Support

The JDBC 4.0 standard introduces the `Wrapper` interface, which lets client applications discover vendor-specific extensions to the JDBC standards. Your client application can use this interface to find Vertica-specific extensions to standard JDBC classes such as `VerticaConnection` (an extension to the standard `Connection` class) and `VerticaPreparedStatement` (an extension to the `PreparedStatement` class) which implement the `Wrapper` interface.

Additional DatabaseMetaData Methods

Past versions of the JDBC standard contained the `DatabaseMetaData` interface which let client applications determine which features the JDBC driver and database implemented. JDBC 4.0 extends this interface with new methods that the HP Vertica JDBC driver implements. Among these new methods are `getFunctions()` and `getFunctionColumns()` which let your client application find which User Defined Functions (UDFs) are defined in the HP Vertica catalog, and determine their input and output values. See Developing and Using User Defined Extensions for more information about UDFs.

Improved Connection Pooling

The HP Vertica Version 7.0 JDBC driver implements the improvements in connection pooling defined in the JDBC 4.0 standard. These improvements include:

- new methods to determine if a connection is still valid.
- new methods to report information about the client back to the server.
- new callback methods to alert client applications to the expiration of a connection or statement.

Normally, your client application will not use this API directly. Instead, you use an application framework that handles connection pooling for you. See Using a Pooling Data Source for more information.

Native Connection Load Balancing Support

The HP Vertica Version 7.0 server, client libraries, and vsql client support native connection load balancing, which helps spread the resources used by client connections across all of the hosts in the database. See "About Native Connection Load Balancing" on page 1 in the Administrator's Guide for details. To support this feature, the JDBC driver has a new connection parameter named ConnectionLoadBalance to enable load balancing on the client side of the connection. See Enabling Native Connection Load Balancing in JDBC for more information.

Connection Failover Support

The JDBC driver now supports two forms of connection failover: DNS based and parameter based. This feature lets the JDBC driver automaitcally attempt to one or more backup hosts if ithe primary host does not respond to its connection request.

See JDBC Connection Failover for more information.

Creating and Configuring a Connection

Before your Java application can interact with HP Vertica, it must create a connection. Connecting to HP Vertica via JDBC is similar to connecting to most other databases.

Importing SQL Packages

Before creating a connection, you must import the Java SQL packages. The easiest way to do this to import the entire package using a wildcard:

```
import java.sql.*;
```

You may also want to import the `Properties` class. You can use an instance of this class to pass connection properties when instantiating a connection, rather than encoding everything within the connection string:

```
import java.util.Properties;
```

If your application needs to run in a Java 5 JVM, it will use the older JDBC 3.0-compliant driver. This driver requires you to manually load the HP Vertica JDBC driver using the `Class.forName()` method:

```
// Only required for old JDBC 3.0 driver
try {
       Class.forName("com.vertica.jdbc.Driver");
} catch (ClassNotFoundException e) {
       // Could not find the driver class. Likely an issue
       // with finding the .jar file.
       System.err.println("Could not find the JDBC driver class.");
       e.printStackTrace();
       return; // Exit. Cannot do anything further.
}
```

If your application runs in a Java 6 or later JVM, the JVM automatically loads the HP Vertica JDBC 4.0-compatible driver without requiring the call to `Class.forName`. There is no harm in making this call, so if you want your application to be compatible with both Java 5 and Java 6 (or later) JVMs, it can still call `Class.forName`.

Opening the Connection

With SQL packages imported, you are ready to create your connection by calling the `DriverManager.getConnection()` method. You supply this method with at least the following information:

- The name of a host in the database cluster
- The port number for the database

- The name of the database
- The username of a database user account
- The password of the user (if the user has a password)

The first three parameters are always supplied as part of the connection string (a URL that tells the JDBC driver where to find the database). The format of the connection string is:

```
"jdbc:vertica://VerticaHost:portNumber/databaseName"
```

The first portion of the connection string selects the HP Vertica JDBC driver, followed by the location of the database.

The last two parameters, username and password, can be given to the JDBC driver in one of three ways:

- as part of the connection string. The parameters are encoded similarly to URL parameters:

  ```
  "jdbc:vertica://VerticaHost:portNumber/databaseName?user=username&password=password"
  ```

- passed as separate parameters to `DriverManager.getConnection()`:

  ```
  Connection conn = DriverManager.getConnection(
          "jdbc:vertica://VerticaHost:portNumber/databaseName",
          "username", "password");
  ```

- passed in a `Properties` object:

  ```
  Properties myProp = new Properties();myProp.put("user", "username");
  myProp.put("password", "password");
  Connection conn = DriverManager.getConnection(
         "jdbc:vertica://VerticaHost:portNumber/databaseName", myProp);
  ```

Of these three methods, the `Properties` object is the most flexible because it makes passing additional connection properties to the `getConnection()` method easy. See Connection Properties and Setting and Getting Connection Property Values for more information about the additional connection properties.

The `getConnection()` method throws a `SQLException` or one of its subclasses if there is any problem establishing a connection to the database, so you want to enclose it within a try-catch block, as shown in the following complete example of establishing a connection.

```
import java.sql.*;
import java.util.Properties;
public class VerySimpleVerticaJDBCExample {
    public static void main(String[] args) {
        /*
         * If your client needs to run under a Java 5 JVM, It will use the older
```

```
         * JDBC 3.0-compliant driver, which requires you manually load the
         * driver using Class.forname
         */
        /*
         * try { Class.forName("com.vertica.jdbc.Driver"); } catch
         * (ClassNotFoundException e) { // Could not find the driver class.
         * Likely an issue // with finding the .jar file.
         * System.err.println("Could not find the JDBC driver class.");
         * e.printStackTrace(); return; // Bail out. We cannot do anything
         * further. }
         */
        Properties myProp = new Properties();
        myProp.put("user", "dbadmin");
        myProp.put("password", "vertica");
        myProp.put("loginTimeout", "35");
        myProp.put("binaryBatchInsert", "true");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                    "jdbc:vertica://docd04.verticacorp.com:5433/vmart", myProp);
            System.out.println("Connected!");
            conn.close();
        } catch (SQLTransientConnectionException connException) {
            // There was a potentially temporary network error
            // Could automatically retry a number of times here, but
            // instead just report error and exit.
            System.out.print("Network connection issue: ");
            System.out.print(connException.getMessage());
            System.out.println(" Try again later!");
            return;
        } catch (SQLInvalidAuthorizationSpecException authException) {
            // Either the username or password was wrong
            System.out.print("Could not log into database: ");
            System.out.print(authException.getMessage());
            System.out.println(" Check the login credentials and try again.");
            return;
        } catch (SQLException e) {
            // Catch-all for other exceptions
            e.printStackTrace();
        }
    }
}
```

Notes

- When you disconnect a user session, any uncommitted transactions are automatically rolled back.
- If your database is not compliant with your HP Vertica license terms, HP Vertica issues a `SQLWarning` when you establish the connection to the database. You can retrieve this warning using the `Connection.getWarnings()` method. See Managing Your License Key in the Administrator's Guide for more information about complying with your license terms.

JDBC Connection Properties

You use connection properties to configure the connection between your JDBC client application and your HP Vertica database. The properties provide the basic information about the connections, such as the server name and port number to use to connect to your database. They also let you tune the performance of your connection and enable logging.

There are three ways to set a connection property:

- Including the property name and value as part of the connection string you pass to the `DriverManager.getConnection()` method.
- Setting the properties in a `Properties` object, then passing it to the `DriverManager.getConnection()`method.
- Using the `VerticaConnection.setProperty()` method. Only the connection properties that can be changed after the connection has been established can be changed using the `setProperty()` method.

In addition, some of the standard JDBC connection properties have getters and setters on the `Connection` interface (such as `Connection.setAutocommit()`).

The following tables list the properties supported by the HP Vertica JDBC driver, and explain which are required in order for the connection to be established.

Connection Properties

The properties in the following table can only be set before you open the connection to the database. Two of them are required for every connection.

Property	**Description**
ConnSettings	A string containing SQL statements that the JDBC driver automatically runs after it connects to the database. This property is useful to set the locale, set the schema search path, or perform other configuration that the connection requires. **Required?:** No **Default Value:** none
Label	Sets a label for the connection on the server. This value appears in the session_id column of the V_MONITOR.SESSIONS system table. **Required?:** No **Default Value:** jdbc-*driver_version-random_number*

Property	Description
LoginTimeout	The number of seconds HP Vertica waits for a connection to be established to the database before throwing a `SQLException`. When set to 0 (the default) there is no TCP timeout. **Required?:** No **Default Value:** 0
SSL	When set to true, uses SSL to encrypt the connection to the server. HP Vertica needs to be configured to handle SSL connections before you can establish an SSL-encrypted connection to it. See Implementing SSL in the Administrator's Guide. **Required?:** No **Default Value:** false
Password	The password to use to log into the database. **Required?:** Yes **Default Value:** none
User	The database user name to use to connection to the database. **Required?:** Yes **Default Value:** none
ConnectionLoadBalance	A Boolean indicating whether the client is willing to have its connection redirected to another host in the HP Vertica database. This setting only has an effect if the server has also enabled connection load balancing. See About Native Connection Load Balancing in the Administrator's Guide for more information about native connection load balancing. **Required?:** No **Default Value:** false
BackupServerNode	A string containing the host name or IP address of one or more hosts in the database that he client should attempt to connect to if the connection to the host specified in the connection string times out. The host name or IP address can also include a colon followed by the port number for the database. If no port number is specified, the client uses the standard port number (5433) . Separate multiple host name or IP address entries with commas. **Required?:** No **Default Value:** none

General Properties

The following properties can be set after the connection is established. None of these properties are required.

Property	Description
AutoCommit	Controls whether the connection automatically commits transactions. Set this parameter to false to prevent the connection from automatically committing its transactions. You often want to do this when you are bulk loading multiple batches of data and you want the ability to roll back all of the loads if an error occurs. **Note:** This property was called defaultAutoCommit in previous versions of the HP Vertica JDBC driver. **Set After Connection:** `Connection.setAutoCommit()` **Default Value:** true
DirectBatchInsert	Determines whether a batch insert stored data directly into **ROS** (true) or using AUTO mode (false). When loading data using AUTO mode, HP Vertica inserts the data first into the **WOS**. If the WOS is full, then HP Vertica inserts the data directly into **ROS**. See the COPY statement for more details. **Set After Connection:** `VerticaConnection.setProperty()` **Default Value:** false
ReadOnly	When set to true, makes the data connection read-only. Any queries attempting to update the database using a read-only connection cause a `SQLException`. **Set After Connection:** `Connection.setReadOnly()` **Default Value:** false
ResultBufferSize	Sets the size of the buffer the HP Vertica JDBC driver uses to temporarily store result sets. **Note:** This property was named maxLRSMemory in previous versions of the HP Vertica JDBC driver. **Set After Connection:** `VerticaConnection.setProperty()` **Default Value:** 8912 (8KB)

Property	Description
SearchPath	Sets the schema search path for the connection. The value for this property is a string containing a comma-separated list of schema names. See Setting Search Paths in the Administrator's Guide for more information on the schema search path. **Set After Connection:** `VerticaConnection.setProperty()` **Default Value:** "$user", public, v_catalog, v_monitor, v_internal
ThreePartNaming	A Boolean that controls how DatabaseMetaData reports the catalog name. When set to true, the database name is returned as the catalog name in the database metadata. When set to false, NULL is returned as the catalog name. Enable this option if your client software expects to be able to get the catalog name from the database metadata and use it as part of a three-part name reference. **Set After Connection:** `VerticaConnection.setProperty()` **Default Value:** true
TransactionIsolation	Sets the isolation level of the transactions that use the connection. See Changing the Transaction Isolation Level for details. **Note:** In previous versions of the HP Vertica JDBC driver, this property was only available using a getter and setter on the `PGConnection` object. It can now be set like other connection properties. **Set After Connection:** `Connection.setTransactionIsolation()` **Default Value:** TRANSACTION_READ_COMMITTED

Logging Properties

The properties that control client logging must be set before the connection is opened. None of these properties are required, and none can be changed after the `Connection` object has been instantiated.

Property	Description
LogLevel	Sets the type of information logged by the JDBC driver. The value is set to one of the following values: • `"DEBUG"` • `"ERROR"` • `"TRACE"` • `"WARNING"` • `"INFO"` • `"OFF"` **Default Value:** `"OFF"`
LogNameSpace	Restricts logging to just messages generated by a specific packages. Valid values are: • `com.vertica` (all messages generated by the JDBC driver) • `com.vertica.jdbc` (all messages generated by the top-level JDBC API) • `com.vertica.jdbc.core` (connection and statement settings) • `com.vertica.jdbc.io` (client/server protocol messages) • `com.vertica.jdbc.util` (miscellaneous utilities) • `com.vertica.jdbc.dataengine` (query execution and result set iteration). **Default Value:** none
LogPath	Sets the path where the log file is written. **Default Value:** The current working directory

Kerberos Connection Parameters

Use the following parameters to set the service and host name principals for client authentication using Kerberos.

Parameters	Description
JAASConfigName	Provides the name of the JAAS configuration that contains the JAAS Krb5LoginModule and its settings. **Default Value:** verticajdbc
KerberosServiceName	Provides the service name portion of the HP Vertica Kerberos principal; for example: `vertica/host@EXAMPLE.COM` **Default Value:** vertica
KerberosHostname	Provides the instance or host name portion of theHP Vertica Kerberos principal; for example: `vertica/host@EXAMPLE.COM` **Default Value:** Value specified in the servername connection string property

Key/Value API Connection Parameters

Use the following parameters to set properties to enable and configure the connection for Key/Value lookups.

Parameters	Description
EnableRoutableQueries	Enables Key/Value lookup. See About the JDBC Key/Value API **Default Value:** false
FailOnMultiNodePlans	If the query plan requires more than one node, then the query fails. Only applicable when EnableRoutableQueries = true. **Default Value:** true
MetadataCacheLifetime	The time in seconds to keep projection metadata. Only applicable when EnableRoutableQueries = true. **Default Value:**
MaxPooledConnections	Cluster-wide maximum number of connections to keep in the VerticaRoutableConnection’s internal pool. Only applicable when EnableRoutableQueries = true. **Default Value:** 20

Parameters	Description
MaxPooledConnections PerNode	Per-node maximum number of connections to keep in the VerticaRoutableConnection's internal pool. Only applicable when EnableRoutableQueries = true. **Default Value:** 5

Note: The `VerticaConnection.setProperty()` method can also be used to set properties that have standard JDBC Connection setters, such as AutoCommit.

For information about manipulating these attributes, see Setting and Getting Connection Property Values.

Setting and Getting Connection Property Values

There are three ways to set a connection property:

- Including the property name and value as part of the connection string you pass to the `DriverManager.getConnection()` method.
- Setting the properties in a `Properties` object, then passing it to the `DriverManager.getConnection()`method.
- Using the `VerticaConnection.setProperty()` method. Only the connection properties that can be changed after the connection has been established can be changed using the `setProperty()` method.

In addition, some of the standard JDBC connection properties have getters and setters on the `Connection` interface (such as `Connection.setAutocommit()`).

Setting Properties When Connecting

There are two ways you can set connection properties when creating a connection to HP Vertica:

- In the connection string, using the same URL parameter format that you can use to set the username and password. The following example sets the SSL connection parameter to true:

```
"jdbc:vertica://VerticaHost:5433/db?user=UserName&password=Password&ssl=true"
```

- In a `Properties` object that you pass to the `getConnection()` call. You will need to import the `java.util.Properties` class in order to instantiate a `Properties` object. Then you use the `put()` method to add the property name and value to the object:

```
Properties myProp = new Properties();
myProp.put("user", "ExampleUser");
myProp.put("password", "password123");
myProp.put("LoginTimeout", "35");
Connection conn;
try {
    conn = DriverManager.getConnection(
        "jdbc:vertica://VerticaHost:/ExampleDB", myProp);
} catch (SQLException e) {
    e.printStackTrace();
}
```

Note: The data type of all of the values you set in the Properties object are strings, even if the property value's data type is integer or Boolean.

Getting and Setting Properties After Connecting

The `VerticaConnection.getProperty()` method lets you get the value of some connection properties. You can use `VerticaConnection.setProperty()` method to change the value for properties that can be set after the database connection has been established. Since these methods are HP Vertica-specific, you must cast your `Connection` object to the `VerticaConnection` interface to be able to use them. To cast to `VerticaConnection`, you must either import it into your client application or use a fully-qualified reference (`com.vertica.jdbc.VerticaConnection`). The following example demonstrates getting and setting the value of the DirectBatchInsert property.

```
import java.sql.*;
import java.util.Properties;
import com.vertica.jdbc.*;
public class SetConnectionProperties {
    public static void main(String[] args) {
        // Note: If your application needs to run under Java 5, you need to
        // load the JDBC driver using Class.forName() here.
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        // Set DirectBatchInsert to true initially
        myProp.put("DirectBatchInsert", "true");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                            "jdbc:vertica://VerticaHost:5433/ExampleDB",
                            myProp);
            // Show state of the DirectBatchInsert property. This was set at the
            // time the connection was created.
            System.out.println("DirectBatchInsert state: "
                            + ((VerticaConnection) conn).getProperty(
                                            "DirectBatchInsert"));

            // Change it and show it again
            ((VerticaConnection) conn).setProperty("DirectBatchInsert", false);
            System.out.println("DirectBatchInsert state is now: " +
                                 ((VerticaConnection) conn).getProperty(
                                                 "DirectBatchInsert"));
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

When run, the example prints the following on the standard output:

```
DirectBatchInsert state: true
DirectBatchInsert state is now: false
```

Setting the Locale for JDBC Sessions

You set the locale for a connection while opening it by including a SET LOCALE statement in the ConnSettings property, or by executing a SET LOCALE statement at any time after opening the connection. Changing the locale of a `Connection` object affects all of the `Statement` objects you instantiated using it.

You can get the locale by executing a SHOW LOCALE query. The following example demonstrates setting the locale using ConnSettings and executing a statement, as well as getting the locale:

```
import java.sql.*;
import java.util.Properties;
public class GetAndSetLocale {
    public static void main(String[] args) {
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");


        // Set Locale to true en_GB on connection. After the connection
        // is established, the JDBC driver runs the statements in the
        // ConnSettings property.
        myProp.put("ConnSettings", "SET LOCALE TO en_GB");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                                "jdbc:vertica://VerticaHost:5433/ExampleDB",
                                myProp);

            // Execute a query to get the locale. The results should
            // show "en_GB" as the locale, since it was set by the
            // conn settings property.
            Statement stmt = conn.createStatement();
            ResultSet rs = null;
            rs = stmt.executeQuery("SHOW LOCALE");
            System.out.print("Query reports that Locale is set to: ");
            while (rs.next()) {
                System.out.println(rs.getString(2).trim());
            }

            // Now execute a query to set locale.
            stmt.execute("SET LOCALE TO en_US");

            // Run query again to get locale.
            rs = stmt.executeQuery("SHOW LOCALE");
            System.out.print("Query now reports that Locale is set to: ");
            while (rs.next()) {
                System.out.println(rs.getString(2).trim());
            }
            // Clean up
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Running the above example displays the following on the system console:

```
Query reports that Locale is set to: en_GB (LEN)
Query now reports that Locale is set to: en_US (LEN)
```

Notes:

- JDBC applications use a UTF-16 character set encoding and are responsible for converting any non-UTF-16 encoded data to UTF-16. Failing to convert the data can result in errors or the data being stored incorrectly.
- The JDBC driver converts UTF-16 data to UTF-8 when passing to the HP Vertica server and converts data sent by HP Vertica server from UTF-8 to UTF-16 .

Changing the Transaction Isolation Level

Changing the transaction isolation level lets you choose how transactions prevent interference from other transactions. By default, the JDBC driver matches the transaction isolation level of the HP Vertica server. The HP Vertica default transaction isolation level is `READ_COMMITTED`, which means any changes made by a transaction cannot be read by any other transaction until after they are committed. This prevents a transaction from reading data inserted by another transaction that is later rolled back.

HP Vertica also supports the `SERIALIZABLE` transaction isolation level. This level locks tables to prevent queries from having the results of their `WHERE` clauses changed by other transactions. Locking tables can have a performance impact, since only one transaction is able to access the table at a time.

A transaction retains its isolation level until it completes, even if the session's transaction isolation level changes mid-transaction. HP Vertica internal processes (such as the **Tuple Mover** and

refresh operations) and DDL operations are always run at `SERIALIZABLE` isolation level to ensure consistency.

You can change the transaction isolation level connection property after the connection has been established using the `Connection` object's setter (`setTransactionIsolation()`) and getter (`getTransactionIsolation()`). The value for transaction isolation property is an integer. The `Connection` interface defines constants to help you set the value in a more intuitive manner:

Constant	Value
`Connection.TRANSACTION_READ_COMMITTED`	2
`Connection.TRANSACTION_SERIALIZABLE`	8

Note: The `Connection` interface also defines several other transaction isolation constants (`READ_UNCOMMITTED` and `REPEATABLE_READ`). Since HP Vertica does not support these isolation levels, they are converted to `READ_COMMITTED` and `SERIALIZABLE`, respectively.

The following example demonstrates setting the transaction isolation level to SERIALIZABLE.

```
import java.sql.*;
import java.util.Properties;
public class SetTransactionIsolation {
    public static void main(String[] args) {
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                            "jdbc:vertica://VerticaHost:5433/ExampleDB",
                            myProp);
            // Get default transaction isolation
            System.out.println("Transaction Isolation Level: "
                            + conn.getTransactionIsolation());
            // Set transaction isolation to SERIALIZABLE
            conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
            // Get the transaction isolation again
            System.out.println("Transaction Isolation Level: "
                            + conn.getTransactionIsolation());
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Running the example results in the following being printed out to the console:

```
Transaction Isolation Level: 2Transaction Isolation Level: 8
```

Using a Pooling Data Source

A pooling data source uses a collection of persistent connections in order to reduce the overhead of repeatedly opening network connections between the client and server. Opening a new connection for each request is costly for both the server and the client than keeping a small pool of connections open constantly, ready to be used by new requests. When a request comes in, one of the pre-existing connections in the pool is assigned to it. Only if there are no free connections in the pool is a new connection created. Once the request is complete, the connection returns to the pool and waits to service another request.

The HP Vertica JDBC driver supports connection pooling as defined in the JDBC 4.0 standard. If you are using a J2EE-based application server in conjunction with HP Vertica, it should already have a built-in data pooling feature. All that is required is that the application server work with the `PooledConnection` interface implemented by HP Vertica's JDBC driver. An application server's pooling feature is usually well-tuned for the works loads that the server is designed to handle. See your application server's documentation for details on how to work with pooled connections. Normally, using pooled connections should be transparent in your code—you will just open connections and the application server will worry about the details of pooling them.

If you are not using an application server, or your application server does not offer connection pooling that is compatible with HP Vertica, you can use a third-party pooling library, such as the open-source c3p0 or DBCP libraries, to implement connection pooling.

Note: The Vertica Analytics Platform client driver's native connection load balancing feature works with third-party connection pooling supplied by application servers and third-party pooling libraries. See Enabling Native Connection Load Balancing in JDBC for more information.

Enabling Native Connection Load Balancing in JDBC

Native connection load balancing helps spread the overhead caused by client connections on the hosts in the HP Vertica database. Both the server and the client must enable native connection load balancing in order for it to have an effect. If both have enabled it, then when the client initially connects to a host in the database, the host picks a host to handle the client connection from a list of the currently up hosts in the database, and informs the client which host it has chosen. If the initially-contacted host did not choose itself to handle the connection, the client disconnects, then opens a second connection to the host selected by the first host. The connection process to this second host proceeds as usual—if SSL is enabled, then SSL negotiations begin, otherwise the client begins the authentication process. See About Native Connection Load Balancing in the Administrator's Guide for details.

To enable native load balancing on your client, set the ConnectionLoadBalance connection parameter to true. The following example demonstrates connecting to the database several times with native connection load balancing enabled, and fetching the name of the node handling the connection from the V_MONITOR.CURRENT_SESSION system table.

```
import java.sql.*;
import java.util.Properties;

import com.vertica.jdbc.DataSource;

public class ConnectionLoadBalancingExample {
    public static void main(String[] args) {
        /*
         * If your client needs to run under a Java 5 JVM, It will use the older
         * JDBC 3.0-compliant driver, which requires you manually load the
         * driver using Class.forname
         */

        Properties myProp = new Properties();
        myProp.put("user", "dbadmin");
        myProp.put("password", "vertica");
        // Enable connection load balancing on this client. Only works if it is set on th
e
        // server as well.
        myProp.put("ConnectionLoadBalance", 1);
        Connection conn;
        // Connect 4 times. See if we connect to a different node each time.
        for (int x=1; x <= 4; x++) {
            try {
                System.out.print("Connect attempt #" + x + "...");
                conn = DriverManager.getConnection(
                        "jdbc:vertica://docd03.verticacorp.com:5433/vmart", myProp);
                Statement stmt = conn.createStatement();

                // Query system to table to see what node we are connected to. Assume a s
ingle row
                // in response set.
                ResultSet rs = stmt.executeQuery("SELECT node_name FROM v_monitor.curren
t_session;");
                rs.next();
                System.out.println("Connected to node " + rs.getString(1).trim());
                conn.close();
            } catch (SQLException e) {
                // Catch-all for other exceptions
                System.out.println("Error!");
                e.printStackTrace();
            }
        }
    }
}
```

Running the above example produces the following output:

```
Connect attempt #1...Connected to node v_vmart_node0001
Status of load balance policy on server: roundrobin
Connect attempt #2...Connected to node v_vmart_node0002
Connect attempt #3...Connected to node v_vmart_node0003
Connect attempt #4...Connected to node v_vmart_node0001
```

JDBC Connection Failover

If a client application attempts to connect to a host in the Vertica Analytics Platform cluster that is down, the connection attempt fails when using the default connection configuration. This failure usually returns an error to the user. The user must either wait until the host recovers and retry the connection or manually edit the connection settings to choose another host.

Due to Vertica Analytics Platform's distributed architecture, you usually do not care which database host handles a client application's connection. You can use the client driver's connection failover feature to prevent the user from getting connection errors when the host specified in the connection settings is unreachable. It gives you two ways to let the client driver automatically attempt to connect to a different host if the one specified in the connection parameters is unreachable:

- Configure your DNS server to return multiple IP addresses for a host name. When you use this host name in the connection settings, the client attempts to connect to the first IP address from the DNS lookup. If the host at that IP address is unreachable, the client tries to connect to the second IP, and so on until it either manages to connect to a host or it runs out of IP addresses.

- Supply a list of backup hosts for the client driver to try if the primary host you specify in the connection parameters is unreachable.

For both methods, the process of failover is transparent to the client application (other than specifying the list of backup hosts, if you choose to use the list method of failover). If the primary host is unreachable, the client driver automatically tries to connect to other hosts.

Failover only applies to the initial establishment of the client connection. If the connection breaks, the driver does not automatically try to reconnect to another host in the database.

Choosing a Failover Method

You usually choose to use one of the two failover methods. However, they do work together. If your DNS server returns multiple IP addresses and you supply a list of backup hosts, the client first tries all of the IPs returned by the DNS server, then the hosts in the backup list.

Note: If a host name in the backup host list resolves to multiple IP addresses, the client does not try all of them. It just tries the first IP address in the list.

The DNS method of failover centralizes the configuration client failover. As you add new nodes to your Vertica Analytics Platform cluster, you can choose to add them to the failover list by editing the DNS server settings. All client systems that use the DNS server to connect to Vertica Analytics Platform automatically use connection failover without having to change any settings. However, this method does require administrative access to the DNS server that all clients use to connect to the Vertica Analytics Platform cluster. This may not be possible in your organization.

Using the backup server list is easier than editing the DNS server settings. However, it decentralizes the failover feature. You may need to update the application settings on each client system if you make changes to your Vertica Analytics Platform cluster.

Using DNS Failover

To use DNS failover, you need to change your DNS server's settings to map a single host name to multiple IP addresses of hosts in your Vertica Analytics Platform cluster. You then have all client applications use this host name to connect to Vertica Analytics Platform.

You can choose to have your DNS server return as many IP addresses for the host name as you want. In smaller clusters, you may choose to have it return the IP addresses of all of the hosts in your cluster. However, for larger clusters, you should consider choosing a subset of the hosts to return. Otherwise there can be a long delay as the client driver tries unsuccessfully to connect to each host in a database that is down.

Using the Backup Host List

To enable backup list-based connection failover, your client application has to specify at least one IP address or host name of a host in the `BackupServerNode` parameter. The host name or IP can optionally be followed by a colon and a port number. If not supplied, the driver defaults to the standard HP Vertica port number (5433). To list multiple hosts, separate them by a comma.

The following example demonstrates setting the `BackupServerNode` connection parameter to specify additional hosts for the connection attempt. The connection string intentionally has a non-existent node, so that the initial connection fails. The client driver has to resort to trying the backup hosts to establish a connection to HP Vertica.

```
import java.sql.*;
import java.util.Properties;

public class ConnectionFailoverExample {
    public static void main(String[] args) {
        // Assume using JDBC 4.0 driver on JVM 6+. No driver loading needed.
        Properties myProp = new Properties();
        myProp.put("user", "dbadmin");
        myProp.put("password", "vertica");
        // Set two backup hosts to be used if connecting to the first host
        // fails. All of these hosts will be tried in order until the connection
        // succeeds or all of the connections fail.
        myProp.put("BackupServerNode", "VerticaHost02,VerticaHost03");
        Connection conn;
        try {
            // The connection string is set to try to connect to a known
            // bnad host (in this case, a host that never existed).
            conn = DriverManager.getConnection(
                    "jdbc:vertica://BadVerticaHost:5433/vmart", myProp);
            System.out.println("Connected!");
            // Query system to table to see what node we are connected to.
            // Assume a single row in response set.
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(
                    "SELECT node_name FROM v_monitor.current_session;");
            rs.next();
            System.out.println("Connected to node " + rs.getString(1).trim());
            // Done with connection.
            conn.close();
        } catch (SQLException e) {
            // Catch-all for other exceptions
            e.printStackTrace();
        }
    }
}
```

When run, the example outputs output similar to the following on the system console:

```
Connected!
Connected to node v_vmart_node0002
```

Notice that the connection was made to the first node in the backup list (node 2).

Notes

- When native connection load balancing is enabled, the additional servers specified in the BackupServerNode connection parameter are only used for the initial connection to an HP Vertica host. If host redirects the client to another host in the database cluster to handle its connection request, the second connection does not use the backup node list. This is rarely an issue, since native connection load balancing is aware of which nodes are currently up in the database. See Enabling Native Connection Load Balancing in JDBC for more information.

JDBC Data Types

The JDBC driver transparently converts most HP Vertica data types to the appropriate Java data type. In a few cases, an HP Vertica data type cannot be directly translated to a Java data type; these exceptions are explained in this section.

HP Vertica Numeric Data Alias Conversion

The HP Vertica server supports data type aliases for integer, float and numeric types. The JDBC driver reports these as its basic data types (BIGINT, DOUBLE PRECISION, and NUMERIC), as follows:

HP Vertica Server Types and Aliases	HP Vertica JDBC Type
INTEGER INT INT8 BIGINT SMALLINT TINYINT	BIGINT
DOUBLE PRECISION FLOAT5 FLOAT8 REAL	DOUBLE PRECISION
DECIMAL NUMERIC NUMBER MONEY	NUMERIC

If a client application retrieves the values into smaller data types, HP Vertica JDBC driver does not check for overflows. The following example demonstrates the results of this overflow.

```
import java.sql.*;
import java.util.Properties;
public class JDBCDataTypes {
    public static void main(String[] args) {
       // If running under a Java 5 JVM, use you need to load the JDBC driver
       // using Class.forname here

        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                            "jdbc:vertica://VerticaHost:5433/VMart",
                             myProp);
            Statement statement = conn.createStatement();
            // Create a table that will hold a row of different types of
            // numeric data.
            statement.executeUpdate(
                       "DROP TABLE IF EXISTS test_all_types cascade");
            statement.executeUpdate("CREATE TABLE test_all_types ("
                            + "c0 INTEGER, c1 TINYINT, c2 DECIMAL, "
                            + "c3 MONEY, c4 DOUBLE PRECISION, c5 REAL)");
            // Add a row of values to it.
            statement.executeUpdate("INSERT INTO test_all_types VALUES("
                            + "111111111111, 444, 55555555555.5555, "
                            + "77777777.77,  88888888888888888.88, "
                            + "10101010.10101010101010)");
            // Query the new table to get the row back as a result set.
            ResultSet rs = statement
                            .executeQuery("SELECT * FROM test_all_types");
            // Get the metadata about the row, including its data type.
            ResultSetMetaData md = rs.getMetaData();
            // Loop should only run once...
            while (rs.next()) {
                // Print out the data type used to defined the column, followed
                // by the values retrieved using several different retrieval
                // methods.

               String[] vertTypes = new String[] {"INTEGER", "TINYINT",
                                "DECIMAL", "MONEY", "DOUBLE PRECISION", "REAL"};

               for (int x=1; x<7; x++) {
                       System.out.println("\n\nColumn " + x + " (" + vertTypes[x-1]
                                       + ")");
                       System.out.println("\tgetColumnType()\t\t"
                            + md.getColumnType(x));
                       System.out.println("\tgetColumnTypeName()\t"
                            + md.getColumnTypeName(x));
                    System.out.println("\tgetShort()\t\t"
                            + rs.getShort(x));
                    System.out.println("\tgetLong()\t\t" + rs.getLong(x));
                    System.out.println("\tgetInt()\t\t" + rs.getInt(x));
                    System.out.println("\tgetByte()\t\t" + rs.getByte(x));

               }
            }
            rs.close();
            statement.executeUpdate("drop table test_all_types cascade");
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

The above example prints the following on the console when run:

```
Column 1 (INTEGER)
       getColumnType()         -5
       getColumnTypeName()     BIGINT
       getShort()              455
       getLong()               111111111111
       getInt()                -558038585
       getByte()               -57
Column 2 (TINYINT)
       getColumnType()         -5
       getColumnTypeName()     BIGINT
       getShort()              444
       getLong()               444
       getInt()                444
       getByte()               -68
Column 3 (DECIMAL)
       getColumnType()         2
       getColumnTypeName()     NUMERIC
       getShort()              -1
       getLong()               55555555555
       getInt()                2147483647
       getByte()               -1
Column 4 (MONEY)
       getColumnType()         2
       getColumnTypeName()     NUMERIC
       getShort()              -13455
       getLong()               77777777
       getInt()                77777777
       getByte()               113
Column 5 (DOUBLE PRECISION)
       getColumnType()         8
       getColumnTypeName()     DOUBLE PRECISION
       getShort()              -1
       getLong()               88888888888888900
       getInt()                2147483647
       getByte()               -1
Column 6 (REAL)
       getColumnType()         8
       getColumnTypeName()     DOUBLE PRECISION
       getShort()              8466
       getLong()               10101010
       getInt()                10101010
       getByte()               18
```

Using Intervals with JDBC

The JDBC standard does not contain a data type for intervals (the duration between two points in time). To handle HP Vertica's INTERVAL data type, you must use JDBC's database-specific object type.

When reading an interval value from a result set, use the `ResultSet.getObject()` method to retrieve the value, and then cast it to one of the HP Vertica interval classes: `VerticaDayTimeInterval` (which represents all ten types of day/time intervals) or `VerticaYearMonthInterval` (which represents all three types of year/month intervals).

Note: The units interval style is not supported. Do not use the SET INTERVALSTYLE statement to change the interval style in your client applications.

Using Intervals in Batch Inserts

When inserting batches into tables that contain interval data, you must create instances of the `VerticaDayTimeInterval` or `VerticaYearMonthInterval` classes to hold the data you want to insert. You set values either when calling the class's constructor, or afterwards using setters. You then insert your interval values using the `PreparedStatement.setObject()` method. You can also use the `.setString()` method, passing it a string in "*`DD HH:MM:SS`*" or "*`YY-MM`*" format.

The following example demonstrates inserting data into a table containing a day/time interval and a year/month interval:

```
import java.sql.*;
import java.util.Properties;
// Need to import the Vertica JDBC classes to be able to instantiate
// the interval classes.
import com.vertica.jdbc.*;
public class IntervalDemo {
    public static void main(String[] args) {
        // If running under a Java 5 JVM, use you need to load the JDBC driver
        // using Class.forname here
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                    "jdbc:vertica://VerticaHost:5433/VMart", myProp);
            // Create table for interval values
            Statement stmt = conn.createStatement();
            stmt.execute("DROP TABLE IF EXISTS interval_demo");
            stmt.executeUpdate("CREATE TABLE interval_demo("
                    + "DayInt INTERVAL DAY TO SECOND, "
                    + "MonthInt INTERVAL YEAR TO MONTH)");
            // Insert data into interval columns using
            // VerticaDayTimeInterval and VerticaYearMonthInterval
            // classes.
            PreparedStatement pstmt = conn.prepareStatement(
                    "INSERT INTO interval_demo VALUES(?,?)");
            // Create instances of the Vertica classes that represent
            // intervals.
            VerticaDayTimeInterval dayInt = new VerticaDayTimeInterval(10, 0,
                    5, 40, 0, 0, false);
            VerticaYearMonthInterval monthInt = new VerticaYearMonthInterval(
                    10, 6, false);
            // These objects can also be manipulated using setters.
            dayInt.setHour(7);
            // Add the interval values to the batch
            ((VerticaPreparedStatement) pstmt).setObject(1, dayInt);
            ((VerticaPreparedStatement) pstmt).setObject(2, monthInt);
            pstmt.addBatch();
            // Set another row from strings.
            // Set day interval in "days HH:MM:SS" format
            pstmt.setString(1, "10 10:10:10");
            // Set year to month value in "MM-YY" format
            pstmt.setString(2, "12-09");
            pstmt.addBatch();
            // Execute the batch to insert the values.
            try {
                pstmt.executeBatch();
            } catch (SQLException e) {
                System.out.println("Error message: " + e.getMessage());
            }
```

Reading Interval Values

You read an interval value from a result set using the `ResultSet.getObject()` method, and cast the object to the appropriate HP Vertica object class: `VerticaDayTimeInterval` for day/time intervals or `VerticaYearMonthInterval` for year/month intervals. This is easy to do if you know that the column contains an interval, and you know what type of interval it is. If your application cannot assume the structure of the data in the result set it reads in, you can test whether a column contains a database-specific object type, and if so, determine whether the object belongs to either the `VerticaDayTimeInterval` or `VerticaYearMonthInterval` classes.

```
            // Retrieve the interval values inserted by previous demo.
            // Query the table to get the row back as a result set.
            ResultSet rs = stmt.executeQuery("SELECT * FROM interval_demo");
            // If you do not know the types of data contained in the result set,
            // you can read its metadata to determine the type, and use
            // additional information to determine the interval type.
            ResultSetMetaData md = rs.getMetaData();
            while (rs.next()) {
                for (int x = 1; x <= md.getColumnCount(); x++) {
                    // Get data type from metadata
                    int colDataType = md.getColumnType(x);
                    // You can get the type in a string:
                    System.out.println("Column " + x + " is a "
                            + md.getColumnTypeName(x));
                    // Normally, you'd have a switch statement here to
                    // handle all sorts of column types, but this example is
                    // simplified to just handle database-specific types
                    if (colDataType == Types.OTHER) {
                        // Column contains a database-specific type. Determine
                        // what type of interval it is. Assuming it is an
                        // interval...
                        Object columnVal = rs.getObject(x);
                        if (columnVal instanceof VerticaDayTimeInterval) {
                            // We know it is a date time interval
                            VerticaDayTimeInterval interval =
                                    (VerticaDayTimeInterval) columnVal;
                            // You can use the getters to access the interval's
                            // data
                            System.out.print("Column " + x + "'s value is ");
                            System.out.print(interval.getDay() + " Days ");
                            System.out.print(interval.getHour() + " Hours ");
                            System.out.println(interval.getMinute()
                                    + " Minutes");
                        } else if (columnVal instanceof VerticaYearMonthInterval) {
                            VerticaYearMonthInterval interval =
                                    (VerticaYearMonthInterval) columnVal;
                            System.out.print("Column " + x + "'s value is ");
                            System.out.print(interval.getYear() + " Years ");
                            System.out.println(interval.getMonth() + " Months");
                        } else {
                            System.out.println("Not an interval.");
                        }
                    }
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

The example prints the following to the console:

```
Column 1 is a INTERVAL DAY TO SECOND
Column 1's value is 10 Days 7 Hours 5 Minutes
Column 2 is a INTERVAL YEAR TO MONTH
Column 2's value is 10 Years 6 Months
Column 1 is a INTERVAL DAY TO SECOND
Column 1's value is 10 Days 10 Hours 10 Minutes
Column 2 is a INTERVAL YEAR TO MONTH
Column 2's value is 12 Years 9 Months
```

Another option is to use database metadata to find columns that contain intervals.

```
// Determine the interval data types by examining the database
// metadata.
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet dbMeta = dbmd.getColumns(null, null, "interval_demo", null);
int colcount = 0;
while (dbMeta.next()) {

    // Get the metadata type for a column.
    int javaType = dbMeta.getInt("DATA_TYPE");

    System.out.println("Column " + ++colcount + " Type name is " +
                    dbMeta.getString("TYPE_NAME"));

    if(javaType == Types.OTHER) {
      // The SQL_DATETIME_SUB column in the metadata tells you
      // Specifically which subtype of interval you have.
      // The VerticaDayTimeInterval.isDayTimeInterval()
      // methods tells you if that value is a day time.
      //
      int intervalType = dbMeta.getInt("SQL_DATETIME_SUB");
      if(VerticaDayTimeInterval.isDayTimeInterval(intervalType)) {
           // Now you know it is one of the 10 day/time interval types.
           // When you select this column you can cast to
           // VerticaDayTimeInterval.
           // You can get more specific by checking intervalType
           // against each of the 10 constants directly, but
           // they all are represented by the same object.
           System.out.println("column " + colcount + " is a " +
                           "VerticaDayTimeInterval intervalType = "
                          + intervalType);
      } else if(VerticaYearMonthInterval.isYearMonthInterval(
                      intervalType)) {
          //now you know it is one of the 3 year/month intervals,
          //and you can select the column and cast to
          // VerticaYearMonthInterval
          System.out.println("column " + colcount + " is a " +
                          "VerticaDayTimeInterval intervalType = "
                          + intervalType);
      } else {
          System.out.println("Not an interval type.");
      }
    }
}
```

Executing Queries Through JDBC

To run a query through JDBC:

1. Connect with the HP Vertica database. See Creating and Configuring a Connection.

2. Run the query.

The method you use to run the query depends on the type of query you want to run:

- a DDL query that does not return a result set.
- a DDL query that returns a result set.
- a DML query

Executing DDL (Data Definition Language) Queries

To run DDL queries, such as CREATE TABLE and COPY, use the `Statement.execute()` method. You get an instance of this class by calling the `createStatement` method of your connection object.

The following example creates an instance of the `Statement` class and uses it to execute a CREATE TABLE and a COPY query:

```
Statement stmt = conn.createStatement();
stmt.execute("CREATE TABLE address_book (Last_Name char(50) default ''," +
    "First_Name char(50),Email char(50),Phone_Number char(50))");
```

```
stmt.execute("COPY address_book FROM 'address.dat' DELIMITER ',' NULL 'null'");
```

Executing Queries That Return Result Sets

Use the `Statement` class's `executeQuery` method to execute queries that return a result set, such as SELECT. To get the data from the result set, use methods such as `getInt`, `getString`, and `getDouble` to access column values depending upon the data types of columns in the result set. Use `ResultSet.next` to advance to the next row of the data set.

```
ResultSet rs = null;
rs = stmt.executeQuery("SELECT First_Name, Last_Name FROM address_book");
int x = 1;
while(rs.next()){
    System.out.println(x + ". " + rs.getString(1).trim() + " "
                       + rs.getString(2).trim());
    x++;
}
```

Note: The HP Vertica JDBC driver does not support scrollable cursors. You can only read forwards through the result set.

Executing DML (Data Manipulation Language) Queries Using executeUpdate

Use the `executeUpdate` method for DML SQL queries that change data in the database, such as INSERT, UPDATE and DELETE which do not return a result set.

```
stmt.executeUpdate("INSERT INTO address_book " +
                   "VALUES ('Ben-Shachar', 'Tamar', 'tamarrow@example.com'," +
                   "'555-380-6466')");
stmt.executeUpdate("INSERT INTO address_book (First_Name, Email) " +
                   "VALUES ('Pete','pete@example.com')");
```

Note: The HP Vertica JDBC driver's `Statement` class supports executing multiple statements in the SQL string you pass to the `execute` method. The `PreparedStatement` class does not support using multiple statements in a single execution.

Loading Data Through JDBC

You can use any of the following methods to load data via the JDBC interface:

- Executing a SQL INSERT statement to insert a single row directly.
- Batch loading data using a prepared statement.
- Bulk loading data from files or streams using COPY.

When loading data into HP Vertica, you need to decide whether to write data to the **Write Optimized Store (WOS)** or the **Read Optimized Store (ROS)**. By default, most data loading methods insert data into the WOS until it fills up, then insert any additional data directly into ROS containers (called AUTO mode). This is the best method to use when frequently loading small amounts of data (often referred to as trickle-loading). When performing less frequent large data loads (any loads over 100MB of data at once), you should change this behavior to insert data directly into the ROS.

The following sections explain in detail how you load data using JDBC.

Using a Single Row Insert

The simplest way to insert data into a table is to use the SQL INSERT statement. You can use this statement by instantiating a member of the `Statement` class, and use its `executeUpdate()` method to run your SQL statement.

The following code fragment demonstrates how you can create a `Statement` object and use it to insert data into a table named address_book:

```
Statement stmt = conn.createStatement();
stmt.executeUpdate("INSERT INTO address_book " +
                        "VALUES ('Smith', 'John', 'jsmith@example.com', " +
                        "'555-123-4567')");
```

This method has a few drawbacks: you need convert your data to string and escape any special characters in your data. A better way to insert data is to use prepared statements. See Batch Inserts Using JDBC Prepared Statements.

Batch Inserts Using JDBC Prepared Statements

You can load batches of data into HP Vertica using prepared INSERT statements—server-side statements that you set up once, and then call repeatedly. You instantiate a member of the `PreparedStatement` class with a SQL statement that contains question mark placeholders for data. For example:

```
PreparedStatement pstmt = conn.prepareStatement(
                    "INSERT INTO customers(last, first, id) VALUES(?,?,?)");
```

You then set the parameters using data-type-specific methods on the `PreparedStatement` object, such as `setString()` and `setInt()`. Once your parameters are set, call the `addBatch()` method to add the row to the batch. When you have a complete batch of data ready, call the `executeBatch()` method to execute the insert batch.

Behind the scenes, the batch insert is converted into a COPY statement. When the connection's AutoCommit parameter is disabled, HP Vertica keeps the COPY statement open and uses it to load subsequent batches until the transaction is committed, the cursor is closed, or your application executes anything else (or executes any statement using another `Statement` or `PreparedStatement` object). Using a single COPY statement for multiple batch inserts makes loading data more efficient. If you are loading multiple batches, you should disable the AutoCommit property of the database to take advantage of this increased efficiency.

When performing batch inserts, experiment with various batch and row sizes to determine the settings that provide the best performance.

The following example demonstrates using a prepared statement to batch insert data.

```
import java.sql.*;
import java.util.Properties;
public class BatchInsertExample {
    public static void main(String[] args) {
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");

        //Set streamingBatchInsert to True to enable streaming mode for batch inserts.
        //myProp.put("streamingBatchInsert", "True");

        Connection conn;
        try {
            conn = DriverManager.getConnection(
                            "jdbc:vertica://VerticaHost:5433/ExampleDB",
                            myProp);
            // establish connection and make a table for the data.
            Statement stmt = conn.createStatement();


            // Set AutoCommit to false to allow Vertica to reuse the same
            // COPY statement
            conn.setAutoCommit(false);
            // Drop table and recreate.
            stmt.execute("DROP TABLE IF EXISTS customers CASCADE");
            stmt.execute("CREATE TABLE customers (CustID int, Last_Name"
                            + " char(50), First_Name char(50),Email char(50), "
                            + "Phone_Number char(12))");
            // Some dummy data to insert.
            String[] firstNames = new String[] { "Anna", "Bill", "Cindy",
                            "Don", "Eric" };
            String[] lastNames = new String[] { "Allen", "Brown", "Chu",
                            "Dodd", "Estavez" };
            String[] emails = new String[] { "aang@example.com",
                            "b.brown@example.com", "cindy@example.com",
                            "d.d@example.com", "e.estavez@example.com" };
            String[] phoneNumbers = new String[] { "123-456-7890",
                            "555-444-3333", "555-867-5309",
                            "555-555-1212", "781-555-0000" };
            // Create the prepared statement
            PreparedStatement pstmt = conn.prepareStatement(
                            "INSERT INTO customers (CustID, Last_Name, " +
                            "First_Name, Email, Phone_Number)" +
                            " VALUES(?,?,?,?,?)");
            // Add rows to a batch in a loop. Each iteration adds a
            // new row.
            for (int i = 0; i < firstNames.length; i++) {
                // Add each parameter to the row.
                pstmt.setInt(1, i + 1);
                pstmt.setString(2, lastNames[i]);
                pstmt.setString(3, firstNames[i]);
                pstmt.setString(4, emails[i]);
                pstmt.setString(5, phoneNumbers[i]);
                // Add row to the batch.
                pstmt.addBatch();
            }

            try {
                // Batch is ready, execute it to insert the data
                pstmt.executeBatch();
            } catch (SQLException e) {
                System.out.println("Error message: " + e.getMessage());
                return; // Exit if there was an error
            }

            // Commit the transaction to close the COPY command
            conn.commit();


            // Print the resulting table.
            ResultSet rs = null;
            rs = stmt.executeQuery("SELECT CustID, First_Name, "
                            + "Last_Name FROM customers ORDER BY CustID");
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " - "
                                + rs.getString(2).trim() + " "
                                + rs.getString(3).trim());
            }
            // Cleanup
            conn.close();
```

```
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

The result of running the example code is:

```
1 - Anna Allen
2 - Bill Brown
3 - Cindy Chu
4 - Don Dodd
5 - Eric Estavez
```

Streaming Batch Inserts

By default, HP Vertica performs batch inserts by caching each row and inserting the cache when the user calls the `executeBatch()` method. HP Vertica also supports streaming batch inserts. A streaming batch insert adds a row to the database each time the user calls `addBatch()`. Streaming batch inserts improve database performance by allowing parallel processing and reducing memory demands.

Note: Once you begin a streaming batch insert, you cannot make other JDBC calls that require client-server communication until you have executed the batch or closed or rolled back the connection.

To enable streaming batch inserts, set the `streamingBatchInsert` property to True. The preceding code sample includes a line enabling `streamingBatchInsert` mode. Remove the // comment marks to enable this line and activate streaming batch inserts.

The following table explains the various batch insert methods and how their behavior differs between default batch insert mode and streaming batch insert mode.

Method	**Default Batch Insert Behavior**	**Streaming Batch Insert Behavior**
`addBatch()`	Adds a row to the row cache.	Inserts a row into the database.
`executeBatch()`	Adds the contents of the row cache to the database in a single action.	Sends an end-of-batch message to the server and returns an array of integers indicating the success or failure of each `addBatch()` attempt.
`clearBatch()`	Clears the row cache without inserting any rows.	Not supported. Triggers an exception if used when streaming batch inserts are enabled.

Notes

- Using the `PreparedStatement.setFloat()` method can cause rounding errors. If precision is important, use the `.setDouble()` method instead.
- The `PreparedStatement` object caches the connection's AutoCommit property when the statement is prepared. Later changes to the AutoCommit property have no effect on the prepared statement.

Loading Batches Directly into ROS

When loading large batches of data (more than 100MB or so), you should load the data directly into ROS containers. Inserting directly into ROS is more efficient for large loads than AUTO mode, since it avoids overflowing the **WOS** and spilling the remainder of the batch to ROS. Otherwise, the Tuple Mover has to perform a **moveout** on the data in the WOS, while subsequent data is directly written into ROS containers causing your data to be segmented across storage containers.

When loading data using AUTO mode, HP Vertica inserts the data first into the **WOS**. If the WOS is full, then HP Vertica inserts the data directly into **ROS**. See the COPY statement for more details.

To directly load batches into ROS, set the directBatchInsert connection property to true. See Setting and Getting Connection Property Values for an explanation of how to set connection properties. When this property is set to true, all batch inserts bypass the WOS and load directly into a ROS container.

If all of batches being inserted using a connection should be inserted into the ROS, you should set the DirectBatchInsert connection property to true in the `Properties` object you use to create the connection:

```
Properties myProp = new Properties();
myProp.put("user", "ExampleUser");
myProp.put("password", "password123");
// Enable directBatchInsert for this connection
myProp.put("DirectBatchInsert", "true");
Connection conn;
try {
    conn = DriverManager.getConnection(
         "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
. . .
```

If you will be using the connection for inserting both large and small batches (or you do not know the size batches you will be inserting when you create the `Connection` object), you can set the DirectBatchInsert property after the connection has been established using the `VerticaConnection.setProperty` method:

```
((VerticaConnection)conn).setProperty("DirectBatchInsert", true);
```

See Setting and Getting Connection Property Values for a full example of setting DirectBatchInsert.

Error Handling During Batch Loads

When loading individual batches, you can find how many rows were accepted and what rows were rejected (see Identifying Accepted and Rejected Rows for details). If you have disabled the AutoCommit connection setting, other errors (such as disk space errors, for example) do not occur while inserting individual batches. This behavior is caused by having a single SQL COPY statement perform the loading of multiple consecutive batches (which makes the load process more efficient). It is only when the COPY statement closes that the batched data is committed and HP Vertica reports other types of errors.

Therefore, your bulk loading application should be prepared to check for errors when the COPY statement closes. You can trigger the COPY statement to close by:

- ending the batch load transaction by calling `Connection.commit()`
- closing the statement using `Statement.close()`
- setting the connection's AutoCommit property to true before inserting the last batch in the load

Note: The COPY statement also closes if you execute any non-insert statement or execute any statement using a different `Statement` or `PreparedStatement` object. Ending the COPY statement using either of these methods can lead to confusion and a harder-to-maintain application, since you would need to handle batch load errors in a non-batch load statement. You should explicitly end the COPY statement at the end of your batch load and handle any errors at that time.

Identifying Accepted and Rejected Rows (JDBC)

The return value of `PreparedStatement.executeBatch` is an integer array containing the success or failure status of inserting each row. A value 1 means the row was accepted and a value of -3 means that the row was rejected. In the case where an exception occurred during the batch execution, you can also get the array using `BatchUpdateException.getUpdateCounts()`.

The following example extends the example shown in Batch Inserts Using JDBC Prepared Statements to retrieve this array and display the results the batch load.

```
import java.sql.*;
import java.util.Arrays;
import java.util.Properties;
public class BatchInsertErrorHandlingExample {
    public static void main(String[] args) {
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;

        // establish connection and make a table for the data.
        try {
```

```
        conn = DriverManager.getConnection(
                        "jdbc:vertica://VerticaHost:5433/ExampleDB",
                        myProp);


        // Disable auto commit
        conn.setAutoCommit(false);

        // Create a statement
        Statement stmt = conn.createStatement();
        // Drop table and recreate.
        stmt.execute("DROP TABLE IF EXISTS customers CASCADE");
        stmt.execute("CREATE TABLE customers (CustID int, Last_Name"
                        + " char(50), First_Name char(50),Email char(50), "
                        + "Phone_Number char(12))");

        // Some dummy data to insert. The one row won't insert because
        // the phone number is too long for the phone column.
        String[] firstNames = new String[] { "Anna", "Bill", "Cindy",
                        "Don", "Eric" };
        String[] lastNames = new String[] { "Allen", "Brown", "Chu",
                        "Dodd", "Estavez" };
        String[] emails = new String[] { "aang@example.com",
                        "b.brown@example.com", "cindy@example.com",
                        "d.d@example.com", "e.estavez@example.com" };
        String[] phoneNumbers = new String[] { "123-456-789",
                        "555-444-3333", "555-867-53093453453",
                        "555-555-1212", "781-555-0000" };

        // Create the prepared statement
        PreparedStatement pstmt = conn.prepareStatement(
                        "INSERT INTO customers (CustID, Last_Name, " +
                        "First_Name, Email, Phone_Number)" +
                        " VALUES(?,?,?,?,?)");

        // Add rows to a batch in a loop. Each iteration adds a
        // new row.
        for (int i = 0; i < firstNames.length; i++) {
            // Add each parameter to the row.
            pstmt.setInt(1, i + 1);
            pstmt.setString(2, lastNames[i]);
            pstmt.setString(3, firstNames[i]);
            pstmt.setString(4, emails[i]);
            pstmt.setString(5, phoneNumbers[i]);
            // Add row to the batch.
            pstmt.addBatch();
        }

        // Integer array to hold the results of inserting
        // the batch. Will contain an entry for each row,
        // indicating success or failure.
        int[] batchResults = null;

        try {
            // Batch is ready, execute it to insert the data
            batchResults = pstmt.executeBatch();
        } catch (BatchUpdateException e) {
            // We expect an exception here, since one of the
```

```
                // inserted phone numbers is too wide for its column. All of the
                // rest of the rows will be inserted.
                System.out.println("Error message: " + e.getMessage());

                // Batch results isn't set due to exception, but you
                // can get it from the exception object.
                //
                // In your own code, you shouldn't assume the a batch
                // exception occurred, since exceptions can be thrown
                // by the server for a variety of reasons.
                batchResults = e.getUpdateCounts();
            }
            // You should also be prepared to catch SQLExceptions in your own
            // application code, to handle dropped connections and other general
            // problems.

            // Commit the transaction
            conn.commit();


            // Print the array holding the results of the batch insertions.
            System.out.println("Return value from inserting batch: "
                            + Arrays.toString(batchResults));
            // Print the resulting table.
            ResultSet rs = null;
            rs = stmt.executeQuery("SELECT CustID, First_Name, "
                            + "Last_Name FROM customers ORDER BY CustID");
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " - "
                                + rs.getString(2).trim() + " "
                                + rs.getString(3).trim());
            }

            // Cleanup
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Running the above example produces the following output on the console:

```
Error message: [Vertica][VJDBC](100172) One or more rows were rejected by the server.Retu
rn value from inserting batch: [1, 1, -3, 1, 1]
1 - Anna Allen
2 - Bill Brown
4 - Don Dodd
5 - Eric Estavez
```

Notice that the third row failed to insert because its phone number is too long for the `Phone_Number` column. All of the rest of the rows in the batch (including those after the error) were correctly inserted.

Note: It is more efficient for you to ensure that the data you are inserting is the correct data type and width for the table column you are inserting it into than to handle exceptions after the fact.

Rolling Back Batch Loads on the Server

Batch loads always insert all of their data, even if one or more rows is rejected. Only the rows that caused errors in a batch are not loaded. When the database connection's AutoCommit property is true, batches automatically commit their transactions when they complete, so once the batch finishes loading, the data is committed.

In some cases, you may want all of the data in a batch to be successfully inserted—none of the data should be committed if an error occurs. The best way to accomplish this is to turn off the database connection's AutoCommit property to prevent batches from automatically committing themselves. Then, if a batch encounters an error, you can roll back the transaction after catching the `BatchUpdateException` caused by the insertion error.

The following example demonstrates performing a rollback if any error occurs when loading a batch.

```
import java.sql.*;
import java.util.Arrays;
import java.util.Properties;
public class RollbackBatchOnError {
    public static void main(String[] args) {
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                            "jdbc:vertica://VerticaHost:5433/ExampleDB",
                            myProp);
            // Disable auto-commit. This will allow you to roll back a
            // a batch load if there is an error.
            conn.setAutoCommit(false);
            // establish connection and make a table for the data.
            Statement stmt = conn.createStatement();
            // Drop table and recreate.
            stmt.execute("DROP TABLE IF EXISTS customers CASCADE");
            stmt.execute("CREATE TABLE customers (CustID int, Last_Name"
                            + " char(50), First_Name char(50),Email char(50), "
                            + "Phone_Number char(12))");

            // Some dummy data to insert. The one row won't insert because
            // the phone number is too long for the phone column.
            String[] firstNames = new String[] { "Anna", "Bill", "Cindy",
                            "Don", "Eric" };
            String[] lastNames = new String[] { "Allen", "Brown", "Chu",
                            "Dodd", "Estavez" };
            String[] emails = new String[] { "aang@example.com",
                            "b.brown@example.com", "cindy@example.com",
                            "d.d@example.com", "e.estavez@example.com" };
            String[] phoneNumbers = new String[] { "123-456-789",
                            "555-444-3333", "555-867-53094535", "555-555-1212",
                            "781-555-0000" };
            // Create the prepared statement
            PreparedStatement pstmt = conn.prepareStatement(
                            "INSERT INTO customers (CustID, Last_Name, " +
                            "First_Name, Email, Phone_Number) "+
                            "VALUES(?,?,?,?,?)");
            // Add rows to a batch in a loop. Each iteration adds a
            // new row.
            for (int i = 0; i < firstNames.length; i++) {
                // Add each parameter to the row.
                pstmt.setInt(1, i + 1);
                pstmt.setString(2, lastNames[i]);
                pstmt.setString(3, firstNames[i]);
                pstmt.setString(4, emails[i]);
                pstmt.setString(5, phoneNumbers[i]);
                // Add row to the batch.
                pstmt.addBatch();
            }
            // Integer array to hold the results of inserting
            // the batch. Will contain an entry for each row,
            // indicating success or failure.
            int[] batchResults = null;
            try {
                // Batch is ready, execute it to insert the data
                batchResults = pstmt.executeBatch();
                // If we reach here, we inserted the batch without errors.
                // Commit it.
                System.out.println("Batch insert successful. Committing.");
                conn.commit();
            } catch (BatchUpdateException e) {
                    System.out.println("Error message: " + e.getMessage());
                    // Batch results isn't set due to exception, but you
                    // can get it from the exception object.
                    batchResults =  e.getUpdateCounts();
                    // Roll back the batch transaction.
                    System.out.println("Rolling back batch insertion");
                    conn.rollback();
            }
            catch  (SQLException e) {
                // General SQL errors, such as connection issues, throw
                // SQLExceptions. Your application should do something more
                // than just print a stack trace,
                e.printStackTrace();
            }
            System.out.println("Return value from inserting batch: "
                            + Arrays.toString(batchResults));
            System.out.println("Customers table contains:");


            // Print the resulting table.
            ResultSet rs = null;
            rs = stmt.executeQuery("SELECT CustID, First_Name, "
                            + "Last_Name FROM customers ORDER BY CustID");
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " - "
                                + rs.getString(2).trim() + " "
                                + rs.getString(3).trim());
            }
```

```
            // Cleanup
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Running the above example prints the following on the system console:

```
Error message: [Vertica][VJDBC](100172) One or more rows were rejected by the server.Roll
ing back batch insertion
Return value from inserting batch: [1, 1, -3, 1, 1]
Customers table contains:
```

The return values indicate whether each rows was successfully inserted. The value 1 means the row inserted without any issues, and a -3 indicates the row failed to insert.

The customers table is empty since the batch insert was rolled back due to the error caused by the third column.

Bulk Loading Using the COPY Statement

One of the fastest ways to load large amounts of data into HP Vertica at once (bulk loading) is to use the COPY statement. This statement loads data from a file stored on an HP Vertica host (or in a data stream) into a table in the database. You can pass the COPY statement parameters that define the format of the data in the file, how the data is to be transformed as it is loaded, how to handle errors, and how the data should be loaded. See the COPY documentation in the SQL Reference Manual for details.

One parameter that is particularly important is the DIRECT option, which tells COPY to load the data directly into **ROS** rather than going through the **WOS**. You should use this option when you are loading large files (over 100MB) into the database. Without this option, your load may fill the WOS and overflow into ROS, requiring the Tuple Mover to perform a **Moveout** on the data in the WOS. It is more efficient to directly load into ROS and avoid forcing a moveout.

Only a **superuser** can use the COPY statement to copy a file stored on a host, so you must connect to the database using a superuser account. If you want to have a non-superuser user bulk-load data, you can use COPY to load from a stream on the host (such as STDIN) rather than a file or stream data from the client (see Streaming Data Via JDBC). You can also perform a standard batch insert using a prepared statement, which uses the COPY statement in the background to load the data.

The following example demonstrates using the COPY statement through the JDBC to load a file name `customers.txt` into a new database table. This file must be stored on the database host to which your application connects (in this example, a host named VerticaHost). Since the `customers.txt` file used in the example is very large, this example uses the DIRECT option to bypass WOS and load directly into ROS.

```
import java.sql.*;
import java.util.Properties;
import com.vertica.jdbc.*;
public class COPYFromFile {
    public static void main(String[] args) {
        Properties myProp = new Properties();
        myProp.put("user", "ExampleAdmin"); // Must be superuser
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                            "jdbc:vertica://VerticaHost:5433/ExampleDB",myProp);
            // Disable AutoCommit
            conn.setAutoCommit(false);
            Statement stmt = conn.createStatement();
            // Create a table to hold data.
            stmt.execute("DROP TABLE IF EXISTS customers;");
            stmt.execute("CREATE TABLE IF NOT EXISTS customers (Last_Name char(50) "
                            + "NOT NULL, First_Name char(50),Email char(50), "
                            + "Phone_Number char(15))");

            // Use the COPY command to load data. Load directly into ROS, since
            // this load could be over 100MB. Use ENFORCELENGTH to reject
            // strings too wide for their columns.
            boolean result = stmt.execute("COPY customers FROM "
                            + " '/data/customers.txt' DIRECT ENFORCELENGTH");

            // Determine if execution returned a count value, or a full result
            // set.
            if (result) {
                System.out.println("Got result set");
            } else {
                // Count will usually return the count of rows inserted.
                System.out.println("Got count");
                int rowCount = stmt.getUpdateCount();
                System.out.println("Number of accepted rows = " + rowCount);
            }


            // Commit the data load
            conn.commit();
        } catch (SQLException e) {
            System.out.print("Error: ");
            System.out.println(e.toString());
        }
    }
}
```

The example prints the following out to the system console when run (assuming that the `customers.txt` file contained two million valid rows):

```
Number of accepted rows = 2000000
```

Streaming Data Via JDBC

There are two options to stream data from a file on the client to your HP Vertica database:

- Use the `VerticaCopyStream` class to stream data in an object-oriented manner
- Execute a COPY LOCAL SQL statement to stream the data

The topics in this section explain how to use these options.

Using VerticaCopyStream

The `VerticaCopyStream` class lets you stream data from the client system to an HP Vertica database. It lets you use the SQL COPY statement directly without having to copy the data to a host in the database cluster first. Using the COPY command to load data from the host requires superuser privileges to be able to access the host's filesystem. The COPY statement used to load data from a stream does not require superuser privileges so your client can connect using any user account that has INSERT privileges on the table that will receive the data.

To copy streams into the database:

1. Disable the database connections AutoCommit connection parameter.
2. Instantiate a `VerticaCopyStreamObject`, passing it at least the database connection objects and a string containing a COPY statement to load the data. This statement **must** copy data from the STDIN into your table. You can use whatever parameters are appropriate for your data load.

   Note: The `VerticaCopyStreamObject` constructor optionally takes a single `InputStream` object, or a `List` of `InputStream` objects. This option lets you pre-populate the list of streams to be copied into the database.

3. Call `VerticaCopyStreamObject.start()` to start the COPY statement and begin streaming the data in any streams you have already added to the `VerticaCopyStreamObject`.
4. Call `VerticaCopyStreamObject.addStream()` to add additional streams to the list of streams to send to the database. You can then call `VerticaCopyStreamObject.execute()` to stream them to the server.
5. Optionally, call `VerticaCopyStreamObject.getRejects()` to get a list of rejected rows from the last `.execute()` call. The list of rejects is reset by each call to `.execute()` or `.finish()`.

   Note: If you used either the REJECTED DATA or EXCEPTIONS options in the COPY statement you passed to `VerticaCopyStreamObject` the object in step 2, `.getRejects()` returns an empty list. You can only use one method of tracking the rejected rows at a time.

6. When you are finished adding streams, call `VerticaCopyStreamObject.finish()` to send any remaining streams to the database and close the COPY statement.
7. Call `Connection.commit()` to commit the loaded data.

Getting Rejected Rows

The `VerticaCopyStreamObject.getRejects()` method returns a List containing the row numbers of rows that were rejected after the previous `.execute()` method call. Each call to `.execute()` clears the list of rejected rows, so you need to call `.getRejects()` after each call to `.execute()`. Since `.start()` and `.finish()` also call `.execute()` to send any pending streams to the server, you should also call `.getRejects()` after these methods as well.

The following example demonstrates loading the content of five text files stored on the client system into a table.

```
import java.io.File;
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
import com.vertica.jdbc.VerticaConnection;
import com.vertica.jdbc.VerticaCopyStream;
public class CopyMultipleStreamsExample {
    public static void main(String[] args) {
        // Note: If running on Java 5, you need to call Class.forName
        // to manually load the JDBC driver.
        // Set up the properties of the connection
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser"); // Must be superuser
        myProp.put("password", "password123");
        // When performing bulk loads, you should always disable the
        // connection's AutoCommit property to ensure the loads happen as
        // efficiently as possible by reusing the same COPY command and
        // transaction.
        myProp.put("AutoCommit", "false");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                          "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
            Statement stmt = conn.createStatement();

            // Create a table to receive the data
            stmt.execute("DROP TABLE IF EXISTS customers");
            stmt.execute("CREATE TABLE customers (Last_Name char(50), "
                            + "First_Name char(50),Email char(50), "
                            + "Phone_Number char(15))");

            // Prepare the query to insert from a stream. This query must use
            // the COPY statement to load data from STDIN. Unlike copying from
            // a file on the host, you do not need superuser privileges to
            // copy a stream. All your user account needs is INSERT privileges
            // on the target table.
            String copyQuery = "COPY customers FROM STDIN "
                            + "DELIMITER '|' DIRECT ENFORCELENGTH";

            // Create an instance of the stream class. Pass in the
            // connection and the query string.
            VerticaCopyStream stream = new VerticaCopyStream(
                            (VerticaConnection) conn, copyQuery);

            // Keep running count of the number of rejects
            int totalRejects = 0;

            // start() starts the stream process, and opens the COPY command.
            stream.start();

            // If you added streams to VerticaCopyStream before calling start(),
            // You should check for rejects here (see below). The start() method
            // calls execute() to send any pre-queued streams to the server
            // once the COPY statement has been created.

            // Simple for loop to load 5 text files named customers-1.txt to
            // customers-5.txt
            for (int loadNum = 1; loadNum <= 5; loadNum++) {
                // Prepare the input file stream. Read from a local file.
                String filename = "C:\\Data\\customers-" + loadNum + ".txt";
                System.out.println("\n\nLoading file: " + filename);
                File inputFile = new File(filename);
                FileInputStream inputStream = new FileInputStream(inputFile);

                // Add stream to the VerticaCopyStream
                stream.addStream(inputStream);

                // call execute() to load the newly added stream. You could
                // add many streams and call execute once to load them all.
                // Which method you choose depends mainly on whether you want
                // the ability to check the number of rejections as the load
                // progresses so you can stop if the number of rejects gets too
                // high. Also, high numbers of InputStreams could create a
                // resource issue on your client system.
                stream.execute();

                // Show any rejects from this execution of the stream load
                // getRejects() returns a List containing the
                // row numbers of rejected rows.
                List<Long> rejects = stream.getRejects();

                // The size of the list gives you the number of rejected rows.
                int numRejects = rejects.size();
                totalRejects += numRejects;
                System.out.println("Number of rows rejected in load #"
                                + loadNum + ": " + numRejects);

                // List all of the rows that were rejected.
                Iterator<Long> rejit = rejects.iterator();
                long linecount = 0;
                while (rejit.hasNext()) {
                    System.out.print("Rejected row #" + ++linecount);
                    System.out.println(" is row " + rejit.next());
                }
            }
            // Finish closes the COPY command. It returns the number of
            // rows inserted.
            long results = stream.finish();
```

```
            System.out.println("Finish returned " + results);

            // If you added any streams that hadn't been executed(),
            // you should also check for rejects here, since finish()
            // calls execute() to

            // You can also get the number of rows inserted using
            // getRowCount().
            System.out.println("Number of rows accepted: "
                            + stream.getRowCount());
            System.out.println("Total number of rows rejected: " + totalRejects);

            // Commit the loaded data
            conn.commit();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Running the above example on some sample data results in the following output:

```
Loading file: C:\Data\customers-1.txtNumber of rows rejected in load #1: 3
Rejected row #1 is row 3
Rejected row #2 is row 7
Rejected row #3 is row 51
Loading file: C:\Data\customers-2.txt
Number of rows rejected in load #2: 5Rejected row #1 is row 4143
Rejected row #2 is row 6132
Rejected row #3 is row 9998
Rejected row #4 is row 10000
Rejected row #5 is row 10050
Loading file: C:\Data\customers-3.txt
Number of rows rejected in load #3: 9
Rejected row #1 is row 14142
Rejected row #2 is row 16131
Rejected row #3 is row 19999
Rejected row #4 is row 20001
Rejected row #5 is row 20005
Rejected row #6 is row 20049
Rejected row #7 is row 20056
Rejected row #8 is row 20144
Rejected row #9 is row 20236
Loading file: C:\Data\customers-4.txt
Number of rows rejected in load #4: 8
Rejected row #1 is row 23774
Rejected row #2 is row 24141
Rejected row #3 is row 25906
Rejected row #4 is row 26130
Rejected row #5 is row 27317
Rejected row #6 is row 28121
Rejected row #7 is row 29321
Rejected row #8 is row 29998
Loading file: C:\Data\customers-5.txt
Number of rows rejected in load #5: 1
```

```
Rejected row #1 is row 39997
Finish returned 39995
Number of rows accepted: 39995
Total number of rows rejected: 26
```

Note: The above example shows a simple load process that targets one node in the HP Vertica cluster. It is more efficient to simultaneously load multiple streams to multiple database nodes. Doing so greatly improves performance because it spreads the processing for the load across the cluster.

Using COPY LOCAL with JDBC

To use COPY LOCAL with JDBC, just execute a COPY LOCAL statement with the path to the source file on the client system. This method is simpler than using the `VerticaCopyStream` class. However, you may prefer using `VerticaCopyStream` if you have many files to copy to the database or if your data comes from a source other than a file (streamed over a network connection, for example).

The following example code demonstrates using COPY LOCAL to copy a file from the client to the database. It is the same as the code shown in Bulk Loading Using the COPY Statement, except for the use of the LOCAL option in the COPY statement, and the path to the data file is on the client system, rather than on the server.

Note: The exceptions/rejections files are created on the client machine when the exceptions and rejected data modifiers are specified on the copy local command. Specify a local path and filename for these modifiers when executing a COPY LOCAL query from the driver.

```
import java.sql.*;
import java.util.Properties;
public class COPYLocal {
    public static void main(String[] args) {
        // Note: If using Java 5, you must call Class.forName to load the
        // JDBC driver.
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser"); // Do not need to superuser
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                            "jdbc:vertica://VerticaHost:5433/ExampleDB",myProp);
            // Disable AutoCommit
            conn.setAutoCommit(false);
            Statement stmt = conn.createStatement();
            // Create a table to hold data.
            stmt.execute("DROP TABLE IF EXISTS customers;");
            stmt.execute("CREATE TABLE IF NOT EXISTS customers (Last_Name char(50) "
                            + "NOT NULL, First_Name char(50),Email char(50), "
                            + "Phone_Number char(15))");
            // Use the COPY command to load data. Load directly into ROS, since
            // this load could be over 100MB. Use ENFORCELENGTH to reject
            // strings too wide for their columns.
            boolean result = stmt.execute("COPY customers FROM LOCAL "
                            + " 'C:\\Data\\customers.txt' DIRECT ENFORCELENGTH");

            // Determine if execution returned a count value, or a full result
            // set.
            if (result) {
                System.out.println("Got result set");
            } else {
                // Count will usually return the count of rows inserted.
                System.out.println("Got count");
                int rowCount = stmt.getUpdateCount();
                System.out.println("Number of accepted rows = " + rowCount);
            }

            conn.close();
        } catch (SQLException e) {
            System.out.print("Error: ");
            System.out.println(e.toString());
        }
    }
}
```

The result of running this code appears below. In this case, the customers.txt file contains 10000 rows, seven of which get rejected because they contain data too wide to fit into their database columns.

```
Got countNumber of accepted rows = 9993
```

Handling Errors

When the HP Vertica JDBC driver encounters an error, it throws a `SQLException` or one of its subclasses. The specific subclass it throws depends on the type of error that has occurred. Most of the JDBC method calls can result in several different types of errors, in response to which the JDBC driver throws a specific `SQLException` subclass. Your client application can choose how to react to the error based on the specific exception that the JDBC driver threw.

Note: The specific `SQLException` subclasses were introduced in the JDBC 4.0 standard. If your client application runs in a Java 5 JVM, it will use the older JDBC 3.0-compliant driver which lacks these subclasses. In that case, all errors throw a `SQLException`.

The hierarchy of `SQLException` subclasses is arranged to help your client application determine what actions it can take in response to an error condition. For example:

- The JDBC driver throws `SQLTransientException` subclasses when the cause of the error may be a temporary condition, such as a timeout error (`SQLTimeoutException`) or a connection issue (`SQLTransientConnectionIssue`). Your client application can choose to retry the operation without making any sort of attempt to remedy the error, since it may not reoccur.

- The JDBC driver throws `SQLNonTransientException` subclasses when the client needs to take some action before it could retry the operation. For example, executing a statement with a SQL syntax error results in the JDBC driver throwing the a `SQLSyntaxErrorException` (a subclass of `SQLNonTransientException`). Often, your client application just has to report these errors back to the user and have him or her resolve them. For example, if the user supplied your application with a SQL statement that triggered a `SQLSyntaxErrorException`, it could prompt the user to fix the SQL error.

See Vertica Analytics Platform SQLState Mapping to Java Exception Classes for a list Java exceptions thrown by the JDBC driver.

Vertica Analytics Platform SQLState Mapping to Java Exception Classes

SQLSTATE Class or Value	Description	Java Exception Class
Class 00	Successful Completion	`SQLException`
Class 01	Warning	`SQLWarning`
Class 02	No Data	`SQLException`
Class 03	SQL Statement Not Yet Complete	`SQLException`

SQLSTATE Class or Value	Description	Java Exception Class
Class 08	Client Connection Exception	`SQLNonTransientConnectionException`
Class 09	Triggered Action Exception	`SQLException`
Class 0A	Feature Not Supported	`SQLFeatureNotSupportedException`
Class 0B	Invalid Transaction Initiation	`SQLException`
Class 0F	Locator Exception	`SQLException`
Class 0L	Invalid Grantor	`SQLException`
Class 0P	Invalid Role Specification	`SQLException`
Class 21	Cardinality Violation	`SQLException`
Class 22	Data Exception	`SQLDataException`
22V21	ERRCODE_INVALID_EPOCH	`SQLNonTransientException`
Class 23	Integrity Constraint Violation	`SQLIntegrityConstraintViolationException`
Class 24	Invalid Cursor State	`SQLException`
Class 25	Invalid Transaction State	`SQLTransactionRollbackException`
Class 26	Invalid SQL Statement Name	`SQLException`
Class 27	Triggered Data Change Violation	`SQLException`
Class 28	Invalid Authorization Specification	`SQLInvalidAuthorizationException`
Class 2B	Dependent Privilege Descriptors Still Exist	`SQLDataException`
Class 2D	Invalid Transaction Termination	`SQLException`
Class 2F	SQL Routine Exception	`SQLException`
Class 34	Invalid Cursor Name	`SQLException`

SQLSTATE Class or Value	Description	Java Exception Class
Class 38	External Routine Exception	`SQLException`
Class 39	External Routine Invocation Exception	`SQLException`
Class 3B	Savepoint Exception	`SQLException`
Class 3D	Invalid Catalog Name	`SQLException`
Class 3F	Invalid Schema Name	`SQLException`
Class 40	Transaction Rollback	`SQLTransactionRollbackException`
Class 42	Syntax Error or Access Rule Violation	`SQLSyntaxErrorException`
Class 44	WITH CHECK OPTION Violation	`SQLException`
Class 53	Insufficient Resources	`SQLTransientException`
53300	ERRCODE_TOO_ MANY_CONNECTIONS	`SQLNonTransientConnectionException`
Class 54	Program Limit Exceeded	`SQLNonTransientException`
Class 55	Object Not In Prerequisite State	`SQLNonTransientException`
55V03	ERRCODE_LOCK_ NOT_AVAILABLE	`SQLTransactionRollbackException`
Class 57	Operator Intervention	`SQLTransientException`
57V01	ERRCODE_ADMIN_ SHUTDOWN	`SQLNonTransientConnectionException`
57V02	ERRCODE_CRASH_ SHUTDOWN	`SQLNonTransientConnectionException`
57V03	ERRCODE_CANNOT_ CONNECT_NOW	`SQLNonTransientConnectionException`
Class 58	System Error	`SQLException`
Class V1	Vertica-specific multi-node errors class	`SQLException`

SQLSTATE Class or Value	Description	Java Exception Class
Class V2	Vertica-specific miscellaneous errors class	`SQLException`
V2000	ERRCODE_AUTH_ FAILED	`SQLInvalidAuthorizationException`
Class VC	Configuration File Error	`SQLNonTransientException`
Class VD	DB Designer errors	`SQLNonTransientException`
Class VP	User procedure errors	`SQLNonTransientException`
Class VX	Internal Error	`SQLException`

Error Handling Example

The following example code demonstrates catching two subclasses of `SQLException`. In this example, the program just prints out different error messages. Your own client application could respond to these errors in different ways.

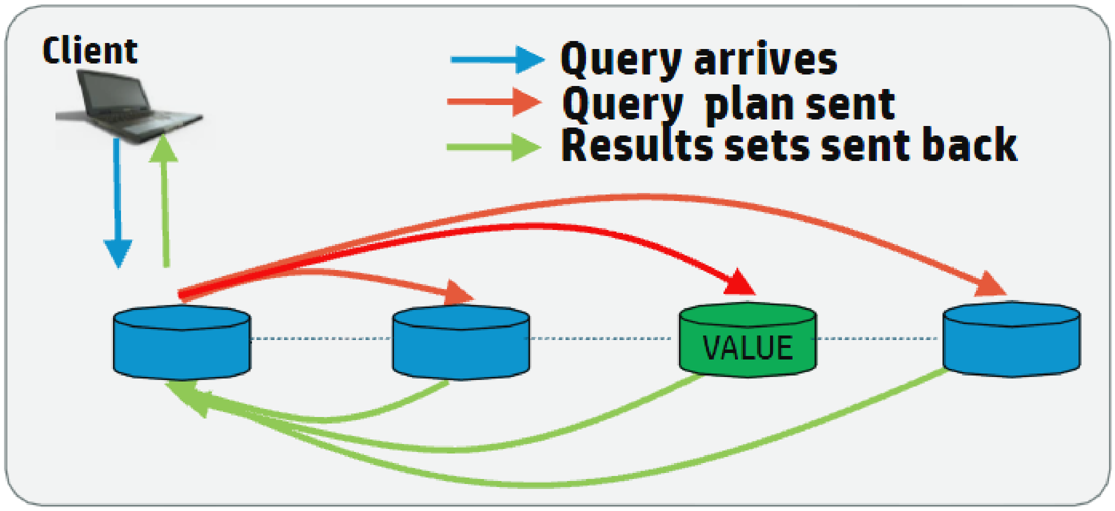
```
import java.sql.*;
import java.util.Properties;
// Demonstrate catching specific SQLException subclasses.
public class ExceptionClassExample {
    public static void main(String[] args)  {
        Properties myProp = new Properties();
        myProp.put("user", "myuser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                "jdbc:vertica://verticahost:5433/vmart", myProp);
            System.out.println("Connected!");

            // Array of statements that we want to run. Some contain errors.
            String[] statements = new String[] {
                    "DROP TABLE IF EXISTS t;",
                    "CREATE TABLE t (id INTEGER, name VARCHAR(50));",
                    "INSERT INTO t (id, name) VALUES (1, 'Alice');",
                    "INSERT INTO t (id, name) VALUES ('Bob', 2)",
                    "DRUP TABLE t CASCADE;",
                    "DROP TABLE t CASCADE;",
                    "CREATE TABLE nonExistentSchema.t (id INTEGER, name VARCHAR(50));"};

            Statement stmt = conn.createStatement();

            // Loop through the strings, executing each.
            for (String statement : statements ) {
                try {
                    System.out.println("Executing statement: '" + statement + "'");
                    stmt.execute(statement);
                    System.out.println("Success!");

                // Handle some specific types of SQL Exceptions

                // Syntax errors in statements are handled here.
                } catch (SQLSyntaxErrorException e) {
                    System.out.println("Statement '" + statement + "' has a syntax erro
r.");
                    System.out.println("    SQLSTATE = " + e.getSQLState());
                    System.out.println("    Error code: " + e.getErrorCode() +
                            " Error message: " + e.getMessage());

                // SQLDataException is thrown for various data-releted errors, such
                // as trying to put the wrong data type in a column.
                } catch (SQLDataException e) {
                    System.out.println("Statement '" + statement + "' caused a data erro
r.");
                    System.out.println("    SQLSTATE = " + e.getSQLState());
                    System.out.println("    Error code: " + e.getErrorCode() +
                            " Error message: " + e.getMessage());

                // Catch-all for other exceptions.
                } catch (SQLException e) {
                    System.out.println("Statement '" + statement + "' caused  a "
                            + e.getClass().getCanonicalName() + " exception.");
                    System.out.println("    SQLSTATE = " + e.getSQLState());
                    System.out.println("    Error code: " + e.getErrorCode() +
                            " Error message: " + e.getMessage());
                    e.getClass().getCanonicalName();
                    //e.printStackTrace();
                }
            }
            conn.close();
        // This catches exceptions caused by a bad connection.
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

About the JDBC Key/Value API

The JDBC Key/Value API allows you to quickly query data when a single or only a few rows are being returned and the data exists on a single node. This feature is ideal for high-volume "short" requests that return a small number of results. The common likely scenario for using the Key/Value API is for doing high-volume NoSQL-like lookups on data that is identified with unique primary key.

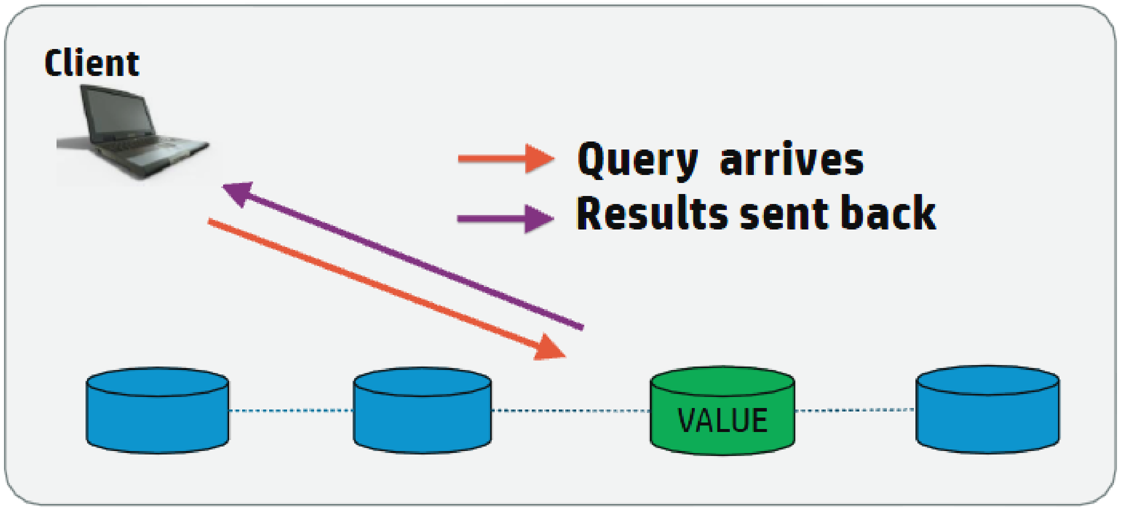
Typical analytic queries require dense computation on data across all nodes in the cluster and benefit from having all nodes involved in the planning and execution of the queries.

HP Vertica Typical Analytic Query



However, for high-volume queries that return a single or a few rows of data, it is more efficient to execute the query on the single node that contains the data.

HP Vertica Key/Value API Query



To effectively route a request to a single node, the client must determine the specific node on which the data resides. For the client to be able to determine the correct node, the table must segmented on one or more columns. For example, if you segment a table on a Primary Key (PK) column, then the client can determine on which node the data resides based on the Primary Key and directly connect to that node to quickly fulfill the request. However, it is not required that you segment on a PK column.

The Key/Value API does not use a traditional SQL interface. Instead, it uses a data structure that you build by defining predicates and predicate expressions and outputs and output expressions.

The data structure used for querying the table must provide a predicate for each segmented column defined in the projection for the table. You must provide, at a minimum, a predicate with a constant value for each segmented column. For example, an `id` with a value of 12234 if the table is segmented only on the `id` column. You can also specify additional predicates for the other, non-segmented, columns in the table. Predicates act like an SQL *WHERE* clause and multiple predicates/predicate expressions are joined together with a SQL AND modifier. Predicates must be defined with a constant value. Predicate expressions can be used to refine the query and can contain any arbitrary SQL expressions (such as less than, greater than, etc.) for any of the non-segmented columns in the table.

Java doc for all classes and methods in the Key/Value API is available in the HP Vertica JDBC Java doc.

Note: The JDBC Key/Value API is read-only and requires JDK 1.6 or greater.

See Also

- Creating Tables and Projections for use with the Key/Value API
- Creating a Connection for Key/Value Queries
- Defining the Query for Key/Value Lookups
- Key/Value Performance and Troubleshooting

Creating Tables and Projections for use with the Key/Value API

For Key/Value queries, the client needs to determine the appropriate node to get the data. The client does this by comparing all of the projections available for the table and determining the best projection to use to find the single node that contains data. You can simplify this comparison creating a projection so that the data is segmented and sorted on the key that you are using to query the data.

Note: Tables must be segmented by hash for Key/Value queries. See Hash-Segmentation-Clause. Other segmentation types are not supported.

Creating Tables for use with Key/Value

To create a table that can be used with the Key/Value API, segment (by hash) the table on a uniformly distributed column. Typically, you segment on a primary key. For faster lookups, sort the projection on the same columns on which you segmented. For example, to create a typical table that is well suited to key/value queries:

```
CREATE TABLE users (
       id INT NOT NULL PRIMARY KEY,
       username VARCHAR(32),
       email VARCHAR(64),
       business_unit VARCHAR(16))
ORDER BY id
```

```
SEGMENTED BY HASH(id)
ALL NODES;
```

This table is segmented based on the `id` column (and ordered by `id` to make lookups faster). To build a query for this table using the Key/Value API, you only need to provide a single predicate for the `id` column which returns a single row when queried.

However, if you were to add multiple columns to the segmentation clause, such as this table:

```
CREATE TABLE users (
       id INT NOT NULL PRIMARY KEY,
       username VARCHAR(32),
       email VARCHAR(64),
       business_unit VARCHAR(16))
ORDER BY id, business_unit
SEGMENTED BY HASH(id, business_unit)
ALL NODES;
```

Then you would need to provide two predicates when querying the *users2* table, since the segmentation clause uses both the *id* and the *business_unit* columns. However, if you know both *id* and *business_unit* when you perform the queries, then it is beneficial to segment on both columns, as it makes it easier for the client to determine that this projection is the best projection to use to determine the correct node.

Verifying Existing Projections for Tables

If you have existing tables that are already segmented by hash (for example, on an ID column), then you can determine what predicates are needed to query the table by using the `select get_table_projections('tableName')` command to view the projections associated with the table. The example table displays the following when `select get_table_projections('users')` is run:

```
Projection Name: [Segmented] [Seg Cols] [# of Buddies] [Buddy Projections] [Safe] [UptoDa
te] [Stats]
------------------------------------------------------------------------------------------
-----------
public.users_b1 [Segmented: Yes] [Seg Cols: "public.users.id"] [K: 1] [public.users_b0] [
Safe: Yes] [UptoDate: Yes] [Stats: RowCounts]
public.users_b0 [Segmented: Yes] [Seg Cols: "public.users.id"] [K: 1] [public.users_b1] [
Safe: Yes] [UptoDate: Yes] [Stats: RowCounts]
```

Note that for each projection, only the "`public.users.id`" column is specified,meaning you need to provide a predicate for this column when you build your query.

If the table was segmented on multiple columns, for example *id* and *business_unit*, then you would need to provide both columns as predicates to the query, to query this table with the Key/Value API.

Creating a Connection for Key/Value Queries

The JDBC Key/Value API provides the VerticaRoutableConnection interface to connect to a cluster and allow for Key/Value queries. This interface provides advanced routing capabilities beyond those of a normal VerticaConnection. The VerticaRoutableConnection provides access to the VGet class; see Defining the Query for Key/Value Lookups , which performs efficient key-based lookups.

You enable access to this class by setting the `EnableRoutableQueries` JDBC connection property to true.

The VerticaRoutableConnection maintains an internal pool of connections and a cache of table metadata that is shared by all VGet objects that are produced by this connection's `prepareGet()` method. It is also a fully-fledged JDBC connection on its own and supports all the functionality that a VerticaConnection supports. When this connection is closed, all pooled connections managed by this VerticaRoutableConnection and all child VGet objects are closed too. The connection pool and metadata is only used by child VGet operations.

Example:

You can create the connection using a JDBC data source:

```
com.vertica.jdbc.DataSource jdbcSettings = new com.vertica.jdbc.DataSource();
jdbcSettings.setDatabase("exampleDB");
jdbcSettings.setHost("doc1.verticacorp.com");
jdbcSettings.setUserID("dbadmin");
jdbcSettings.setPassword("password");
jdbcSettings.setEnableRoutableQueries(true);
jdbcSettings.setPort((short) 5433);

VerticaRoutableConnection conn;
```

You can also create the connection using a connection string and the `DriverManager.getConnection()` method:

```
String connectionString = "jdbc:vertica://doc.verticacorp.com:5433/exampleDB?user=dbadmin
&password=&EnableRoutableQueries=true";
VerticaRoutableConnection conn = (VerticaRoutableConnection) DriverManager.getConnection(
connectionString);
```

Both methods result in a `conn` connection object that is identical.

Note: Avoid opening many `VerticaRoutableConnection` connections because this connection maintains its own private pool of connections which are not shared with other connections. Instead, your application should use a single connection and issue multiple queries through that connection.

In addition to the `setEnableRoutableQueries` property that the Key/Value API adds to the HP Vertica JDBC connection class, the API also adds additional properties. The complete list is below.

Note: Preface these parameters with `set` when used as JDBC data source settings. For example, EnableRoutableQueries becomes `setEnableRoutableQueries` if used as a JDBC data source setting. If used in a connection string, then use `EnableRoutableQueries=true`.

- `EnableRoutableQueries`: Enables Key/Value lookup capability. Default is false.
- `FailOnMultiNodePlans`: If the plan requires more than one node, and FailOnMultiNodePlans is true, then the query fails. If it is set to false then a warning is generated and the query continues. However, latency is greatly increased as the Key/Value query must first determine the data is on multiple nodes,then a normal query is run using traditional (all node) execution and execution. Defaults to true. Note that this failure cannot occur on simple calls using only predicates and constant values.
- `MetadataCacheLifetime`: The time in seconds to keep projection metadata. The API caches metadata about the projection used for the query (such as projections). The cache is used on subsequent queries to reduce response time. The default is 300 seconds.
- `MaxPooledConnections`: Cluster-wide maximum number of connections to keep in the VerticaRoutableConnection's internal pool. Default 20.
- `MaxPooledConnectionsPerNode`: Per-node maximum number of connections to keep in the VerticaRoutableConnection's internal pool. Default 5.

Defining the Query for Key/Value Lookups

The VGet class is used to access table data directly from a single node. VGet directly queries HP Vertica nodes that have the data needed for the query, avoiding the distributed planning and execution costs associated with a normal HP Vertica execution.

VGet does not use SQL to define the query. Instead, it allows you to build a query using a data structure for which you define predicates, outputs, and sort and limit options.

You create a VGet by calling `prepareGet(`*`schema, table/proj`*`)` on a connection object. `prepareGet()` takes the name of the schema and the name of a table or projection as arguments.

Note: You can query projections directly instead of querying a table. When querying a table, the client driver attempts to find the best projection to use to return the data quickly. However, if there are many projections for the table, then HP Vertica may not always pick the most efficient projection.

VGet has the following methods:

- `addPredicate(`*`string, object`*`)` - adds a predicate column and a constant value to the query. You must include a predicate for each column on which the table is segmented. The predicate acts as the "WHERE" clause to the query. Multiple `addPredicate()` method calls are joined by AND modifiers. Note that the VGet retains this value after each call to execute. To remove it, use `ClearPredicates()`.

Note: The following data types cannot be used as predicates. Additionally, if a table is segmented on any columns with the following data types then the table cannot be queried using the Key/Value API:

- interval
- timetz
- timsestamptz
- numeric

- `addPredicateExpression(`*`string`*`)` - Accepts arbitrary SQL expressions that operate on the table's columns as input to the query. Predicate expressions and predicates arejoing by AND modifiers. You can use segmented columns in predicate expressions, but they must also be specified as a regular predicate with `addPredicate()`. Note that the VGet retains this value after each call to execute. To remove it, use `ClearPredicates()`.

  Note: The driver does not verify the syntax of the expression before it sends it to the server. If your expression is incorrect then the query fails.

- `addOutputColumn(`*`string`*`)` - Adds a column to be included in the output. By default the query runs as `SELECT *` and you do not need to define any output columns to return the data. If you add output columns then you must add all the columns you want returned. Note that the VGet retains this value after each call to execute. To remove it, use `ClearOutputs()`.

- `addOutputExpression(`*`string`*`)` - Accepts arbitrary SQL expressions that operate on the table's columns as output. Note that the VGet retains this value after each call to execute. To remove it, use `ClearOutputs()`.

  Note: The driver does not verify the syntax of the expression before it sends it to the server. If your expression is incorrect then the query fails.

- `addSortColumn(`*`string, SortOrder`*`)` - Adds a sort order to an output column. The output column can be either the one returned by the default query (SELECT *) or one of the columns defined in addOutputColumn or addOutputExpress. You can defined multiple sort columns.

- `setLimit(`*`int`*`)` - Sets a limit on the number of results returned. A limit of 0 is unlimited.

- `clearPredicates()` - Removes predicates that were added by `addPredicate()` and `addPredicateExpression()`.

- `clearOutputs()` - Removes outputs added by `addOutput()` and `addOutputExpression()`.

- `clearSortColumns()` - Removes sort columns previously added by `addSortColumn()`.

- `execute()` - Runs the query. Care must be taken to ensure that the predicate columns exist on the table and projection used by VGet, and that the expressions do not require multiple nodes to execute. If an expression is sufficiently complex as to require more than one node to execute, execute() throws a SQLException if the FailOnMultiNodePlans connection property is true.

- `close()` - Closes this VGet by releasing resources used by this VGet. It does not close the parent JDBC connection to HP Vertica.

- `getWarnings()` - Retrieves the first warning reported by calls on this VGet. Additional warning are chained and can be accessed with the JDBC `getNextWarning()` method.

You call the `execute()` method to run query. By default, the VGet fetches all the columns of all the rows that satisfy the logical AND of all the predicates passed via the `addPredicate()` method. To further customize the get operation use the `addOutputColumn()`, `addOutputExpression()`, `addPredicateExpression()`, `addSortColumn()` and `setLimit()` methods.

Note: VGet operations span multiple JDBC connections (and multiple HP Vertica sessions) and do not honor the parent connection's transaction semantics. If consistency is required across multiple executions, the parent VerticaRoutableConnection's consistent read API can be used to guarantee all operations occur at the same epoch.

VGet is thread safe, but all methods are synchronized, so threads that share a VGet instance are never run in parallel. For better parallelism, each thread should have its own VGet instance. Different VGet instances that operate on the same table share pooled connections and metadata in a manner that enables a high degree of parallelism.

Example

You can query the table defined in Creating Tables and Projections for use with the Key/Value API with the following example code. The table defines an id column that is segmented by hash.

```
import java.sql.*;
import com.vertica.jdbc.kv.*;

public class verticaKV2 {
       public static void main(String[] args) {
               com.vertica.jdbc.DataSource jdbcSettings
                       = new com.vertica.jdbc.DataSource();
               jdbcSettings.setDatabase("exampleDB");
               jdbcSettings.setHost("docg01.verticacorp.com");
               jdbcSettings.setUserID("dbadmin");
               jdbcSettings.setPassword("password");
               jdbcSettings.setEnableRoutableQueries(true);
               jdbcSettings.setPort((short) 5433);

               VerticaRoutableConnection conn;
               try {
                       conn = (VerticaRoutableConnection)
                               jdbcSettings.getConnection();
                   System.out.println("Connected.");
```

```
                        VGet get = conn.prepareGet("public", "users");
                        get.addPredicate("id", 5);
                        ResultSet rs = get.execute();
                        rs.next();
                        System.out.println("ID: " +
                                rs.getString("id"));
                        System.out.println("Username: "
                                + rs.getString("username"));
                        System.out.println("Email: "
                                + rs.getString("email"));
                        System.out.println("Closing Connection.");
                    conn.close();
                } catch (SQLException e) {
                        System.out.println("Error! Stacktrace:");
                    e.printStackTrace();
                }
        }
}
```

The output:

```
Connected.
ID: 5
Username: userE
Email: usere@example.com
Closing Connection.
```

Key/Value Performance and Troubleshooting

This topic details performance considerations and common issues you might encounter when using the Key/Value API.

Using Resource Pools with Key/Value Queries

Individual Key/Value queries are serviced quickly since they directly access a single node and return only one or a few rows of data. However, by default, HP Vertica resource pools use an AUTO setting for the `execution parallelism` parameter. When set to AUTO, the setting is determined by the number of CPU cores available and generally results in multi-threaded execution of queries in the resource pool. It is not efficient to create parallel threads on the server because VGet operations return data so quickly and VGet only uses a single thread to find a row. To prevent the server from opening unneeded processing threads, you should create a specific resource pool for Key/Value clients. Consider the following settings for the resource pool you use for Key/Value queries:

- Set execution parallelism to 1 to force single-threaded queries. This setting improves key/value performance.

- Use CPU affinity to limit the resource pool to a specific CPU or CPU set. The setting ensures that the key/value queries have resources available to them, but it also prevents key/value queries from significantly impacting performance on the system for other general queries.
- If you do not set a CPU affinity for the resource pool, consider setting the maximum concurrency value of the resource pool to a setting that ensures good performance for Key/Value queries, but does not negatively impact the performance of general queries.

Performance Considerations for Key/Value Connections

Because a VerticaRoutableConnection opens an internal pool of connections, it is important to configure `MaxPooledConnections` and `MaxPooledConnectionsPerNode` appropriately for your cluster size and the amount of simultaneous client connections. It is possible to impact normal database connections if you are overloading the cluster with `VerticaRoutableConnection`s.

The initial connection to the initiator node discovers all other nodes in the cluster. The internal-pool connections are not opened until a VGet query is sent. All VGets in a connection object use connections from the internal pool and are limited by the `MaxPooledConnections` settings. Connections remain open until they until the are close so a new connection can be opened elsewhere if the connection limit has been reached.

Troubleshooting Key/Value Queries

Key/Value query issues generally fall into two categories:

- Not providing enough predicates.
- Queries having to span multiple nodes.

Predicate Requirements

You must provide the same number of predicates that correspond to the columns of the table segmented by hash. To determine the segmented columns, run `select get_table_projections ('`*`tableName`*`')`. You must provide a predicate for each column displayed in the "Seg Cols" field.

Multi-node Failures

It is possible to define the correct number of predicates, but still have a failure because multiple nodes contain the data. This failure occurs because the projection's data is not segmented in such a way that the data being queried is contained on a single node. Enable logging for the connection and view the logs to verify the projection being used. If the client is not picking the correct projection, then try to query the projection directly by specifying the projection instead of the table in `conn.prepareGet('schema','table/projection')`.

Programming ADO.NET Applications

The HP Vertica driver for ADO.NET allows applications written in C# to read data from, update, and load data into HP Vertica databases. It provides a data adapter (HP Vertica Data Adapter) that facilitates reading data from a database into a data set, and then writing changed data from the data set back to the database. It also provides a data reader (HP VerticaDataReader) for reading data. The driver requires the .NET framework version 3.5+.

For more information about ADO.NET, see:

- Overview of ADO.NET
- ADO.NET .NET Framework Developer's Guide

Note: All of the examples provided in this section are in C#.

Updating ADO.NET Client Code From Previous Driver Versions

Starting in release 5.1.1, the HP Vertica client drivers have been updated to improve standards compliance, performance, and reliability. As a result, some HP Vertica-specific features and past incompatibilities have been eliminated. You must update any client code written for the prior versions of the ADO.NET driver to work with the version 5.1.1 driver and beyond.

Auto Commit Change

- All queries are now Auto Committed. The only exception is that queries run using a Transaction are not committed until the Commit(); method is called.

Performance Improvements

- Prepared INSERT statements now run significantly faster than in previous driver versions. For the best performance, prepared statements should be executed as part of a transaction.

Namespace Change

- The namespace has changed from vertica to Vertica.Data.VerticaClient

Connection Properties

- DSN is no longer a valid connection string keyword. You cannot connect to HP Vertica using ADO.net with a DSN.
- The RowBufferSize connection property has been renamed to ResultBufferSize.

- Getters on the VerticaConnection to get various connection string options (CacheDirectoryPooling, MinPoolSize, MaxPoolSize, SyncNotification, Timeout, Enlist, UseExtendedTypes, Password, Pooling, MinPoolSize, MaxPoolSize) have been removed.
- There is no longer a locale connection string keyword and you cannot set the locale through the connection string. To change the locale, run the query "set locale to..."
- The connection property to enable or disable auto commit has been removed. All queries outside of transactions are auto-committed.

Result Buffering

- The driver now buffers all results, and always uses streaming. Because of this, the following functionality has changed:
  - VerticaCommandBehavior enum has been removed. This enum extended the ADO.NET CommandBehavior enum to add support for buffering results. Results are now buffered in HP Vertica 5.1.x.
  - The VerticaCommand.ExecuteReader(CommandBehavior, bool) argument has been removed.
  - CacheDirectory or PreloadReader connection string options have been removed.

Logging Changes

- Log properties are no longer configured on the connection string. Log properties are now configured through the VerticaLogProperties class.

Data Type Changes

The following data types have changed:

Old Datatype Name	New Datatype Name
`verticaType.Integer`	`VerticaType.BigInt`
`verticaType.Bigint`	`VerticaType.BigInt`
`verticaType.Timestamp`	`VerticaType.DateTime`
`verticaType.Interval`	Changed to specific type of interval, for example: • VerticaType.IntervalDay • VerticaType.IntervalDayToHour • VerticaType.IntervalDayToMinute • etc.

Old Datatype Name	New Datatype Name
`verticaType.Real`	`VerticaType.Double`
`verticaType.Text`	`VerticaType.VarChar`
`verticaType.Smallint`	`VerticaType.BigInt`
`verticaType.Varbinary`	`VerticaType.VarBinary`

Multiple Commands Now Supported

Multiple commands in a single statement are now supported, provided that parameters are not used in any of the commands in the statement. The exception is COPY commands. You cannot issue multiple COPY commands in the same statement.

Setting the Locale for ADO.NET Sessions

- ADO.NET applications use a UTF-16 character set encoding and are responsible for converting any non-UTF-16 encoded data to UTF-16. The same cautions as for ODBC apply if this encoding is violated.
- The ADO.NET driver converts UTF-16 data to UTF-8 when passing to the HP Vertica server and converts data sent by HP Vertica server from UTF-8 to UTF-16
- ADO.NET applications should set the correct server session locale by executing the SET LOCALE TO command in order to get expected collation and string functions behavior on the server.
- If there is no default session locale at the database level, ADO.NET applications need to set the correct server session locale by executing the SET LOCALE TO command in order to get expected collation and string functions behavior on the server. See the SET LOCALE command in the SQL Reference Manual

Connecting to the Database

This section describes:

- Using SSL: Installing SSL Certificates on Windows
- Opening and Closing the Database Connection (ADO.NET)
- ADO.NET Connection Properties
- Configuring Log Properties

Using SSL: Installing Certificates on Windows

You can optionally secure communication between your ADO.NET application and HP Vertica using SSL. The HP Vertica ADO.NET driver uses the default Windows key store when looking for

SSL certificates. This is the same key store that Internet Explorer uses.

Before you can use SSL on the client side, you must implement SSL on the server. See Implementing SSL in the Administrator's Guide, perform those steps, then return to this topic to install the SSL certificate on Windows.

To use SSL for ADO.NET connections to HP Vertica:

- Import the server and client certificates into the Windows Key Store.
- If required by your certificates, import the public certificate of your Certifying Authority.

Import the Server and Client Certificates into the Windows Key store:

1. Copy the server.crt file you generated when you enabled SSL on the server to your Windows Machine.
2. Double-click the certificate.
3. Let Windows determine the key type, and click **Install**.

Import the Public Certificate of Your CA:

You must establish a chain of trust for the certificates. You may need to import the public certificate for your Certifying Authority (CA) (especially if it is a self-signed certificate).

1. using the same certificate as above, double-click the certificate.
2. Select **Place all certificates in the following store**.
3. Click **Browse,** select **Trusted Root Certification Authorities** and click **Next**.
4. Click **Install**.

Enable SSL in Your ADO.NET Applications

In your connection string, be sure to enable SSL by setting the `SSL` property in `VerticaConnectionStringBuilder` to true, for example:

```
//configure connection properties    VerticaConnectionStringBuilder builder = new Vertica
ConnectionStringBuilder();
    builder.Host = "192.168.17.10";
    builder.Database = "VMart";
    builder.User = "dbadmin";
    builder.SSL = true;
    //open the connection
    VerticaConnection _conn = new VerticaConnection(builder.ToString());
    _conn.Open();
```

Opening and Closing the Database Connection (ADO.NET)

Before you can access data in HP Vertica through ADO.NET, you must create a connection to the database using the VerticaConnection class which is an implementation of System.Data.DbConnection. The VerticaConnection class takes a single argument that contains the connection properties as a string. You can manually create a string of property keywords to use as the argument, or you can use the VerticaConnectionStringBuilder class to build a connection string for you.

This topic details the following:

- Manually building a connection string and connecting to HP Vertica
- Using VerticaConnectionStringBuilder to create the connection string and connecting to HP Vertica
- Closing the connection

To Manually Create a Connection string:

See ADO.NET Connection Properties for a list of available properties to use in your connection string. At a minimum, you need to specify the Host, Database, and User.

1. For each property, provide a value and append the properties and values one after the other, separated by a semicolon. Assign this string to a variable. For example:

```
String connectString = "DATABASE=VMart;HOST=node01;USER=dbadmin";
```

2. Build an HP Vertica connection object that specifies your connection string.

```
VerticaConnection _conn = new VerticaConnection(connectString)
```

3. Open the connection.

```
_conn.Open();
```

4. Create a command object and associate it with a connection. All VerticaCommand objects must be associated with a connection.

```
VerticaCommand command = _conn.CreateCommand();
```

To Use the VerticaConnectionStringBuilder Class to Create a Connection String and Open a connection:

1. Create a new object of the VerticaConnectionStringBuilder class.

   ```
   VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
   ```

2. Update your VerticaConnectionStringBuilder object with property values. See ADO.NET Connection Properties for a list of available properties to use in your connection string. At a minimum, you need to specify the Host, Database, and User.

   ```
   builder.Host = "node01";
   builder.Database = "VMart";
   builder.User = "dbadmin";
   ```

3. Build an HP Vertica connection object that specifies your connection VerticaConnectionStringBuilder object as a string.

   ```
   VerticaConnection _conn = new VerticaConnection(builder.ToString());
   ```

4. Open the connection.

   ```
   _conn.Open();
   ```

5. Create a command object and associate it with a connection. All VerticaCommand objects must be associated with a connection.

   ```
   VerticaCommand command = _conn.CreateCommand;
   ```

Note: If your database is not in compliance with your HP Vertica license, the call to `VerticaConnection.open()` returns a warning message to the console and the log. See Managing Your License Key in the Administrator's Guide for more information.

To Close the connection:

When you're finished with the database, close the connection. Failure to close the connection can deteriorate the performance and scalability of your application. It can also prevent other clients from obtaining locks.

```
_conn.Close();
```

Example Usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
           //Perform some operations
            _conn.Close();
        }
    }
}
```

ADO.NET Connection Properties

You use connection properties to configure the connection between your ADO.NET client application and your HP Vertica database. The properties provide the basic information about the connections, such as the server name and port number to use to connect to your database.

You can set a connection property in two ways:

- Include the property name and value as part of the the connection string you pass to a `VerticaConnection`.
- Set the properties in a `VerticaConnectionStringBuilder` object, then pass the object as a string to a `VerticaConnection`.

Property	Description	Default Value
`Database`	Name of the HP Vertica database to connect to. For example, if you installed the example VMart database, the database is "VMart"	string.Empty

Property	Description	Default Value
`User`	Name of the user to log into HP Vertica/	string.Empty
`Port`	Port on which HP Vertica is running.	5433
`Host`	The hostname or IP address of the server on which HP Vertica is running.	string.Empty
`Password`	The password associated with the user connecting to the server.	string.Empty
`ConnSettings`	SQL commands to run upon connection. Uses %3B for semicolons.	string.Empty
`IsolationLevel`	Sets the transaction isolation level for HP Vertica. See Transactions for a description of the different transaction levels. This value is either Serializable, ReadCommitted, or Unspecified. See Setting the Transaction Isolation Level for an example of setting the isolation level using this keyword. **Note:** By default, this value is set to IsolationLevel.Unspecified, which means the connection uses the server's default transaction isolation level. HP Vertica's default isolation level is IsolationLevel.ReadCommitted.	System.Data.IsolationLevel.Unspecified
`Label`	A string to identify the session on the server.	string
`DirectBatchInsert`	A boolean value, whether to Bulk insert to ROS (true) or WOS (false).	false
`ResultBufferSize`	The size of the buffer to use when streaming results.	8192
`ConnectionTimeout`	Number seconds to wait for a connection. A value of 0 means no timeout.	0

Property	Description	Default Value
`ReadOnly`	Boolean, if true, throw an exception on write attempts.	false
`Pooling`	A boolean value, whether to enable connection pooling. Connection pooling is useful for server applications because it allows the server to reuse connections. This saves resources and enhances the performance of executing commands on the database. It also reduces the amount of time a user must wait to establish a connection to the database	false
`MinPoolSize`	An integer that defines the minimum number of connections to pool. Cannot be greater than the number of connections that the server is configured to allow or an exception is thrown when that threshold is exceeded. The default number of connections allowed is 55.	1
`MaxPoolSize`	An integer that defines the maximum number of connections to pool. Cannot be greater than the number of connections that the server is configured to allow or an exception is thrown when that threshold is exceeded.	20

Property	Description	Default Value
`LoadBalanceTimeout`	The amount of time, in seconds, to timeout/remove pooled connections if the connections are unused. Set to 0 to disable this parameter and have no timeouts occur. If you are using a cluster environment to load-balance the work, then pool is restricted to the servers in the cluster when the pool was created. If additional servers are added to the cluster, and the pool is not removed, then the new servers will never be added to the connection pool unless LoadBalanceTimeout is set and exceeded or `VerticaConnection.ClearAllPools()` is manually called from an application. If you are using load balancing then set this to a value that takes into account when new servers are added to the cluster. However, do not set it so low that pools are frequently removed and rebuilt, as this defeats the purpose of using pooling in the first place.	0 (no timeout)
`SSL`	A boolean value, whether to use SSL for the connection.	false
`IntegratedSecurity`	Provides a Boolean value that, when set to true, uses the user's Windows credentials for authentication, instead of user/password in the connection string.	false
`KerberosServiceName`	Provides the service name portion of the HP Vertica Kerberos principal; for example: `vertica/host@EXAMPLE.COM`	vertica
`KerberosHostname`	Provides the instance or host name portion of the HP Vertica Kerberos principal; for example: `vertica/host@EXAMPLE.COM`	Value specified in the servername connection string property

Enabling Native Connection Load Balancing in ADO.NET

Native connection load balancing helps spread the overhead caused by client connections on the hosts in the HP Vertica database. Both the server and the client must enable native connection load balancing in order for it to have an effect. If both have enabled it, then when the client initially connects to a host in the database, the host picks a host to handle the client connection from a list of the currently up hosts in the database, and informs the client which host it has chosen. If the initially-contacted host did not choose itself to handle the connection, the client disconnects, then opens a second connection to the host selected by the first host. The connection process to this second host proceeds as usual—if SSL is enabled, then SSL negotiations begin, otherwise the client begins the authentication process. See About Native Connection Load Balancing in the Administrator's Guide for details.

To enable native load balancing on your client, set the `ConnectionLoadBalance` connection parameter to true either in the connection string or using the `ConnectionStringBuilder()`. The following example demonstrates connecting to the database several times with native connection load balancing enabled, and fetching the name of the node handling the connection from the V_MONITOR.CURRENT_SESSION system table.

```
using System;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder =
               new VerticaConnectionStringBuilder();
            builder.Host = "node01.example.com";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            // Enable native client load balancing in the client,
            // must also be enabled on the server!
            builder.ConnectionLoadBalance = true;
            // Connect 3 times to verify a new node is connected
            // for each connection.
            for (int i = 1; i <= 4; i++)
            {
                try
                {
                    VerticaConnection _conn =
                       new VerticaConnection(builder.ToString());
                    _conn.Open();
                    if (i == 1)
                    {
                        // On the first connection, check the server policy for load bala
nce
                        VerticaCommand sqlcom = _conn.CreateCommand();
                        sqlcom.CommandText =
               "SELECT LOAD_BALANCE_POLICY FROM V_CATALOG.DATABASES";
                        var returnValue = sqlcom.ExecuteScalar();
                        Console.WriteLine("Status of load balancy policy
                        on server: " + returnValue.ToString() + "\n");
                    }
                    VerticaCommand command = _conn.CreateCommand();
                    command.CommandText =
               "SELECT node_name FROM V_MONITOR.CURRENT_SESSION";
                    VerticaDataReader dr = command.ExecuteReader();
                    while (dr.Read())
                    {
                        Console.Write("Connect attempt #" + i + "... ");
                        Console.WriteLine("Connected to node " + dr[0]);
                    }
                    dr.Close();
                    _conn.Close();
                    Console.WriteLine("Disconnecting.\n");
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.Message);
                }
            }
        }
    }
}
```

Running the above example produces the following output:

```
Status of load balancy policy on server: roundrobin

Connect attempt #1... Connected to node v_vmart_node0001
Disconnecting.

Connect attempt #2... Connected to node v_vmart_node0002
Disconnecting.

Connect attempt #3... Connected to node v_vmart_node0003
Disconnecting.

Connect attempt #4... Connected to node v_vmart_node0001
Disconnecting.
```

ADO.NET Connection Failover

If a client application attempts to connect to a host in the Vertica Analytics Platform cluster that is down, the connection attempt fails when using the default connection configuration. This failure usually returns an error to the user. The user must either wait until the host recovers and retry the connection or manually edit the connection settings to choose another host.

Due to Vertica Analytics Platform's distributed architecture, you usually do not care which database host handles a client application's connection. You can use the client driver's connection failover feature to prevent the user from getting connection errors when the host specified in the connection settings is unreachable. It gives you two ways to let the client driver automatically attempt to connect to a different host if the one specified in the connection parameters is unreachable:

- Configure your DNS server to return multiple IP addresses for a host name. When you use this host name in the connection settings, the client attempts to connect to the first IP address from the DNS lookup. If the host at that IP address is unreachable, the client tries to connect to the second IP, and so on until it either manages to connect to a host or it runs out of IP addresses.
- Supply a list of backup hosts for the client driver to try if the primary host you specify in the connection parameters is unreachable.

For both methods, the process of failover is transparent to the client application (other than specifying the list of backup hosts, if you choose to use the list method of failover). If the primary host is unreachable, the client driver automatically tries to connect to other hosts.

Failover only applies to the initial establishment of the client connection. If the connection breaks, the driver does not automatically try to reconnect to another host in the database.

Choosing a Failover Method

You usually choose to use one of the two failover methods. However, they do work together. If your DNS server returns multiple IP addresses and you supply a list of backup hosts, the client first tries all of the IPs returned by the DNS server, then the hosts in the backup list.

Note: If a host name in the backup host list resolves to multiple IP addresses, the client does not try all of them. It just tries the first IP address in the list.

The DNS method of failover centralizes the configuration client failover. As you add new nodes to your Vertica Analytics Platform cluster, you can choose to add them to the failover list by editing the DNS server settings. All client systems that use the DNS server to connect to Vertica Analytics Platform automatically use connection failover without having to change any settings. However, this method does require administrative access to the DNS server that all clients use to connect to the Vertica Analytics Platform cluster. This may not be possible in your organization.

Using the backup server list is easier than editing the DNS server settings. However, it decentralizes the failover feature. You may need to update the application settings on each client system if you make changes to your Vertica Analytics Platform cluster.

Using DNS Failover

To use DNS failover, you need to change your DNS server's settings to map a single host name to multiple IP addresses of hosts in your Vertica Analytics Platform cluster. You then have all client applications use this host name to connect to Vertica Analytics Platform.

You can choose to have your DNS server return as many IP addresses for the host name as you want. In smaller clusters, you may choose to have it return the IP addresses of all of the hosts in your cluster. However, for larger clusters, you should consider choosing a subset of the hosts to return. Otherwise there can be a long delay as the client driver tries unsuccessfully to connect to each host in a database that is down.

Using the Backup Host List

To enable backup list-based connection failover, your client application has to specify at least one IP address or host name of a host in the `BackupServerNode` parameter. The host name or IP can optionally be followed by a colon and a port number. If not supplied, the driver defaults to the standard HP Vertica port number (5433). To list multiple hosts, separate them by a comma.

The following example demonstrates setting the `BackupServerNode` connection parameter to specify additional hosts for the connection attempt. The connection string intentionally has a non-existent node, so that the initial connection fails. The client driver has to resort to trying the backup hosts to establish a connection to HP Vertica.

```
using System;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder =
               new VerticaConnectionStringBuilder();
            builder.Host = "not.a.real.host:5433";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            builder.BackupServerNode =
               "another.broken.node:5433,docg02.verticacorp.com:5433";
            try
            {
                VerticaConnection _conn =
                       new VerticaConnection(builder.ToString());
                _conn.Open();
                VerticaCommand sqlcom = _conn.CreateCommand();
                sqlcom.CommandText = "SELECT node_name FROM current_session";
                var returnValue = sqlcom.ExecuteScalar();
                Console.WriteLine("Connected to node: " +
                       returnValue.ToString() + "\n");
                _conn.Close();
                Console.WriteLine("Disconnecting.\n");
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Notes

- When native connection load balancing is enabled, the additional servers specified in the BackupServerNode connection parameter are only used for the initial connection to an HP Vertica host. If host redirects the client to another host in the database cluster to handle its connection request, the second connection does not use the backup node list. This is rarely an issue, since native connection load balancing is aware of which nodes are currently up in the database. See Enabling Native Connection Load Balancing in ADO.NET.
- Connections to a host taken from the BackupServerNode list are not pooled for ADO.NET connections.

Configuring Log Properties (ADO.Net)

Log properties for ADO.Net are configured differently than they are other client drivers. On the other client drivers, log properties can be configured as one of the connection properties. The ADO.Net driver user the VerticaLogProperties class to configure the properties.

VerticaLogProperties

VerticaLogProperties is a static class that allows you to set and get the log settings for the ADO.net driver. You can control the log level, log path, and log namespace using this class.

The log is created when the first connection is opened. Once the connection is opened, you cannot change the log path. It must be set prior to opening the connection. You can change the log level and log namespace at any time.

Setting Log Properties

Setting the log properties is done using the three methods in the VerticaLogProperties class. The three methods are:

- SetLogPath(String path, bool persist)
- SetLogNamespace(String lognamespace, bool persist)
- SetLogLevel(VerticaLogLevel loglevel, bool persist)

Each of the methods requires a boolean persist argument. When set to true, the persist argument causes the setting to be written to the client's Windows Registry, where it is used for all subsequent connections. If set to false, then the log property only applies to the current session.

SetLogPath

The SetLogPath method takes as its arguments a string containing the path to the log file and the persist argument. If the path string contains only a directory path, then the log file is created with the name vdp-driver-MM-dd_HH.mm.ss.log (where MM-dd_HH.mm.ss is the date and time the log was created). If the path ends in a filename, such as log.txt or log.log, then the log is created with that filename.

If SetLogPath is called with an empty string for the path argument, then the client executable's current directory is used as the log path.

If SetLogPath is not called and no registry entry exists for the log path, and you have called any of the other VerticaLogProperties methods, then the client executable's current directory is used as the log path.

When the persist argument is set to true, the path specified is copied to the registry verbatim. If no filename was specified, then the filename is not saved to the registry.

Note: Note: The path must exist on the client system prior to calling this method. The method does not create directories.

Example Usage:

```
            //set the log path
            string path = "C:\\log";
            VerticaLogProperties.SetLogPath(path, false);
```

SetLogNamespace

The SetLogNamespace method takes as its arguments a string containing the namespace to log and the persist argument. The namespace string to log can be one of the following:

- Vertica
- Vertica.Data.VerticaClient
- Vertica.Data.Internal.IO
- Vertica.Data.Internal.DataEngine
- Vertica.Data.Internal.Core

Namespaces can be truncated to include multiple child namespaces. For example, you can specify "Vertica.Data.Internal" to log for all of the Vertica.Data.Internal namespaces.

If a log namespace is not set, and no value is stored in the registry, then the "Vertica" namespace is used for logging.

Example Usage:

```
            //set namespace to log
            string lognamespace = "Vertica.Data.VerticaClient";
            VerticaLogProperties.SetLogNamespace(lognamespace, false);
```

SetLogLevel

The SetLogLevel method takes as its arguments a VerticaLogLevel type and the persist argument. The VerticaLogLevel argument can be one of:

- VerticaLogLevel.None
- VerticaLogLevel.Fatal
- VerticaLogLevel.Error
- VerticaLogLevel.Warning
- VerticaLogLevel.Info
- VerticaLogLevel.Debug
- VerticaLogLevel.Trace

If a log level is not set, and no value is stored in the registry, then VerticaLogLevel.None is used.

Example Usage:

```
            //set log level
            VerticaLogLevel level = VerticaLogLevel.Debug;
            VerticaLogProperties.SetLogLevel(level, false);
```

Getting Log Properties

You can get the log property values using the getters included in the VerticaLogProperties class. The properties are:

- LogPath
- LogNamespace
- LogLevel

Example Usage:

```
            //get current log settings
            string logpath = VerticaLogProperties.LogPath;
            VerticaLogLevel loglevel = VerticaLogProperties.LogLevel;
            string logns = VerticaLogProperties.LogNamespace;
            Console.WriteLine("Current Log Settings:");
            Console.WriteLine("Log Path: " + logpath);
            Console.WriteLine("Log Level: " + loglevel);
            Console.WriteLine("Log Namespace: " + logns);
```

Setting and Getting Log Properties Example

This complete example shows how to set and get log properties:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
```

//configure connection properties

```
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
```

//get current log settings

```
            string logpath = VerticaLogProperties.LogPath;
            VerticaLogLevel loglevel = VerticaLogProperties.LogLevel;
            string logns = VerticaLogProperties.LogNamespace;
            Console.WriteLine("\nOld Log Settings:");
            Console.WriteLine("Log Path: " + logpath);
            Console.WriteLine("Log Level: " + loglevel);
            Console.WriteLine("Log Namespace: " + logns);
```

//set the log path

```
            string path = "C:\\log";
            VerticaLogProperties.SetLogPath(path, false);
```

//set log level

```
            VerticaLogLevel level = VerticaLogLevel.Debug;
            VerticaLogProperties.SetLogLevel(level, false);
```

//set namespace to log

```
            string lognamespace = "Vertica";
            VerticaLogProperties.SetLogNamespace(lognamespace, false);
```

//open the connection

```
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
```

//get new log settings

```
            logpath = VerticaLogProperties.LogPath;
            loglevel = VerticaLogProperties.LogLevel;
            logns = VerticaLogProperties.LogNamespace;
            Console.WriteLine("\nNew Log Settings:");
            Console.WriteLine("Log Path: " + logpath);
            Console.WriteLine("Log Level: " + loglevel);
            Console.WriteLine("Log Namespace: " + logns);
```

//close the connection

```
            _conn.Close();        }
    }
}
```

The example produces the following output:

```
Old Log Settings:
Log Path:
Log Level: None
Log Namespace:
New Log Settings:
Log Path: C:\log
Log Level: Debug
Log Namespace: Vertica
```

Querying the Database Using ADO.NET

This section describes how to create queries to do the following:

- Inserting data into the database
- Read data from the database
- Load data into the database

Note: The ExecuteNonQuery() method used to query the database returns an int32 with the number of rows affected by the query. The maximum size of an int32 type is a constant and is defined to be 2,147,483,547. If your query returns more results than the int32 max, then ADO.NET throws an exception because of the overflow of the int32 type. However the query is still processed by HP Vertica even when the reporting of the return value fails. This is a limitation in .NET, as ExecuteNonQuery() is part of the standard ADO.NET interface.

Inserting Data (ADO.NET)

Inserting data can done using the VerticaCommand class. VerticaCommand is an implementation of DbCommand. It allows you to create and send an SQL statement to the database. Use the CommandText method to assign an SQL statement to the command and then execute the SQL by calling the ExecuteNonQuery method. The ExecuteNonQuery method is used for executing statements that do not return result sets.

To Insert a Single Row of data:

1. Create a connection to the database.

2. Create a command object using the connection.

   ```
   VerticaCommand command = _conn.CreateCommand();
   ```

3. Insert data using an INSERT statement. The following is an example of a simple insert. Note that is does not contain a COMMIT statement because the HP Vertica ADO.NET driver operates in autocommit mode.

   ```
   command.CommandText =
        "INSERT into test values(2, 'username', 'email', 'password')";
   ```

4. Execute the query. The rowsAdded variable contains the number of rows added by the insert statement.

   ```
   Int32 rowsAdded = command.ExecuteNonQuery();
   ```

   The ExecuteNonQuery() method returns the number of rows affected by the command for UPDATE, INSERT, and DELETE statements. For all other types of statements it returns -1. If a rollback occurs then it is also set to -1.

Example Usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
               VerticaCommand command = _conn.CreateCommand();
               command.CommandText =
                 "INSERT into test values(2, 'username', 'email', 'password')";
               Int32 rowsAdded = command.ExecuteNonQuery();
               Console.WriteLine( rowsAdded + " rows added!");
            _conn.Close();
        }
    }
}
```

Using Parameters

You can use parameters to execute similar SQL statements repeatedly and efficiently.

Using Parameters

VerticaParameters are an extension of the System.Data.DbParameter base class in ADO.NET and are used to set parameters in commands sent to the server. Use Parameters in all queries (SELECT/INSERT/UPDATE/DELETE) for which the values in the WHERE clause are not static; that is for all queries that have a known set of columns, but whose filter criteria is set dynamically by an application or end user. Using parameters in this way greatly decreases the chances of an SQL injection issue that can occur when simply creating a SQL query from a number of variables.

Parameters require that a valid DbType, VerticaDbType, or System type be assigned to the parameter. See SQL Data Types and ADO.NET Data Types for a mapping of System, Vertica, and DbTypes.

To create a parameter placeholder, place either the at sign (@) or a colon (:) character in front of the parameter name in the actual query string. Do not insert any spaces between the placeholder indicator (@ or :) and the placeholder.

Note: The @ character is the preferred way to identify parameters. The colon (:) character is supported for backward compatibility.

For example, the following typical query uses the string 'MA' as a filter.

```
SELECT customer_name, customer_address, customer_city, customer_state
FROM customer_dimension WHERE customer_state = 'MA';
```

Instead, the query can be written to use a parameter. In the following example, the string MA is replaced by the parameter placeholder @STATE.

```
SELECT customer_name, customer_address, customer_city, customer_state
FROM customer_dimension WHERE customer_state = @STATE;
```

For example, the ADO.net code for the prior example would be written as:

```
VerticaCommand command = new VerticaCommand();
command.Text = “SELECT customer_name, customer_address, customer_city, customer_state
       FROM customer_dimension WHERE customer_state = @STATE”, _conn );
command.Parameters.Add(new VerticaParameter( “STATE”, VerticaType.VarChar));
command.Parameters["STATE"] = 'MA';
```

Note: Although the VerticaCommand class supports a Prepare() method, you do not need to call the Prepare() method for parameterized statements because HP Vertica automatically prepares the statement for you.

Creating and Rolling Back Transactions

Creating Transactions

Transactions in HP Vertica are atomic, consistent, isolated, and durable. When you connect to a database using the Vertica ADO.NET Driver, the connection is in autocommit mode and each individual query is committed upon execution. You can collect multiple statements into a single transaction and commit them at the same time by using a transaction. You can also choose to rollback a transaction before it is committed if your code determines that a transaction should not commit.

Transactions use the VerticaTransaction object, which is an implementation of DbTransaction. You must associate the transaction with the VerticaCommand object.

The following code uses an explicit transaction to insert one row each into to tables of the VMart schema.

To Create a Transaction in HP Vertica Using the ADO.NET driver:

1. Create a connection to the database.

2. Create a command object using the connection.

   ```
   VerticaCommand command = _conn.CreateCommand();
   ```

3. Start an explicit transaction, and associate the command with it.

   ```
   VerticaTransaction txn = _conn.BeginTransaction();
   ```

```
command.Connection = _conn;
command.Transaction = txn;
```

4. Execute the individual SQL statements to add rows.

```
command.CommandText =
     "insert into product_dimension values( ... )";
command.ExecuteNonQuery();
command.CommandText =
     "insert into store_orders_fact values( ... )";
```

5. Commit the transaction.

```
txn.Commit();
```

Rolling Back Transactions

If your code checks for errors, then you can catch the error and rollback the entire transaction.

```
VerticaTransaction txn = _conn.BeginTransaction();
VerticaCommand command = new
        VerticaCommand("insert into product_dimension values( 838929, 5, 'New item 5' )",
_conn);
// execute the insert
command.ExecuteNonQuery();
command.CommandText = "insert into product_dimension values( 838929, 6, 'New item 6' )";
// try insert and catch any errors
bool error = false;
try
{
    command.ExecuteNonQuery();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
    error = true;
}
if (error)
{
    txn.Rollback();
    Console.WriteLine("Errors. Rolling Back.");
}
else
{
    txn.Commit();
    Console.WriteLine("Queries Successful. Committing.");
}
```

Commit and Rollback Example

This example details how you can commit or rollback queries during a transaction.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
            bool error = false;
                VerticaCommand command = _conn.CreateCommand();
                VerticaCommand command2 = _conn.CreateCommand();
                VerticaTransaction txn = _conn.BeginTransaction();
                command.Connection = _conn;
                command.Transaction = txn;
                command.CommandText =
                "insert into test values(1, 'test', 'test', 'test' )";
                Console.WriteLine(command.CommandText);
                try
                {
                    command.ExecuteNonQuery();
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.Message);
                    error = true;
                }
                command.CommandText =
                "insert into test values(2, 'ear', 'eye', 'nose', 'extra' )";
                Console.WriteLine(command.CommandText);
                try
                {
                    command.ExecuteNonQuery();
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.Message);
                    error = true;
                }
                if (error)
                {
                    txn.Rollback();
                    Console.WriteLine("Errors. Rolling Back.");
                }
                else
                {
                    txn.Commit();
                    Console.WriteLine("Queries Successful. Committing.");
                }
            _conn.Close();
        }
    }
}
```

The example displays the following output on the console:

```
insert into test values(1, 'test', 'test', 'test' )
insert into test values(2, 'ear', 'eye', 'nose', 'extra' )
[42601]ERROR: INSERT has more expressions than target columns
Errors. Rolling Back.
```

See Also

- Setting the Transaction Isolation Level

Setting the Transaction Isolation Level

You can set the transaction isolation level on a per-connection and per-transaction basis. See **Transaction** for an overview of the transaction isolation levels supported in HP Vertica. To set the default transaction isolation level for a connection, use the `IsolationLevel` keyword in the VerticaConnectionStringBuilder string (see Connection String Keywords for details). To set the isolation level for an individual transaction, pass the isolation level to the `VerticaConnection.BeginTransaction()` method call to start the transaction.

To set the Isolation Level on a connection-basis:

1. Use the VerticaConnectionStringBuilder to build the connection string.

2. Provide a value for the IsolationLevel builder string. It can take one of two values: IsolationLevel.ReadCommited (default) or IsolationLevel.Serializeable. For example:

```
    VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();
    builder.Host = "192.168.1.100";
    builder.Database = "VMart";
    builder.User = "dbadmin";
    builder.IsolationLevel = System.Data.IsolationLevel.Serializeable
    VerticaConnection _conn1 = new VerticaConnection(builder.ToString());
    _conn1.Open();
```

To set the Isolation Level on a Transaction basis:

1. Set the IsolationLevel on the BeginTransaction method, for example

   ```
   VerticaTransaction txn = _conn.BeginTransaction(IsolationLevel.Serializable);
   ```

Example usage:

The following example demonstrates:

- getting the connection's transaction isolation level.
- setting the connection's isolation level using connection property.
- setting the transaction isolation level for a new transaction.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            VerticaConnection _conn1 = new VerticaConnection(builder.ToString());
             _conn1.Open();
            VerticaTransaction txn1 = _conn1.BeginTransaction();
            Console.WriteLine("\n Transaction 1 Transaction Isolation Level: " +
                        txn1.IsolationLevel.ToString());
            txn1.Rollback();
            VerticaTransaction txn2 = _conn1.BeginTransaction(IsolationLevel.Serializabl
e);
            Console.WriteLine("\n Transaction 2 Transaction Isolation Level: " +
                        txn2.IsolationLevel.ToString());
            txn2.Rollback();
            VerticaTransaction txn3 = _conn1.BeginTransaction(IsolationLevel.ReadCommitte
d);
            Console.WriteLine("\n Transaction 3 Transaction Isolation Level: " +
                        txn3.IsolationLevel.ToString());
            _conn1.Close();
```

```
        }
    }
}
```

When run, the example code prints the following to the system console:

```
Transaction 1 Transaction Isolation Level: ReadCommitted
Transaction 2 Transaction Isolation Level: Serializable
Transaction 3 Transaction Isolation Level: ReadCommitted
```

Reading Data (ADO.Net)

To read data from the database use VerticaDataReader, an implementation of DbDataReader. This implementation is useful for moving large volumes of data quickly off the server where it can be run through analytic applications.

Note: that the VerticaDataReader.HasRows property returns true if the result generated any rows. In versions of HP Vertica prior to 5.1, HasRows returned rows if there were more rows to be read.

Note: A VerticaCommand cannot execute anything else while it has an open VerticaDataReader associated with it. To execute something else, close the data reader or use a different VerticaCommand object.

To Read Data From the Database Using VerticaDataReader:

1. Create a connection to the database.

2. Create a command object using the connection.

   ```
   VerticaCommand command = _conn.CreateCommand();
   ```

3. Create a query. This query works with the example VMart database.

   ```
       command.CommandText =
       "SELECT fat_content, product_description " +
       "FROM (SELECT DISTINCT fat_content, product_description" +
       "      FROM product_dimension " +
       "      WHERE department_description " +        "      IN ('Dairy') " +
       "      ORDER BY fat_content) AS food " +
       "LIMIT 10;";
   ```

4. Execute the reader to return the results from the query. The following command calls the

ExecuteReader method of the VerticaCommand object to obtain the VerticaDataReader object.

```
VerticaDataReader dr = command.ExecuteReader();
```

5. Read the data. The data reader returns results in a sequential stream. Therefore, you must read data from tables row-by-row. The following example uses a while loop to accomplish this:

```
 Console.WriteLine("\n\n Fat Content\t  Product Description");
     Console.WriteLine("------------\t  -------------------");
     int rows = 0;
     while (dr.Read())
     {
        Console.WriteLine("     " + dr[0] + "    \t  " + dr[1]);
        ++rows;
     }
     Console.WriteLine("------------\n  (" + rows + " rows)\n");
```

6. When you're finished, close the data reader to free up resources.

```
     dr.Close();
```

Example Usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
               VerticaCommand command = _conn.CreateCommand();
               command.CommandText =
                 "SELECT fat_content, product_description " +
                 "FROM (SELECT DISTINCT fat_content, product_description" +
                 "      FROM product_dimension " +
                 "      WHERE department_description " +
                 "      IN ('Dairy') " +
                 "      ORDER BY fat_content) AS food " +
                 "LIMIT 10;";
               VerticaDataReader dr = command.ExecuteReader();

       Console.WriteLine("\n\n Fat Content\t  Product Description");
       Console.WriteLine("------------\t  -------------------");
       int rows = 0;
       while (dr.Read())
       {
                       Console.WriteLine("     " + dr[0] + "    \t  " + dr[1]);
                       ++rows;
       }
       Console.WriteLine("------------\n  (" + rows + " rows)\n");
               dr.Close();
            _conn.Close();
        }
    }
}
```

Loading Data Through ADO.Net

This section details the different ways that you can load data in HP Vertica using the ADO.NET client driver:

- Using the HP Vertica Data Adapter
- Using Batch Inserts and Prepared Statements
- Streaming Data Via ADO.NET

Using the HP Vertica Data Adapter

The HP Vertica data adapter (VerticaDataAdapter) enables a client to exchange data between a data set and an HP Vertica database. It is an implementation of DbDataAdapter. You can use VerticaDataAdapter to simply read data, or, for example, read data from a database into a data set, and then write changed data from the data set back to the database.

Batching Updates

When using the Update() method to update a dataset, you can optionally use the UpdateBatchSize () method prior to calling Update() to reduce the number of times the client communicates with the server to perform the update. The default value of UpdateBatchSize is 1. If you have multiple rows.Add() commands for a data set, then you can change the batch size to an optimal size to speed up the operations your client must perform to complete the update.

Reading Data From HP Vertica Using the Data adapter:

The following example details how to perform a select query on the VMart schema and load the result into a DataTable, then output the contents of the DataTable to the console.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
```

// Try/Catch any exceptions

```
            try
            {
                using (_conn)
                {
```

// Create the command

```
                    VerticaCommand command = _conn.CreateCommand();
                    command.CommandText = "select product_key, product_description " +
                        "from product_dimension where product_key < 10";
```

// Associate the command with the connection

```
                    command.Connection = _conn;
```

// Create the DataAdapter

```
                    VerticaDataAdapter adapter = new VerticaDataAdapter();
                    adapter.SelectCommand = command;
```

// Fill the DataTable

```
                    DataTable table = new DataTable();
                    adapter.Fill(table);
```

// Display each row and column value.

```
                    int i = 1;
                    foreach (DataRow row in table.Rows)
                    {
                        foreach (DataColumn column in table.Columns)
                        {
                            Console.Write(row[column] + "\t");
                        }
                        Console.WriteLine();
                        i++;
                    }
                    Console.WriteLine(i + " rows returned.");
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            _conn.Close();
        }
    }
}
```

Reading Data From HP Vertica into a Data set and Changing data:

The following example shows how to use a data adapter to read from and insert into a dimension table of the VMart schema.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using Vertica.Data.VerticaClient
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
```

// Try/Catch any exceptions

```
            try
            {
                using (_conn)
                {
```

//Create a data adapter object using the connection

```
                    VerticaDataAdapter da = new VerticaDataAdapter();
```

//Create a select statement that retrieves data from the table

```
                    da.SelectCommand = new
                        VerticaCommand("select * from product_dimension where product_key
< 10",
                        _conn);
```

//Set up the insert command for the data adapter, and bind variables for some of the columns

```
                    da.InsertCommand = new
                        VerticaCommand("insert into product_dimension values( :key, :vers
ion, :desc )",
                        _conn);
                    da.InsertCommand.Parameters.Add(new VerticaParameter("key", VerticaTy
pe.BigInt));
                    da.InsertCommand.Parameters.Add(new VerticaParameter("version", Verti
caType.BigInt));
                    da.InsertCommand.Parameters.Add(new VerticaParameter("desc", VerticaT
ype.VarChar));
                    da.InsertCommand.Parameters[0].SourceColumn = "product_key";
                    da.InsertCommand.Parameters[1].SourceColumn = "product_version";
                    da.InsertCommand.Parameters[2].SourceColumn = "product_description";
                    da.TableMappings.Add("product_key", "product_key");
                    da.TableMappings.Add("product_version", "product_version");
                    da.TableMappings.Add("product_description", "product_description");
```

//Create and fill a Data set for this dimension table, and get the resulting DataTable.

```
                    DataSet ds = new DataSet();
                    da.Fill(ds, 0, 0, "product_dimension");
                    DataTable dt = ds.Tables[0];
```

//Bind parameters and add two rows to the table.

```
                    DataRow dr = dt.NewRow();
                    dr["product_key"] = 838929;
                    dr["product_version"] = 5;
                    dr["product_description"] = "New item 5";
                    dt.Rows.Add(dr);
                    dr = dt.NewRow();
                    dr["product_key"] = 838929;
                    dr["product_version"] = 6;
                    dr["product_description"] = "New item 6";
                    dt.Rows.Add(dr);
```

//Extract the changes for the added rows.

```
                    DataSet ds2 = ds.GetChanges();
```

//Send the modifications to the server.

```
                    int updateCount = da.Update(ds2, "product_dimension");
```

//Merge the changes into the original Data set, and mark it up to date.

```
                    ds.Merge(ds2);
                    ds.AcceptChanges();
                    Console.WriteLine(updateCount + " updates made!");
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            _conn.Close();
        }
    }
}
```

Using Batch Inserts and Prepared Statements

You can load data in batches using a prepared statement with parameters. You can also use transactions to rollback the batch load if any errors are encountered.

If you are loading large batches of data (more than 100MB), then consider using a direct batch insert.

The following example details using data contained in arrays, parameters, and a transaction to batch load data.

The test table used in the example is created with the command: create table test (id INT, username VARCHAR(24), email VARCHAR(64), password VARCHAR(8));

Example Batch Insert Using Parameters and Transactions

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
```

```
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
```

// create arrays for column data

```
            int[] ids = {1, 2, 3, 4};
            string[] usernames = {"user1", "user2", "user3", "user4"};
            string[] emails = { "user1@example.com", "user2@example.com","user3@example.c
om","user4@example.com" };
            string[] passwords = { "pass1", "pass2", "pass3", "pass4" };
```

// create counters for accepted and rejected rows

```
            int rows = 0;
            int rejRows = 0;
            bool error = false;
```

// create the transaction

```
            VerticaTransaction txn = _conn.BeginTransaction();
```

// create the parametrized query and assign parameter types

```
            VerticaCommand command = _conn.CreateCommand();
            command.CommandText = "insert into TEST values (@id, @username, @email, @pass
word)";
            command.Parameters.Add(new VerticaParameter("id", VerticaType.BigInt));
            command.Parameters.Add(new VerticaParameter("username", VerticaType.VarCha
r));
            command.Parameters.Add(new VerticaParameter("email", VerticaType.VarChar));
            command.Parameters.Add(new VerticaParameter("password", VerticaType.VarCha
r));
```

// prepare the statement

```
            command.Prepare();
```

// loop through the column arrays and insert the data

```
            for (int i = 0; i < ids.Length; i++)            {
                command.Parameters["id"].Value = ids[i];
                command.Parameters["username"].Value = usernames[i];
                command.Parameters["email"].Value = emails[i];
                command.Parameters["password"].Value = passwords[i];
                try
                {
                    rows += command.ExecuteNonQuery();
                }
                catch (Exception e)
                {
                    Console.WriteLine("\nInsert failed - \n  " + e.Message + "\n");
                    ++rejRows;
                    error = true;
                }
            }
            if (error)
            {
```

//roll back if errors

```
                Console.WriteLine("Errors. Rolling Back Transaction.");
                Console.WriteLine(rejRows + " rows rejected.");
                txn.Rollback();
            }
            else
            {
```

//commit if no errors

```
                Console.WriteLine("No Errors. Committing Transaction.");
                txn.Commit();
                Console.WriteLine("Inserted " + rows + " rows. ");
            }
            _conn.Close();
        }
    }
}
```

Loading Batches Directly into ROS

When loading large batches of data (more than 100MB or so), you should load the data directly into ROS containers. Inserting directly into ROS is more efficient for large loads than AUTO mode, since it avoids overflowing the **WOS** and spilling the remainder of the batch to ROS. Otherwise, the Tuple Mover has to perform a **moveout** on the data in the WOS, while subsequent data is directly written into ROS containers. This results in the data from your batch being segmented across containers.

When loading data using AUTO mode, HP Vertica inserts the data first into the **WOS**. If the WOS is full, then HP Vertica inserts the data directly into **ROS**. See the COPY statement for more details.

To directly load batches into ROS, set the DirectBatchInsert connection property to true. See Opening and Closing the Database Connection for details on all of the connection properties. When the DirectBatchInsert property is set to true, all batch inserts bypass the WOS and load directly into a ROS container.

Example usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
               builder.DirectBatchInsert = true;
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
           //Perform some operations
            _conn.Close();
        }
    }
}
```

Streaming Data Via ADO.NET

There are two options to stream data from a file on the client to your HP Vertica database through ADO.NET:

- Use the `VerticaCopyStream` ADO.NET class to stream data in an object-oriented manner
- Execute a COPY SQL statement to stream the data

The topics in this section explain how to use these options.

Streaming From the Client Via VerticaCopyStream

The `VerticaCopyStream` class lets you stream data from the client system to an HP Vertica database. It lets you use the SQL COPY statement directly without having to copy the data to a host in the database cluster first by substituting one or more data stream(s) for STDIN.

Notes:

- Use Transactions and disable auto commit on the copy command for better performance.
- Disable auto commit using the copy command with the 'no commit' modifier. You must explicitly disable commits. Enabling transactions does not disable autocommit when using VerticaCopyStream.
- The copy command used with VerticaCopyStream uses copy syntax.
- VerticaCopyStream.rejects is zeroed every time execute is called. If you want to capture the number of rejects, assign the value of VerticaCopyStream.rejects to another variable before calling execute again.
- You can add multiple streams using multiple AddStream() calls.

Example usage:

The following example demonstrates using VerticaCopyStream to copy a file stream into HP Vertica.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.IO;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
```

//configure connection properties

```
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
            //open the connection
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
            try
            {
                using (_conn)
                {
```

//start a transaction

```
                    VerticaTransaction txn = _conn.BeginTransaction();
```

//create a table for this example

```
                    VerticaCommand command = new VerticaCommand("DROP TABLE IF EXISTS cop
y_table", _conn);                    command.ExecuteNonQuery();
                    command.CommandText = "CREATE TABLE copy_table (Last_Name char(50), "
                                    + "First_Name char(50),Email char(50), "
                                    + "Phone_Number char(15))";
                    command.ExecuteNonQuery();
```

//create a new filestream from the data file

```
                    string filename = "C:/customers.txt";                    Console.Writ
eLine("\n\nLoading File: " + filename);
                    FileStream inputfile = File.OpenRead(filename);
```

//define the copy command

```
                    string copy = "copy copy_table from stdin record terminator E'\n' del
imiter '|'"                        + " enforcelength "
                        + " no commit";
```

//create a new copy stream instance with the connection and copy statement

```
                    VerticaCopyStream vcs = new VerticaCopyStream(_conn, copy);
```

//start the VerticaCopyStream process

```
                    vcs.Start();
```

//add the file stream

```
                    vcs.AddStream(inputfile, false);
```

//execute the copy

```
                    vcs.Execute();
```

//finish stream and write out the list of inserted and rejected rows

```
                    long rowsInserted = vcs.Finish();                    IList<long> rows
Rejected = vcs.Rejects; // does not work when rejected or exceptions defined
                    Console.WriteLine("Number of Rows inserted: " + rowsInserted);
                    Console.WriteLine("Number of Rows rejected: " + rowsRejected.Count);
                    if (rowsRejected.Count > 0)
                    {
                        for (int i = 0; i < rowsRejected.Count; i++)
                        {
                            Console.WriteLine("Rejected row #{0} is row {1}", i, rowsReje
cted[i]);
                        }
                    }
```

//commit the changes

```
                    txn.Commit();                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
```

//close the connection

```
            _conn.Close();        }
    }
}
```

Using Copy with ADO.NET

To use COPY with ADO.NET, just execute a COPY statement and the path to the source file on the client system. This method is simpler than using the VerticaCopyStream class. However, you may prefer using VerticaCopyStream if you have many files to copy to the database or if your data comes from a source other than a local file (streamed over a network connection, for example).

The following example code demonstrates using COPY to copy a file from the client to the database. It is the same as the code shown in Bulk Loading Using the COPY Statement and the path to the data file is on the client system, rather than on the server.

To load data that is stored on a database node, use a VerticaCommand object to create a COPY command:

1. Create a connection to the database through the node on which the data file is stored.

2. Create a command object using the connection.

   ```
   VerticaCommand command = _conn.CreateCommand();
   ```

3. Copy data. The following is an example of using the COPY command to load data. It uses the LOCAL modifier to copy a file local to the client issuing the command.

   ```
   command.CommandText = "copy lcopy_table from '/home/dbadmin/customers.txt'"
     + " record terminator E'\n' delimiter '|'"
     + " enforcelength ";

   Int32 insertedRows = command.ExecuteNonQuery();
   Console.WriteLine(insertedRows + " inserted.");
   ```

Example Usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.IO;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
```

//configure connection properties

```
        VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder();

        builder.Host = "192.168.1.10";
        builder.Database = "VMart";
        builder.User = "dbadmin";
```

//open the connection

```
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
            try
            {
                using (_conn)
                {
```

//start a transaction

```
                    VerticaTransaction txn = _conn.BeginTransaction();
```

//create a table for this example

```
                    VerticaCommand command = new VerticaCommand("DROP TABLE IF EXISTS lco
py_table", _conn);
                    command.ExecuteNonQuery();
                    command.CommandText = "CREATE TABLE IF NOT EXISTS lcopy_table (Last_N
ame char(50), "
                                    + "First_Name char(50),Email char(50), "
                                    + "Phone_Number char(15))";
                    command.ExecuteNonQuery();
```

//define the copy command

```
                    command.CommandText = "copy lcopy_table from '/home/dbadmin/customers
.txt'"
                       + " record terminator E'\n' delimiter '|'"
                        + " enforcelength "
                               + " no commit";
```

//execute the copy

```
Int32 insertedRows = command.ExecuteNonQuery();
Console.WriteLine(insertedRows + " inserted.");
```

//commit the changes

```
                    txn.Commit();
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception: " + e.Message);
            }
```

//close the connection

```
            _conn.Close();
        }
    }
}
```

Handling Messages (ADO.NET)

You can capture info and warning messages that HP Vertica provides to the ADO.NET driver by using the InfoMessage event on the VerticaConnection delegate class. This class captures messages that are not severe enough to force an exception to be triggered, but might still provide information that can benefit your application.

To Use the VerticaInfoMessageEventHander class:

1. Create a method to handle the message sent from the even handler:

    ```
    static void conn_InfoMessage(object sender, VerticaInfoMessageEventArgs e)
    {
        Console.WriteLine(e.SqlState + ": " + e.Message);
    }
    ```

2. Create a connection and register a new VerticaInfoMessageHandler delegate for the

InfoMessage event:

```
        _conn.InfoMessage += new VerticaInfoMessageEventHandler(conn_InfoMessage);
```

3. Execute your queries. If a message is generated, then the event handle function is run.

4. You can unsubscribe from the event with the following command:

```
_conn.InfoMessage -= new VerticaInfoMessageEventHandler(conn_InfoMessage);
```

Example usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
```

// define message handler to deal with messages

```
        static void conn_InfoMessage(object sender, VerticaInfoMessageEventArgs e)
        {
            Console.WriteLine(e.SqlState + ": " + e.Message);
        }
        static void Main(string[] args)
        {
```

//configure connection properties

```
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
```

//open the connection

```
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
```

//create message handler instance by subscribing it to the InfoMessage event of the connection

```
            _conn.InfoMessage += new VerticaInfoMessageEventHandler(conn_InfoMessage);
```

//create and execute the command

```
            VerticaCommand cmd = _conn.CreateCommand();
            cmd.CommandText = "drop table if exists fakeTable";
            cmd.ExecuteNonQuery();
```

//close the connection

```
            _conn.Close();
        }
    }
}
```

This examples displays the following when run:

```
00000: Nothing was dropped
```

Getting Table Metadata (ADO.Net)

You can get the table metadata by using the GetSchema() method on a connection and loading the metadata into a DataTable:

```
DataTable table = _conn.GetSchema("Tables", new string[] { database_name, schema_name, ta
ble_name, table_type });
```

For example:

DataTable table = _conn.GetSchema("Tables", new string[] { null, null, null, "SYSTEM TABLE" });

database_name, *schema_name*, *table_name* can be set to *null,* be a specific name, or use a LIKE pattern.

table_type can be one of:

- "SYSTEM TABLE"
- "TABLE"
- "GLOBAL TEMPORARY"
- "LOCAL TEMPORARY"
- "VIEW"
- null

If *table_type* is set to null, then the metadata for all metadata tables is returned.

Example Usage:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using Vertica.Data.VerticaClient;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
```

//configure connection properties

```
            VerticaConnectionStringBuilder builder = new VerticaConnectionStringBuilder()
;
            builder.Host = "192.168.1.10";
            builder.Database = "VMart";
            builder.User = "dbadmin";
```

//open the connection

```
            VerticaConnection _conn = new VerticaConnection(builder.ToString());
            _conn.Open();
```

//create a new data table containing the schema

```
            //the last argument can be "SYSTEM TABLE", "TABLE", "GLOBAL TEMPORARY",
             // "LOCAL TEMPORARY", "VIEW", or null for all types
            DataTable table = _conn.GetSchema("Tables", new string[] { null, null, null,
"SYSTEM TABLE" });
```

//print out the schema

```
            foreach (DataRow row in table.Rows)            {
                foreach (DataColumn col in table.Columns)
                {
                    Console.WriteLine("{0} = {1}", col.ColumnName, row[col]);
                }
                Console.WriteLine("============================");
            }
```

//close the connection

```
            _conn.Close();
        }
    }
```

```
}
```

ADO.NET Data Types

his table details the mapping between HP Vertica data type's and .NET and ADO.NET data types.

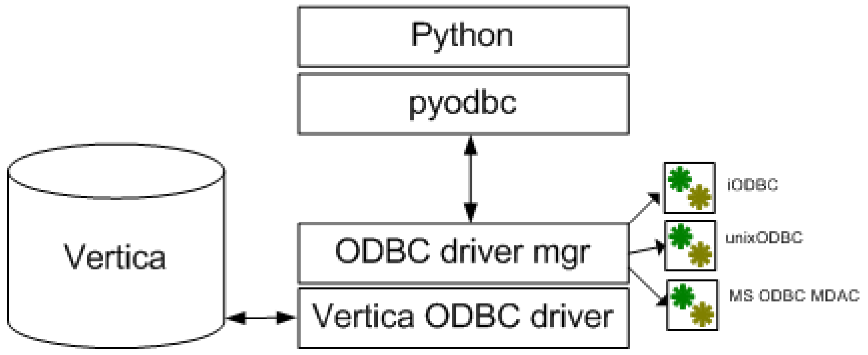
.NET Framework Type	ADO.NET DbType	VerticaType	HP Vertica Data Type	VerticaDataReader getter
Boolean	Boolean	Bit	Boolean	GetBoolean()
byte[]	Binary	Binary VarBinary LongVarBinary	Binary VarBinary LongVarBinary	GetBytes() **Note:** The limit for LongVarBinary is 32 Million bytes. If you attempt to insert more than the limit during a batch transfer for any one row, then they entire batch fails. Verify the size of the data before attempting to insert a LongVarBinary during a batch.

.NET Framework Type	ADO.NET DbType	VerticaType	HP Vertica Data Type	VerticaDataReader getter
Datetime	DateTime	Date Time TimeStamp	Date Time TimeStamp	GetDateTime() **Note:** The Time portion of the DateTime object for vertica dates is set to DateTime.MinValue. Previously, VerticaType.DateTime was used for all date/time types. VerticaType.DateTime still exists for backwards compatibility, but now there are more specific VerticaTypes for each type.
DateTimeOffset	DateTimeOffset	TimestampTZ TimeTZ	TimestampTZ TimeTZ	GetDateTimeOffset() **Note:** The Date portion of the DateTime is set to DateTime.MinValue
Decimal	Decimal	Numeric	Numeric	GetDecimal()
Double	Double	Double	Double Precision	GetDouble() **Note:** HP Vertica Double type uses a default precision of 53.
Int64	Int64	BigInt	Integer	GetInt64()

.NET Framework Type	ADO.NET DbType	VerticaType	HP Vertica Data Type	VerticaDataReader getter
TimeSpan	Object	13 Interval Types	13 Interval Types	GetInterval() **Note:** There are 13 VerticaType values for the 13 types of intervals. The specific VerticaType used determines the conversion rules that the driver applies. Year/Month intervals represented as 365/30 days
String	String	Varchar LongVarChar	Varchar LongVarChar	GetString()
String	StringFixedLengt	Char	Char	GetString()
Object	Object	N/A	N/A	GetValue()

Programming Python Client Applications

HP Vertica provides an ODBC driver so applications can connect to the HP Vertica database.



In order to use Python with HP Vertica, you must install the pyodbc module and an HP Vertica ODBC driver on the machine where Python is installed. See Python Prerequisites.

Python on Linux

Most Linux distributions come with Python preinstalled. If you want a more recent version, you can download and build it from the source code, though sometimes RPMs are also available. See the the Python Web site and click an individual release for details. See also Python documentation.

To determine the Python version on your Linux operating systems, type the following at a command prompt:

```
# python -V
```

The system returns the version; for example:

```
Python 2.5.2
```

Python on Windows

Python is not required to run natively on Windows operating systems, so it is not preinstalled. The ActiveState Web site distributes a free Windows installer for Python called ActivePython.

If you need installation instructions for Windows, see Using Python on Windows at python.org. Python on Windows at diveintopython.org provides installation instructions for both the ActivePython and python.org packages.

The Python Driver Module (pyodbc)

The native python driver is not supported.

Before you can connect to HP Vertica using Python, you need the pyodbc module, which communicates with iODBC/unixODBC driver on UNIX operating systems and the ODBC Driver Manager for Windows operating systems.

The pyodbc module is an open source , MIT-licensed Python module that implements the Python Database API Specification v2.0, letting you use ODBC to connect to almost any database from Windows, Linux, Mac OS/X, and other operating systems.

HP Vertica supports pyodbc version 2.1.6, which requires Python 2.4 or greater, up to 2.6. HP Vertica does not support Python version 3.x. See Python Prerequisites for additional details.

Download the source distribution from the pyodbc Web site, unpack it and build it. See the pyodbc wiki for instructions.

Note: Links to external Web sites could change between HP Vertica releases.

External Resources

- Python Database API Specification v2.0
- Python documentation

Configuring the ODBC Run-Time Environment on Linux

To configure the ODBC run-time environment on Linux:

1. Create the `odbc.ini` file if it does not already exist.
2. Add the ODBC driver directory to the LD_LIBRARY_PATH system environment variable:

```
export LD_LIBRARY_PATH=/path-to-vertica-odbc-driver:$LD_LIBRARY_PATH
```

Important: If you skip Step 2, the ODBC manager cannot find the driver in order to load it.

These steps are relevant only for unixODBC and iODBC. See their respective documentation for details on `odbc.ini`.

See Also

- unixODBC Web site
- iODBC Web site

Querying the Database Using Python

The example session below uses pyodbc with the HP Vertica ODBC driver to connect Python to the HP Vertica database.

> **Note:** SQLFetchScroll and SQLFetch functions cannot be mixed together in iODBC code. When using pyodbc with the iODBC driver manager, skip cannot be used with the fetchall, fetchone, and fetchmany functions.

1. Open a database connection, create a table called `TEST`:

```
cnxn = pyodbc.connect(connection_string, ansi=True) cursor = cnxn.cursor()
# create table
cursor.execute("CREATE TABLE TEST("
               "C_ID  INT,"
               "C_FP  FLOAT,"
               "C_VARCHAR VARCHAR(100),"
               "C_DATE DATE, C_TIME TIME,"
               "C_TS TIMESTAMP,"
               "C_BOOL BOOL)")
```

2. Insert records into table `TEST`:

```
cursor.execute("INSERT into testvalues(1,1.1,'abcdefg1234567890','1901-01-01','23:12:
34','1901-01-01
09:00:09','t')")
```

3. Insert records using bind values:

```
values = (2,2.28,'abcdefg1234567890','1901-01-01','23:12:34','1901-01-0109:00:09','
t')
cursor.execute("INSERT into test values(?,?,?,?,?,?,?)",
               values[0], values[1], values[2], values[3], values[4], values[5], valu
es[6])
```

4. Select data from the `TEST` table:

```
cursor.execute("SELECT * FROM TEST")rows = cursor.fetchall()
for row in rows:
    print row
```

5. The following is the example output:

```
(1L, 1.1000000000000001, 'abcdefg1234567890', datetime.date(1901, 1, 1),
datetime.time(23, 12, 34), datetime.datetime(1901, 1, 1, 9, 0, 9), '1') (2L, 2.279999
9999999998, 'abcdefg1234567890', datetime.date(1901, 1, 1), datetime.time(23, 12, 3
4), datetime.datetime(1901, 1, 1, 9, 0, 9), '1')
```

6. Drop the `TEST` table and its associated projections and close the database connection:

```
cursor.execute("DROP TABLE TEST CASCADE")cursor.close()
cnxn.close()
```

Notes

SQLPrimaryKeys returns the table name in the primary (`pk_name`) column for unnamed primary constraints. For example:

- Unnamed primary key:

```
CREATE TABLE schema.test(c INT PRIMARY KEY);SQLPrimaryKeys
"TABLE_CAT", "TABLE_SCHEM", "TABLE_NAME", "COLUMN_NAME", "KEY_SEQ", "PK_NAME" <Null>,
"SCHEMA", "TEST", "C", 1, "TEST"
```

- Named primary key:

```
CREATE TABLE schema.test(c INT CONSTRAINT pk_1 PRIMARY KEY);SQLPrimaryKeys
"TABLE_CAT", "TABLE_SCHEM", "TABLE_NAME", "COLUMN_NAME", "KEY_SEQ", "PK_NAME" <Null>,
"SCHEMA", "TEST", "C", 1, "PK_1"
```

HP recommends that you name your constraints.

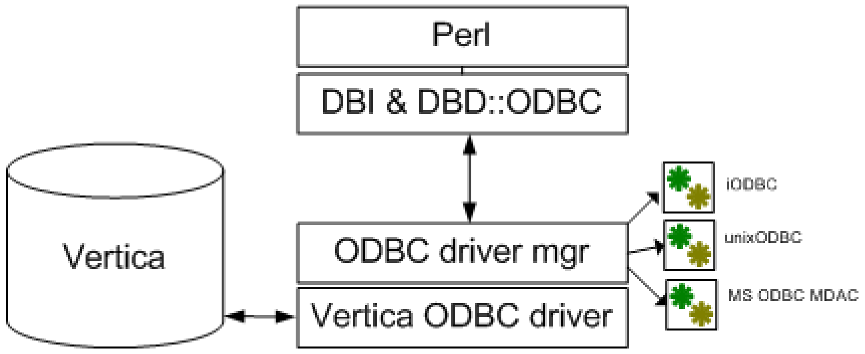
See Also

- Loading Data Through ODBC

Programming Perl Client Applications

The Perl programming language has a Database Interface module (DBI) that creates a standard interface for Perl scripts to interact with databases. The interface module relies on Database Driver modules (DBDs) to handle all of the database-specific communication tasks. The result is an interface that provides a consistent way for Perl scripts to interact with many different types of databases.

Your Perl script can interact with HP Vertica using the Perl DBI module along with the DBD::ODBC database driver to interface to HP Vertica's ODBC driver. See the CPAN pages for Perl's DBI and DBD::ODBC modules for detailed documentation.



The topics in this chapter explain how to:

- Configure Perl to access HP Vertica
- Connect to HP Vertica
- Query data stored in HP Vertica
- Insert data into HP Vertica

Perl Client Prerequisites

In order run a Perl client script that connects to HP Vertica, your client system must have:

- The HP Vertica ODBC drivers installed and configured. See Installing the HP Vertica Client Drivers for details.
- A Data Source Name (DSN) containing the connection parameters for your HP Vertica. See Creating an ODBC Data Source Name. (Optionally, your Perl script can connect to HP Vertica without using a DSN as described in Connecting From Perl Without a DSN).
- A supported version of Perl installed
- The DBI and DBD::ODBC Perl modules (see below)

Supported Perl Versions

HP Vertica supports Perl versions 5.8 and 5.10. Versions later than 5.10 may also work.

Perl on Linux

Most Linux distributions come with Perl preinstalled. See your Linux distribution's documentation for details of installing and configuring its Perl package is it is not already installed.

To determine the Perl version on your Linux operating systems, type the following at a command prompt:

```
# perl -v
```

The system returns the version; for example:

```
This is perl, v5.10.0 built for x86_64-linux-thread-multi
```

Perl on Windows

Perl is not installed by default on Windows platforms. There are several different Perl packages you can download and install on your Windows system:

- ActivePerl by Activestate is a commercially-supported version of Perl for Windows platforms.
- Strawberry Perl is an open-source port of Perl for Windows.

The Perl Driver Modules (DBI and DBD::ODBC)

Before you can connect to HP Vertica using Perl, your Perl installation needs to have the Perl Database Interface module (DBI) and the Database Driver for ODBC (DBD::ODBC). These modules communicate with iODBC/unixODBC driver on UNIX operating systems or the ODBC Driver Manager for Windows operating systems.

HP Vertica supports the following Perl modules:

- DBI version 1.609 (`DBI-1.609.tar.gz`)
- DBD::ODBC version 1.22 (`DBD-ODBC-1.22.tar.gz`)

Later versions of DBI and DBD::ODBC may also work.

DBI is installed by default with many Perl installations. You can test whether it is installed by executing the following command on the Linux or Windows command line:

```
# perl -e "use DBI;"
```

If the command exits without printing anything, then DBI is installed. If it prints an error, such as:

```
Can't locate DBI.pm in @INC (@INC contains: /usr/local/lib64/perl5/usr/local/share/perl5
 /usr/lib64/perl5/vendor_perl /usr/share/perl5/vendor_perl
/usr/lib64/perl5 /usr/share/perl5 .) at -e line 1.
BEGIN failed--compilation aborted at -e line 1.
```

then DBI is not installed.

Similarly, you can see if DBD::ODBC is installed by executing the command:

```
# perl -e "use DBD::ODBC;"
```

You can also run the following Perl script to determine if DBI and DBD::ODBC are installed. If they are, the script lists any available DSNs.

```
#!/usr/bin/perl
use strict;
# Attempt to load the DBI module in an eval using require. Prevents
# script from erroring out if DBI is not installed.
eval
{
    require DBI;
    DBI->import();
};
if ($@) {
    # The eval failed, so DBI must not be installed
    print "DBI module is not installed\n";
} else {
    # Eval was successful, so DBI is installed
    print "DBI Module is installed\n";
    # List the drivers that DBI knows about.
    my @drivers = DBI->available_drivers;
    print "Available Drivers: \n";
    foreach my $driver (@drivers) {
        print "\t$driver\n";
    }
    # See if DBD::ODBC is installed by searching driver array.
    if (grep {/ODBC/i} @drivers) {
        print "\nDBD::ODBC is installed.\n";
        # List the ODBC data sources (DSNs) defined on the system
        print "Defined ODBC Data Sources:\n";
        my @dsns = DBI->data_sources('ODBC');
        foreach my $dsn (@dsns) {
            print "\t$dsn\n";
        }
    } else {
        print "DBD::ODBC is not installed\n";
    }
}
```

The exact output of the above code will depend on the configuration of your system. The following is an example of running the code on a Windows computer:

```
DBI Module is installed
```

```
Available Drivers:
        ADO
        DBM
        ExampleP
        File
        Gofer
        ODBC
        Pg
        Proxy
        SQLite
        Sponge
        mysql
DBD::ODBC is installed.
Defined ODBC Data Sources:
        dbi:ODBC:dBASE Files
        dbi:ODBC:Excel Files
        dbi:ODBC:MS Access Database
        dbi:ODBC:VerticaDSN
```

Installing Missing Perl Modules

If Perl's DBI or DBD::ODBC modules are not installed on your client system, you must install them before your Perl scripts can connect to HP Vertica. How you install modules depends on your Perl configuration:

- For most Perl installations, you use the `cpan` command to install modules. If the `cpan` command alias isn't installed on your system, you can try to start CPAN by using the command:

  ```
  perl -MCPAN -e shell
  ```

- Some Linux distributions provide Perl modules as packages that can be installed with the system package manager (such as yum or apt). See your Linux distribution's documentation for details.

- On ActiveState Perl for Windows, you use the Perl Package Manager (PPM) program to install Perl modules. See the Activestate's PPM documentation for details.

Note: Installing Perl modules usually requires administrator or root privileges. If you do not have these permissions on your client system, you need to ask your system administrator to install these modules for you.

Connecting to HP Vertica Using Perl

You use the Perl DBI module's `connect` function to connect to HP Vertica. This function takes a required data source string argument and optional arguments for the username, password, and connection attributes.

The data source string must start with "dbi:ODBC:", which tells the DBI module to use the DBD::ODBC driver to connect to HP Vertica. The remainder of the string is interpreted by the

DBD::ODBC driver. It usually contains the name of a DSN that contains the connection information needed to connect to your HP Vertica database. For example, to tell the DBD::ODBC driver to use the DSN named VerticaDSN, you use the data source string:

```
"dbi:ODBC:VerticaDSN"
```

The username and password parameters are optional. However, if you do not supply them (or just the username for a passwordless account) and they are not set in the DSN, attempting to connect always fails.

The `connect` function returns a database handle if it connects to HP Vertica. If it does not, it returns `undef`. In that case, you can access the DBI module's error string property (`$DBI::errstr`) to get the error message.

Note: By default, the DBI module prints an error message to STDERR whenever it encounters an error. If you prefer to display your own error messages or handle errors in some other manner, you may want to disable these automatic messages by setting DBI's PrintError connection attribute to false. See Setting Perl DBI Connection Attributes for details. Otherwise, users may see two error messages: the one that DBI prints automatically, and the one that your script prints on its own.

The following example demonstrates connecting to HP Vertica using a DSN named VerticaDSN. The call to `connect` supplies a username and password. After connecting, it calls the database handle's `disconnect` function, which closes the connection.

```
#!/usr/bin/perl -w
use strict;
use DBI;
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123");
unless (defined $dbh) {
    # Conection failed.
    die "Failed to connect: $DBI::errstr";
}
print "Connected!\n";
$dbh->disconnect();
```

Setting ODBC Connection Parameters in Perl

To set ODBC connection parameters, replace the DSN name with a semicolon delimited list of parameter name and value pairs in the source data string. Use the DSN parameter to tell DBD::ODBC which DSN to use, then add in other the other ODBC parameters you want to set. For example, the following code connects using a DSN named VerticaDSN and sets the connection's locale to en_GB.

```
#!/usr/bin/perl -w
use strict;
use DBI;
# Instead of just using the DSN name, use name and value pairs.
```

```
my $dbh = DBI->connect("dbi:ODBC:DSN=VerticaDSN;Locale=en_GB","ExampleUser","password12
3");
unless (defined $dbh) {
    # Conection failed.
    die "Failed to connect: $DBI::errstr";
}
print "Connected!\n";
$dbh->disconnect();
```

See DSN Parameters for a list of the connection parameters you can set in the source data string.

Setting Perl DBI Connection Attributes

The Perl DBI module has attributes that you can use to control the behavior of its database connection. These attributes are similar to the ODBC connection parameters (in several cases, they duplicate each other's functionality). The DBI connection attributes are a cross-platform way of controlling the behavior of the database connection.

You can set the DBI connection attributes when establishing a connection by passing the DBI `connect` function a hash containing attribute and value pairs. For example, to set the DBI connection attribute AutoCommit to false, you would use:

```
# Create a hash that holds attributes for the connection
my $attr = {AutoCommit => 0};
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
    $attr);
```

See the DBI documentation's Database Handle Attributes section for a full description of the attributes you can set on the database connection.

After your script has connected, it can access and modify the connection attributes through the database handle by using it as a hash reference. For example:

```
print "The AutoCommit attribute is: " . $dbh->{AutoCommit} . "\n";
```

The following example demonstrates setting two connection attributes:

- RaiseError controls whether the DBI driver generates a Perl error if it encounters a database error. Usually, you set this to true (1) if you want your Perl script to exit if there is a database error.

- AutoCommit controls whether statements automatically commit their transactions when they complete. DBI defaults to HP Vertica's default AutoCommit value of true. Always set AutoCommit to false (0) when bulk loading data to increase database efficiency.

```
#!/usr/bin/perl
use strict;
use DBI;
```

```
# Create a hash that holds attributes for the connection
 my $attr = {
                RaiseError => 1, # Make database errors fatal to script
                AutoCommit => 0, # Prevent statements from committing
                                 # their transactions.
            };
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
    $attr);

if (defined $dbh->err) {
    # Connection failed.
    die "Failed to connect: $DBI::errstr";
}
print "Connected!\n";
# The database handle lets you access the connection attributes directly:
print "The AutoCommit attribute is: " . $dbh->{AutoCommit} . "\n";
print "The RaiseError attribute is: " . $dbh->{RaiseError} . "\n";
# And you can change values, too...
$dbh->{AutoCommit} = 1;
print "The AutoCommit attribute is now: " . $dbh->{AutoCommit} . "\n";
$dbh->disconnect();
```

The example outputs the following when run:

```
Connected!The AutoCommit attribute is: 0
The RaiseError attribute is: 1
The AutoCommit attribute is now: 1
```

Connecting From Perl Without a DSN

If you do not want to set up a Data Source Name (DSN) for your database, you can supply all of the information Perl's DBD::ODBC driver requires to connect to your HP Vertica database in the data source string. This source string must the `DRIVER=` parameter that tells DBD::ODBC which driver library to use in order to connect. The value for this parameter is the name assigned to the driver by the client system's driver manager:

- On Windows, the name assigned to the HP Vertica ODBC driver by the driver manager is Vertica.

- On Linux and other UNIX-like operating systems, the HP Vertica ODBC driver's name is assigned in the system's `odbcinst.ini` file. For example, if your `/etc/odbcint.ini` contains the following:

  ```
  [HPVertica]
  Description = HP Vertica ODBC Driver
  Driver = /opt/vertica/lib64/libverticaodbc.so
  ```

  you would use the name HPVertica. See Creating an ODBC DSN for Linux, Solaris, AIX, and HP-UX for more information about the `odbcinst.ini` file.

You can take advantage of Perl's variable expansion within strings to use variables for most of the connection properties as the following example demonstrates.

```
#!/usr/bin/perl
use strict;
use DBI;
my $server='VerticaHost';
my $port = '5433';
my $database = 'VMart';
my $user = 'ExampleUser';
my $password = 'password123';
# Connect without a DSN by supplying all of the information for the connection.
# The DRIVER value on UNIX platforms depends on the entry in the odbcinst.ini
# file.
my $dbh = DBI->connect("dbi:ODBC:DRIVER={Vertica};Server=$server;" .
         "Port=$port;Database=$database;UID=$user;PWD=$password")
         or die "Could not connect to database: " . DBI::errstr;
print "Connected!\n";
$dbh->disconnect();
```

Note: Surrounding the driver name with braces ({ and }) in the source string is optional.

Executing Statements Using Perl

Once your Perl script has connected to HP Vertica (see Connecting to HP Vertica Using Perl), it can execute simple statements that return a value rather than a result set by using the Perl DBI module's do function. You usually use this function to execute **DDL** statements or data loading statements such as COPY (see Using COPY LOCAL to Load Data in Perl).

```
#!/usr/bin/perl
use strict;
use DBI;
# Disable autocommit
 my $attr = {AutoCommit => 0};
# Open a connection using a DSN.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
    $attr);
unless (defined $dbh) {
    # Conection failed.
    die "Failed to connect: $DBI::errstr";
}
# You can use the do function to perform DDL commands.
# Drop any existing table.
$dbh->do("DROP TABLE IF EXISTS TEST CASCADE;");
# Create a table to hold data.
$dbh->do("CREATE TABLE TEST( \
               C_ID  INT, \
               C_FP  FLOAT,\
               C_VARCHAR VARCHAR(100),\
               C_DATE DATE, C_TIME TIME,\
               C_TS TIMESTAMP,\
               C_BOOL BOOL)");
# Commit changes and exit.
```

```
$dbh->commit();
$dbh->disconnect();
```

Note: The do function returns the number of rows that were affected by the statement (or -1 if the count of rows doesn't apply or is unavailable). Usually, the only time you need to consult this value is after you deleted a number of rows or if you used a bulk load command such as COPY. You use other DBI functions instead of do to perform batch inserts and selects (see Batch Loading Data Using Perl and Querying HP Vertica Using Perl for details).

Batch Loading Data Using Perl

To load large batches of data into HP Vertica using Perl:

1. Set DBI's AutoCommit connection attribute to false to improve the batch load speed. See Setting Perl DBI Connection Attributes for an example of disabling AutoCommit.

2. Call the database handle's `prepare` function to prepare a SQL INSERT statement that contains placeholders for the data values you want to insert. For example:

   ```
   # Prepare an INSERT statement for the test table
   $sth = $dbh->prepare("INSERT into test values(?,?,?,?,?,?,?)");
   ```

   The `prepare` function returns a statement handle that you will use to insert the data.

3. Assign data to the placeholders. There are several ways to do this. The easiest is to populate an array with a value for each placeholder in your INSERT statement.

4. Call the statement handle's `execute` function to insert a row of data into HP Vertica. The return value of this function call lets you know whether HP Vertica accepted or rejected the row.

5. Repeat steps 3 and 4 until you have loaded all of the data you need to load.

6. Call the database handle's `commit` function to commit the data you inserted.

The following example demonstrates inserting a small batch of data by populating an array of arrays with data, then looping through it and inserting each row.

```
#!/usr/bin/perl
use strict;
use DBI;
# Create a hash reference that holds a hash of parameters for the
# connection.
 my $attr = {AutoCommit => 0, # Turn off autocommit
             PrintError => 0   # Turn off automatic error printing.
                               # This is handled manually.
             };
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
```

```
    $attr);
if (defined DBI::err) {
    # Conection failed.
    die "Failed to connect: $DBI::errstr";
}
print "Connection AutoCommit state is: " . $dbh->{AutoCommit} . "\n";
# Create table to hold inserted data
$dbh->do("DROP TABLE IF EXISTS TEST CASCADE;") or die "Could not drop table";
$dbh->do("CREATE TABLE TEST( \
               C_ID  INT, \
               C_FP  FLOAT,\
               C_VARCHAR VARCHAR(100),\
               C_DATE DATE, C_TIME TIME,\
               C_TS TIMESTAMP,\
               C_BOOL BOOL)") or die "Could not create table";
# Populate an array of arrays with values. One of these rows contains
# data that will not be sucessfully inserted. Another contains an
# undef value, which gets inserted into the database as a NULL.
my @data = (
                [1,1.111,'Hello World!','2001-01-01','01:01:01'
                    ,'2001-01-01 01:01:01','t'],
                [2,2.22222,'How are you?','2002-02-02','02:02:02'
                    ,'2002-02-02 02:02:02','f'],
                ['bad value',2.22222,'How are you?','2002-02-02','02:02:02'
                    ,'2002-02-02 02:02:02','f'],
                [4,4.22222,undef,'2002-02-02','02:02:02'
                    ,'2002-02-02 02:02:02','f'],
             );
# Create a prepared statement to use parameters for inserting values.
my $sth = $dbh->prepare_cached("INSERT into test values(?,?,?,?,?,?,?)");
my $rowcount = 0; # Count # of rows
# Loop through the arrays to insert values
foreach my $tuple (@data) {
    $rowcount++;
    # Insert the row
    my $retval = $sth->execute(@$tuple);

    # See if the row was successfully inserted.
    if ($retval == 1) {
        # Value of 1 means the row was inserted (1 row was affected by insert)
        print "Row $rowcount successfully inserted\n";
    } else {
        print "Inserting row $rowcount failed";
        # Error message is not set on some platforms/versions of DBUI. Check to
        # ensure a message exists to avoid getting an unitialized var warning.
        if ($sth->err()) {
                print ": " . $sth->errstr();
        }
        print "\n";
    }
}
# Commit changes. With AutoCommit off, you need to use commit for batched
# data to actually be committed into the database. If your Perl script exits
# without committing its data, HP Vertica rolls back the transaction and the
# data is not committed.
$dbh->commit();
$dbh->disconnect();
```

The previous example displays the following when successfully run:

```
Connection AutoCommit state is: 0
Row 1 successfully inserted
Row 2 successfully inserted
Inserting row 3 failed with error 01000 [Vertica][VerticaDSII] (20) An
error occurred during query execution: Row rejected by server; see
server log for details (SQL-01000)
Row 4 successfully inserted
```

Note that one of the rows was not inserted because it contained a string value that could not be stored in an integer column. See Conversions Between Perl and HP Vertica Data Types for details of data type handling in Perl scripts that communicate with HP Vertica.

Using COPY LOCAL to Load Data in Perl

If you have delimited files (for example, a file with comma-separated values) on your client system that you want to load into HP Vertica, you can use the COPY LOCAL statement to directly load the file's contents into HP Vertica instead of using Perl to read, parse, and then batch insert the data. You execute a COPY LOCAL statement to load the file from the local filesystem. The result of executing the statement is the number of rows that were successfully inserted.

The following example code demonstrates loading a file named data.txt and located in the same directory as the Perl file into HP Vertica using a COPY LOCAL statement.

```
#!/usr/bin/perl
use strict;
use DBI;
# Filesystem path handling module
use File::Spec;
# Create a hash reference that holds a hash of parameters for the
# connection.
 my $attr = {AutoCommit => 0}; # Turn off AutoCommit
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
    $attr) or die "Failed to connect: $DBI::errstr";
print "Connected!\n";
# Drop any existing table.
$dbh->do("DROP TABLE IF EXISTS Customers CASCADE;");
# Create a table to hold data.
$dbh->do("CREATE TABLE Customers( \
               ID  INT, \
               FirstName  VARCHAR(100),\
               LastName   VARCHAR(100),\
               Email      VARCHAR(100),\
               Birthday   DATE)");
# Find the absolute path to the data file located in the current working
# directory and named data.txt
my $currDir = File::Spec->rel2abs(File::Spec->curdir());
my $dataFile = File::Spec->catfile($currDir, 'data.txt');
print "Loading file $dataFile\n";
# Load local file using copy local. Return value is the # of rows affected
# which equates to the number of rows inserted.
my $rows = $dbh->do("COPY Customers FROM LOCAL '$dataFile' DIRECT")
     or die $dbh->errstr;
print "Copied $rows rows into database.\n";
```

```
$dbh->commit();
# Prepare a query to get the first 15 rows of the results
my $sth = $dbh->prepare("SELECT * FROM Customers WHERE ID < 15 \
                                 ORDER BY ID");

$sth->execute() or die "Error querying table: " . $dbh->errstr;
my @row; # Pre-declare variable to hold result row used in format statement.
# Use Perl formats to pretty print the output. Declare the heading for the
# form.
format STDOUT_TOP =
ID  First           Last          EMail                         Birthday
==  =====           ====          =====                         ========
.
# The Perl write statement will output a formatted line with values from the
# @row array. See http://perldoc.perl.org/perlform.html for details.
format STDOUT =
@>  @<<<<<<<<<<<<<  @<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<<<<<<<<  @<<<<<<<<<
@row
.
# Loop through result rows while we have them
while (@row = $sth->fetchrow_array()) {
         write; # Format command does the work of extracting the columns from
                # the @row array and writing them out to STDOUT.
}
# Call commit to prevent Perl from complaining about uncommitted transactions
# when disconnecting
$dbh->commit();
$dbh->disconnect();
```

The data.txt file is a text file containing a row of data on each line. The columns are delimited by pipe (|) characters. This is the default format that the COPY command accepts, which makes the COPY LOCAL statement in the example code simple. See the COPY statement entry in the SQL Reference Manual for handling data files that are in different formats. Here is an example of the content in this file:

```
1|Georgia|Gomez|Rhiannon@magna.us|1937-10-03
2|Abdul|Alexander|Kathleen@ipsum.gov|1941-03-10
3|Nigel|Contreras|Tanner@et.com|1955-06-01
4|Gray|Holt|Thomas@Integer.us|1945-12-06
5|Candace|Bullock|Scott@vitae.gov|1932-05-27
6|Matthew|Dotson|Keith@Cras.com|1956-09-30
7|Haviva|Hopper|Morgan@porttitor.edu|1975-05-10
8|Stewart|Sweeney|Rhonda@lectus.us|2003-06-20
9|Allen|Rogers|Alexander@enim.gov|2006-06-17
10|Trevor|Dillon|Eagan@id.org|1988-11-27
11|Leroy|Ashley|Carter@turpis.edu|1958-07-25
12|Elmo|Malone|Carla@enim.edu|1978-08-29
13|Laurel|Ball|Zelenia@Integer.us|1989-09-20
14|Zeus|Phillips|Branden@blandit.gov|1996-08-08
15|Alexis|Mclean|Flavia@Suspendisse.org|2008-01-07
```

The example code produces the following output when run on a large sample file:

```
Connected!
```

```
Loading file /home/dbadmin/Perl/data.txt
Copied 1000000 rows into database.
ID  First           Last          EMail                         Birthday
==  =====           ====          =====                         ========
 1  Georgia         Gomez         Rhiannon@magna.us             1937-10-03
 2  Abdul           Alexander     Kathleen@ipsum.gov            1941-03-10
 3  Nigel           Contreras     Tanner@et.com                 1955-06-01
 4  Gray            Holt          Thomas@Integer.us             1945-12-06
 5  Candace         Bullock       Scott@vitae.gov               1932-05-27
 6  Matthew         Dotson        Keith@Cras.com                1956-09-30
 7  Haviva          Hopper        Morgan@porttitor.edu          1975-05-10
 8  Stewart         Sweeney       Rhonda@lectus.us              2003-06-20
 9  Allen           Rogers        Alexander@enim.gov            2006-06-17
10  Trevor          Dillon        Eagan@id.org                  1988-11-27
11  Leroy           Ashley        Carter@turpis.edu             1958-07-25
12  Elmo            Malone        Carla@enim.edu                1978-08-29
13  Laurel          Ball          Zelenia@Integer.us            1989-09-20
14  Zeus            Phillips      Branden@blandit.gov           1996-08-08
```

Note: Loading a single, large data file into HP Vertica through a single data connection is less efficient than loading a number of smaller files onto multiple nodes in parallel. Loading onto multiple nodes prevents any one node from becoming a bottleneck.

Querying HP Vertica Using Perl

To query HP Vertica using Perl:

1. Prepare a query statement using the Perl DBI module's `prepare` function. This function returns a statement handle that you use to execute the query and get the result set.

2. Execute the prepared statement by calling the `execute` function on the statement handle.

3. Retrieve the results of the query from the statement handle using one of several methods, such as calling the statement handle's `fetchrow_array` function to retrieve a row of data, or `fetchall_array` to get an array of arrays containing the entire result set (not a good idea if your result set may be very large!).

The following example demonstrates querying the table created by the example shown in Batch Loading Data Using Perl. It executes a query to retrieve all of the content of the table, then repeatedly calls the statement handle's `fetchrow_array` function to get rows of data in an array. It repeats this process until `fetchrow_array` returns undef, which means that there are no more rows to be read.

```
#!/usr/bin/perl
use strict;
use DBI;
my $attr = {RaiseError => 1 }; # Make errors fatal to the Perl script.
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
                        $attr);
```

```
# Prepare a query to get the content of the table
my $sth = $dbh->prepare("SELECT * FROM TEST ORDER BY C_ID ASC");
# Execute the query by calling execute on the statement handle
$sth->execute();
# Loop through result rows while we have them, getting each row as an array
while (my @row = $sth->fetchrow_array()) {
    # The @row array contains the column values for this row of data
    # Loop through the column values
    foreach my $column (@row) {
        if (!defined $column) {
            # NULLs are signaled by undefs. Set to NULL for clarity
            $column = "NULL";
        }
        print "$column\t"; # Output the column separated by a tab
    }
    print "\n";
}
$dbh->disconnect();
```

The example prints the following when run:

```
1       1.111   Hello World!    2001-01-01      01:01:01        2001-01-01 01:01:01     1
2      2.22222 How are you?    2002-02-02      02:02:02        2002-02-02 02:02:02     0
4      4.22222 NULL    2002-02-02      02:02:02        2002-02-02 02:02:02     0
```

Binding Variables to Column Values

Another method of retrieving the query results is to bind variables to columns in the result set using the statement handle's `bind_columns` function. You may find this method convenient if you need to perform extensive processing on the returned data, since your code can use variables rather than array references to access the data. The following example demonstrates binding variables to the result set, rather than looping through the row and column values.

```
#!/usr/bin/perl
use strict;
use DBI;
my $attr = {RaiseError => 1 }; # Make SQL errors fatal to the Perl script.
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN32","ExampleUser","password123",
                        $attr);
# Prepare a query to get the content of the table
my $sth = $dbh->prepare("SELECT * FROM TEST ORDER BY C_ID ASC");
$sth->execute();
# Create a set of variables to bind to the column values.
my ($C_ID, $C_FP, $C_VARCHAR, $C_DATE, $C_TIME, $C_TS, $C_BOOL);
# Bind the variable references to the columns in the result set.
$sth->bind_columns(\$C_ID, \$C_FP, \$C_VARCHAR, \$C_DATE, \$C_TIME,
                    \$C_TS, \$C_BOOL);

# Now, calling fetch() to get a row of data updates the values of the bound
# variables. Continue calling fetch until it returns undefined.
while ($sth->fetch()) {
    # Note, you should always check that values are defined before using them,
    # since NULL values are translated into Perl as undefined. For this
    # example, just check the VARCHAR column for undefined values.
    if (!defined $C_VARCHAR) {
        $C_VARCHAR = "NULL";
    }
    # Just print values separated by tabs.
    print "$C_ID\t$C_FP\t$C_VARCHAR\t$C_DATE\t$C_TIME\t$C_TS\t$C_BOOL\n";
}
$dbh->disconnect();
```

The output of this example is identical to the output of the previous example.

Preparing, Querying, and Returning a Single Row

If you expect a single row as the result of a query (for example, when you execute a COUNT (*) query), you can use the DBI module's `selectrow_array` function to combine executing a statement and retrieving an array as a result.

The following example shows using `selectrow_array` to execute and get the results of the SHOW LOCALE statement. It also demonstrates changing the locale using the `do` function.

```
#!/usr/bin/perl
use strict;
use DBI;
my $attr = {RaiseError => 1 }; # Make SQL errors fatal to the Perl script.
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
                        $attr);
# Demonstrate setting/getting locale.
# Use selectrow_array to combine preparing a statement, executing it, and
# getting an array as a result.
my @localerv = $dbh->selectrow_array("SHOW LOCALE;");
# The locale name is the 2nd column (array index 1) in the result set.
print "Locale: $localerv[1]\n";
# Use do() to execute a SQL statement to set the locale.
$dbh->do("SET LOCALE TO en_GB");
# Get the locale again.
@localerv = $dbh->selectrow_array("SHOW LOCALE;");
print "Locale is now: $localerv[1]\n";
$dbh->disconnect();
```

The result of running the example is:

```
Locale: en_US@collation=binary (LEN_KBINARY)
Locale is now: en_GB (LEN)
```

Conversions Between Perl and HP Vertica Data Types

Perl is a loosely-typed programming language that does not assign specific data types to values. It converts between string and numeric values based on the operations being performed on the values. For this reason, Perl has little problem extracting most string and numeric data types from HP Vertica. All interval data types (DATE, TIMESTAMP, etc.) are converted to strings. You can use several different date and time handling Perl modules to manipulate these values in your scripts.

HP Vertica NULL values translate to Perl's undefined (`undef`) value. When reading data from columns that can contain NULL values, you should always test whether a value is defined before using it.

When inserting data into HP Vertica, Perl's DBI module attempts to coerce the data into the correct format. By default, it assumes column values are VARCHAR unless it can determine that they are some other data type. If given a string value to insert into a column that has an integer or numeric data type, DBI attempts to convert the string's contents to the correct data type. If the entire string can be converted to a value of the appropriate data type, it inserts the value into the column. If not, inserting the row of data fails.

DBI transparently converts integer values into numeric or float values when inserting into column of FLOAT, NUMERIC, or similar data types. It converts numeric or floating values to integers only when there would be no loss of precision (the value to the right of the decimal point is 0). For example, it can insert the value 3.0 into an INTEGER column since there is no loss of precision when converting the value to an integer. It cannot insert 3.1 into an INTEGER column, since that would result in a loss of precision. It returns an error instead of truncating the value to 3.

The following example demonstrates some of the conversions that the DBI module performs when inserting data into HP Vertica.

```
#!/usr/bin/perl
use strict;
use DBI;
# Create a hash reference that holds a hash of parameters for the
# connection.
 my $attr = {AutoCommit => 0, # Turn off autocommit
             PrintError => 0   # Turn off print error. Manually handled
             };
# Open a connection using a DSN. Supply the username and password.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123",
    $attr);
if (defined DBI::err) {
    # Conection failed.
    die "Failed to connect: $DBI::errstr";
}
print "Connection AutoCommit state is: " . $dbh->{AutoCommit} . "\n";
# Create table to hold inserted data
$dbh->do("DROP TABLE IF EXISTS TEST CASCADE;");
$dbh->do("CREATE TABLE TEST( \
               C_ID  INT, \
               C_FP  FLOAT,\
               C_VARCHAR VARCHAR(100),\
               C_DATE DATE, C_TIME TIME,\
               C_TS TIMESTAMP,\
               C_BOOL BOOL)");
# Populate an array of arrays with values.
my @data = (
                # Start with matching data types
                [1,1.111,'Matching datatypes','2001-01-01','01:01:01'
```

```
                      ,'2001-01-01 01:01:01','t'],
                  # Force floats -> int and int -> float.
                  [2.0,2,"Ints <-> floats",'2002-02-02','02:02:02'
                      ,'2002-02-02 02:02:02',1],
                  # Float -> int *only* works when there is no loss of precision.
                  # this row will fail to insert:
                  [3.1,3,"float -> int with trunc?",'2003-03-03','03:03:03'
                      ,'2003-03-03 03:03:03',1],
                  # String values are converted into numbers
                  ["4","4.4","Strings -> numbers", '2004-04-04','04:04:04',
                      ,'2004-04-04 04:04:04',0],
                  # String -> numbers only works if the entire string can be
                  # converted into a number
                  ["5 and a half","5.5","Strings -> numbers", '2005-05-05',
                      '05:05:05', ,'2005-05-05 05:05:05',0],
                  # Number are converted into string values automatically,
                  # assuming they fit into the column width.
                  [6,6.6,3.14159, '2006-06-06','06:06:06',
                      ,'2006-06-06 06:06:06',0],
                  # There are some variations in the accepted date strings
                  [7,7.7,'Date/time formats', '07/07/2007','07:07:07',
                      ,'07-07-2007 07:07:07',1],
               );
# Create a prepared statement to use parameters for inserting values.
my $sth = $dbh->prepare_cached("INSERT into test values(?,?,?,?,?,?,?)");
my $rowcount = 0; # Count # of rows
# Loop through the arrays to insert values
foreach my $tuple (@data) {
    $rowcount++;
    # Insert the row
    my $retval = $sth->execute(@$tuple);

    # See if the row was successfully inserted.
    if ($retval == 1) {
        # Value of 1 means the row was inserted (1 row was affected by insert)
        print "Row $rowcount successfully inserted\n";
    } else {
        print "Inserting row $rowcount failed with error " .
                $sth->state . " " . $sth->errstr . "\n";
    }
}
# Commit the data
$dbh->commit();
# Prepare a query to get the content of the table
$sth = $dbh->prepare("SELECT * FROM TEST ORDER BY C_ID ASC");
$sth->execute() or die "Error: " . $dbh->errstr;
my @row; # Need to pre-declare to use in the format statement.
# Use Perl formats to pretty print the output.
format STDOUT_TOP =
Int  Float          VarChar        Date      Time      Timestamp     Bool
===  =====  ==================  ========== ======== ================ ====
.
format STDOUT =
@>>  @<<<<  @<<<<<<<<<<<<<<<<<  @<<<<<<<<< @<<<<<<< @<<<<<<<<<<<<<<< @<<<<
@row
.
# Loop through result rows while we have them
while (@row = $sth->fetchrow_array()) {
```

```
         write; # Format command does the work of extracting the columsn from
                 # the array.
}
# Commit to stop Perl complaining about in-progress transactions.
$dbh->commit();
$dbh->disconnect();
```

The example produces the following output when run:

```
Connection AutoCommit state is: 0
Row 1 successfully inserted
Row 2 successfully inserted
Inserting row 3 failed with error 01000 [Vertica][VerticaDSII] (20) An error
occurred during query execution: Row rejected by server; see server log for
details (SQL-01000)
Row 4 successfully inserted
Inserting row 5 failed with error 01000 [Vertica][VerticaDSII] (20) An error
occurred during query execution: Row rejected by server; see server log for
details (SQL-01000)
Row 6 successfully inserted
Row 7 successfully inserted
Int  Float          VarChar        Date      Time      Timestamp     Bool
===  =====  ==================  ========== ======== ================ ====
  1  1.111  Matching datatypes  2001-01-01 01:01:01 2001-01-01 01:01 1
  2  2      Ints <-> floats     2002-02-02 02:02:02 2002-02-02 02:02 1
  4  4.4    Strings -> numbers  2004-04-04 04:04:04 2004-04-04 04:04 0
  6  6.6    3.14159             2006-06-06 06:06:06 2006-06-06 06:06 0
  7  7.7    Date/time formats   2007-07-07 07:07:07 2007-07-07 07:07 1
```

Perl Unicode Support

Perl supports Unicode data with some caveats. See the perlunicode and the perlunitut (Perl Unicode tutorial) manual pages for details. (Be sure to see the copies of these manual pages included with the version of Perl installed on your client system, as the support for Unicode has changed in recent versions of Perl.) Perl DBI and DBD::ODBC also support Unicode, however DBD::ODBC must be compiled with Unicode support. See the DBD::ODBC documentation for details. You can check the DBD::ODBC-specific connection attribute named `odbc_has_unicode` to see if Unicode support is enabled in the driver.

The following example Perl script demonstrates directly inserting UTF-8 strings into HP Vertica and then reading them back. The example writes a text file with the output, since there are may problems displaying Unicode characters in terminal windows or consoles.

```
#!/usr/bin/perl
use strict;
use DBI;
# Open a connection using a DSN.
my $dbh = DBI->connect("dbi:ODBC:VerticaDSN","ExampleUser","password123");
unless (defined $dbh) {
    # Conection failed.
    die "Failed to connect: $DBI::errstr";
}
```

```
# Output to a file. Displaying Unicode characters to a console or terminal
# window has many problems. This outputs a UTF-8 text file that can
# be handled by many Unicode-aware text editors:
open OUTFILE, '>:utf8', "unicodeout.txt";
# See if the DBD::ODBC driver was compiled with Unicode support. If this returns
# 1, your Perl script will get get strings from the driver with the UTF-8
# flag set on them, ensuring that Perl handles them correctly.
print OUTFILE "Was DBD::ODBC compiled with Unicode support? " .
    $dbh->{odbc_has_unicode} . "\n";

# Create a table to hold VARCHARs
$dbh->do("DROP TABLE IF EXISTS TEST CASCADE;");

# Create a table to hold data. Remember that the width of the VARCHAR column
# is the number of bytes set aside to store strings, which often does not equal
# the number of characters it can hold when it comes to Unicode!
$dbh->do("CREATE TABLE test( C_VARCHAR VARCHAR(100) )");
print OUTFILE "Inserting data...\n";
# Use Do to perform simple inserts
$dbh->do("INSERT INTO test VALUES('Hello')");
# This string contains several non-latin accented characters and symbols, encoded
# with Unicode escape notation. They are converted by Perl into UTF-8 characters
$dbh->do("INSERT INTO test VALUES('My favorite band is " .
    "\N{U+00DC}ml\N{U+00E4}\N{U+00FC}t \N{U+00D6}v\N{U+00EB}rk\N{U+00EF}ll" .
    " \N{U+263A}')");
# Some Chinese (Simplified) characters. This again uses escape sequence
# that Perl translates into UTF-8 characters.
$dbh->do("INSERT INTO test VALUES('\x{4F60}\x{597D}')");
print OUTFILE "Getting data...\n";
# Prepare a query to get the content of the table
my $sth = $dbh->prepare_cached("SELECT * FROM test");
# Execute the query by calling execute on the statement handle
$sth->execute();
# Loop through result rows while we have them
while (my @row = $sth->fetchrow_array()) {
    # Loop through the column values
    foreach my $column (@row) {
        print OUTFILE "$column\t";
    }
    print OUTFILE "\n";
}
close OUTFILE;
$dbh->disconnect();
```

Viewing the unicodeout.txt file in a UTF-8-capable text editor or viewer displays:

```
Was DBD::ODBC compiled with Unicode support? 1
Inserting data...
Getting data...
My favorite band is Ümläüt Övërkïll ☺
你好
Hello
```

Note: Terminal windows and consoles often have problems properly displaying Unicode characters. That is why the example writes the output to a text file. With some text editors, you may need to manually set the encoding of the text file to UTF-8 in order for the characters to properly appear (and the font used to display text must have a full Unicode character set). If the character still do not show up, it may be that your version of DBD::ODBC was not compiled with UTF-8 support.

See Also

- Unicode Character Encoding
- Additional ODBC Driver Configuration Settings

Programming PHP Client Applications

You can connect to HP Vertica through PHP-ODBC using the Unix ODBC or iODBC library.

In order to use PHP with HP Vertica, you must install the following packages (and their dependencies):

- php
- php-odbc
- php-pdo
- UnixODBC (if you are using the Unix ODBC driver)
- libiodbc (if you are using the iODBC driver)

PHP on Linux

PHP is available with most Linux operating systems as a module for the Apache web server. Check your particular Linux repository for PHP RPMs or Debian packages. You can also build PHP from source. See the PHP web site for documentation and source downloads.

PHP on Windows

PHP is available for windows for both the Apache and IIS web servers. You can download PHP for Windows and view installation instructions at the PHP web site.

The PHP ODBC Drivers

PHP supports both the UnixODBC drivers and iODBC drivers. Both drivers use PHP's ODBC database abstraction layer.

Setup

You must read Programming ODBC Client Applications before connecting to HP Vertica through PHP. The following example ODBC configuration entries detail the typical settings required for PHP ODBC connections. The driver location assumes you have copied the HP Vertica drivers to /usr/lib64.

Example odbc.ini

```
[ODBC Data Sources]
VerticaDSNunixodbc = exampledb
VerticaDNSiodbc = exampledb2
[VerticaDSNunixodbc]
Description = VerticaDSN Unix ODBC driver
```

```
Driver = /usr/lib64/libverticaodbc.so
Database = Telecom
Servername = localhost
UserName = dbadmin
Password =
Port = 5433
[VerticaDSNiodbc]
Description = VerticaDSN iODBC driver
Driver = /usr/lib64/libverticaodbc.so
Database = Telecom
Servername = localhost
UserName = dbadmin
Password =
Port = 5433
```

Example odbcinst.ini

```
# Vertica
[VerticaDSNunixodbc]
Description = VerticaDSN Unix ODBC driver
Driver = /usr/lib64/libverticaodbc.so
[VerticaDNSiodbc]
Description = VerticaDSN iODBC driver
Driver = /usr/lib64/libverticaodbc.so
[ODBC]
Threading = 1
```

Verify the HP Vertica UnixODBC or iODBC Library

Verify the HP Vertica UnixODBC library can load all dependant libraries with the following command (assuming you have copies the libraries to /usr/lib64):

For example:

```
ldd /usr/lib64/libverticaodbc.so
```

You must resolve any "not found" libraries before continuing.

Test Your ODBC Connection

Test your ODBC connection with the following.

```
isql -v VerticaDSN
```

PHP Unicode Support

PHP does not offer native Unicode support. PHP only supports a 256-character set. However, PHP provides the UTF-8 functions utf8_encode() and utf8_decode() to provide some basic Unicode functionality.

See the PHP manual for strings for more details about PHP and Unicode.

Querying the Database Using PHP

The example script below details the use of PHP ODBC functions to connect to the HP Vertica Analytics Platform.

```
<?php
# Turn on error reporting
error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);
# A simple function to trap errors from queries
function errortrap_odbc($conn, $sql) {
    if(!$rs = odbc_exec($conn,$sql)) {
        echo "<br/>Failed to execute SQL: $sql<br/>" . odbc_errormsg($conn);
    } else {
        echo "<br/>Success: " . $sql;
    }
    return $rs;
}
# Connect to the Database
$dsn = "VerticaDSNunixodbc";
$conn = odbc_connect($dsn,'','') or die ("<br/>CONNECTION ERROR");
echo "<p>Connected with DSN: $dsn</p>";
# Create a table
$sql = "CREATE TABLE TEST(
        C_ID INT,
        C_FP FLOAT,
        C_VARCHAR VARCHAR(100),
        C_DATE DATE, C_TIME TIME,
        C_TS TIMESTAMP,
        C_BOOL BOOL)";
$result = errortrap_odbc($conn, $sql);
# Insert data into the table with a standard SQL statement
$sql = "INSERT into test values(1,1.1,'abcdefg1234567890','1901-01-01','23:12:34
','1901-01-01 09:00:09','t')";
$result = errortrap_odbc($conn, $sql);
# Insert data into the table with odbc_prepare and odbc_execute
$values = array(2,2.28,'abcdefg1234567890','1901-01-01','23:12:34','1901-01-01 0
9:00:09','t');
$statement = odbc_prepare($conn,"INSERT into test values(?,?,?,?,?,?,?)");
if(!$result = odbc_execute($statement, $values)) {
            echo "<br/>odbc_execute Failed!";
} else {
            echo "<br/>Success: odbc_execute.";
}
# Get the data from the table and display it
$sql = "SELECT * FROM TEST";
if($result = errortrap_odbc($conn, $sql)) {
    echo "<pre>";
    while($row = odbc_fetch_array($result) ) {
               print_r($row);
    }
    echo "</pre>";
}
# Drop the table and projection
$sql = "DROP TABLE TEST CASCADE";
$result = errortrap_odbc($conn, $sql);
```

```
# Close the ODBC connection
odbc_close($conn);
?>
```

Example Output

The following is the example output from the script.

```
Success: CREATE TABLE TEST( C_ID INT, C_FP FLOAT, C_VARCHAR VARCHAR(100), C_DATE DATE, C_
TIME TIME, C_TS TIMESTAMP, C_BOOL BOOL)
Success: INSERT into test values(1,1.1,'abcdefg1234567890','1901-01-01','23:12:34 ','190
1-01-01 09:00:09','t')
Success: odbc_execute.
Success: SELECT * FROM TEST
Array
(
    [C_ID] => 1
    [C_FP] => 1.1
    [C_VARCHAR] => abcdefg1234567890
    [C_DATE] => 1901-01-01
    [C_TIME] => 23:12:34
    [C_TS] => 1901-01-01 09:00:09
    [C_BOOL] => 1
)
Array
(
    [C_ID] => 2
    [C_FP] => 2.28
    [C_VARCHAR] => abcdefg1234567890
    [C_DATE] => 1901-01-01
    [C_TIME] => 23:12:34
    [C_TS] => 1901-01-01 23:12:34
    [C_BOOL] => 1
)
Success: DROP TABLE TEST CASCADE
```

Using vsql

vsql is a character-based, interactive, front-end utility that lets you type SQL statements and see the results. It also provides a number of meta-commands and various shell-like features that facilitate writing scripts and automating a variety of tasks.

If you are using the vsql client installed on the server, then you can connect from the:

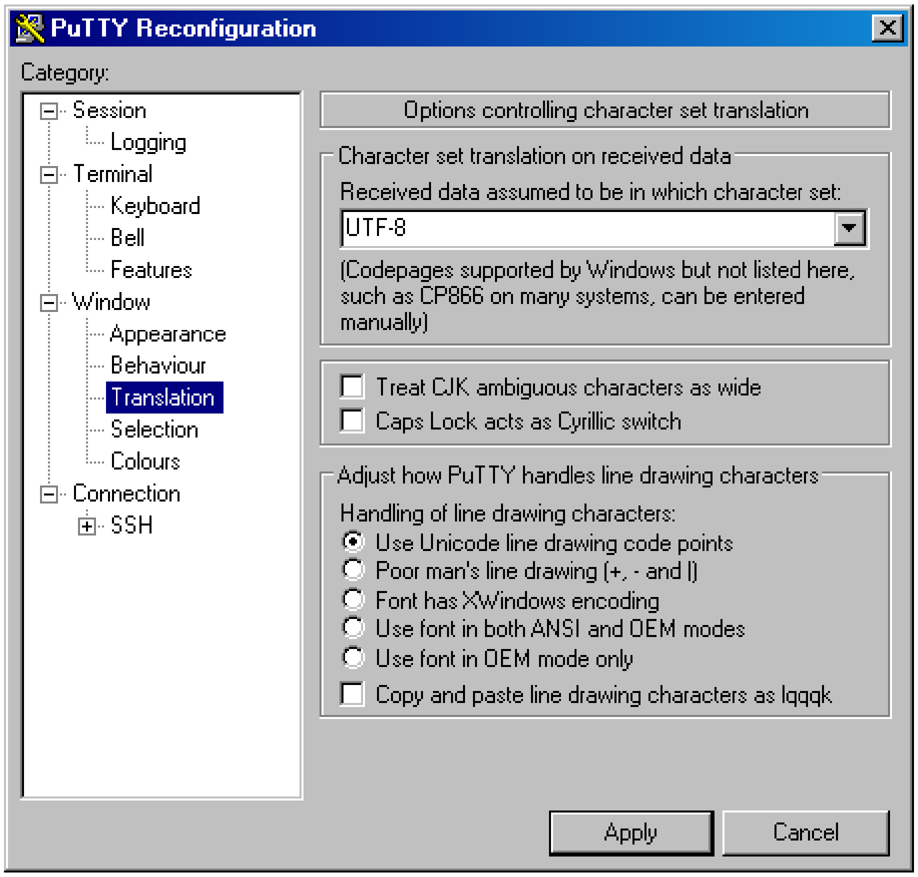
- Administration Tools
- Linux command line

You can also install the vsql client for other supported platforms.

A man page is available for vsql. If you are running as the dbadmin user, simply type: `man vsql`. If you are running as a different user, type: `man -M /opt/vertica/man vsql`.

General Notes

- SQL statements can be spread over several lines for clarity.
- vsql can handles input and output in UTF-8 encoding. Note that the terminal emulator running vsql must be set up to display the UTF-8 characters correctly. Follow the documentation of your terminal emulator. The following example shows the settings in PuTTy from the Change Settings > Window > Translation option:



Options controlling character set translation

Character set translation on received data

Received data assumed to be in which character set: UTF-8

(Codepages supported by Windows but not listed here, such as CP866 on many systems, can be entered manually)

- [ ] Treat CJK ambiguous characters as wide
- [ ] Caps Lock acts as Cyrillic switch

Adjust how PuTTY handles line drawing characters

Handling of line drawing characters:

- [✓ Use Unicode line drawing code points]
- Poor man's line drawing (+, - and |)
- Font has XWindows encoding
- Use font in both ANSI and OEM modes
- Use font in OEM mode only
- [ ] Copy and paste line drawing characters as lqqqk

See also Best Practices for Working with Locales.

- Cancel SQL statements by typing Ctrl+C.
- Traverse command history by typing Ctrl+R.
- When you disconnect a user session, any transactions in progress are automatically rolled back.
- To view wide result sets, use the Linux `less` utility to truncate long lines.

  a. Before connecting to the database, specify that you want to use `less` for query output:

  ```
  $ export PAGER=less
  ```

  b. Connect to the database.

  c. Query a wide table:

  ```
  => select * from wide_table;
  ```

d. At the `less` prompt, type:

```
-S
```

- If a shell running vsql fails (crashes or freezes), the vsql processes continue to run even if you stop the database. In that case, log in as root on the machine on which the shell was running and manually kill the vsql process. For example:

```
# ps -ef | grep vertica
⋮
 fred 2401 1 0 06:02 pts/1 00:00:00 /opt/vertica/bin/vsql -p 5433 -h
test01_site01 quick_start_single
⋮
# kill -9 2401
```

Installing the vsql Client

The vsql client is installed as part of the HP Vertica server rpm, but it is also available as a download for other Unix-based systems such as HP-UX, AIX, and Mac OSX.

How to Install vsql on Unix-Based systems:

1. Use a web browser to log in to the myVertica portal.
2. Click the Download tab, scroll down to the CLIENT PACKAGES section and download the appropriate vsql client from the HP Vertica downloads page. Note that there are both 32-bit and 64-bit versions for most platforms.
3. Extract the tarball. The tarball is organized to extract into /opt/vertica if you extract it at the root (/) of the drive.
4. Optionally add the directory where the vsql client resides to your path.
5. Verify mode on the vsql file is executable. For example: `chmod ugo+x vsql_`*VERSION*
6. Set your shell locale to a locale supported by vsql. For example, in your .profile, add, `export LANG=en_US.UTF-8`

Installing vsql on Windows:

vsql on Windows is installed as part of the Windows Client Driver package. To install vsql on windows see the instructions for installing the Windows Client Driver package.

See vsql Notes for Windows Users for details on using vsql in a windows console.

vsql Notes for Windows Users

Font

Set the console font to "Lucida Console", because the raster font does not work well with the ANSI code page.

Console Encoding

vsql is built as a "console application." The Windows console windows use a different encoding than the rest of the system, so take care when you use 8-bit characters within vsql. If vsql detects a problematic console code page, it warns you at startup. To change the console code page, two things are necessary:

- Set the code page by entering `cmd.exe /c chcp 1252`.

  1252 is a code page that is appropriate for European languages; replace it with your preferred locale code page.

Running Under Cygwin

Verify that your cygwin.bat file does not include the "tty" flag. If the "tty" flag is included in your cywgin.bat file, then banners and prompts are not displayed in vsql.

For example, change:

```
set CYGWIN=binmode tty ntsec
```

to:

```
set CYGWIN=binmode ntsec
```

Additionally, when running under Cygwin, vsql uses Cygwin shell conventions as opposed to Windows console conventions.

Tab Completion

Tab completion is a function of the shell, not vsql. Because of this, tab completion does not work the same way in Windows vsql as it does on Linux versions of vsql.

On Windows, instead of using tab-completion. Press F7 to pop-up a history window of commands, or press F8 after typing a few letters of a command to cycle through commands in the history buffer starting with the same letters.

Connecting From the Administration Tools

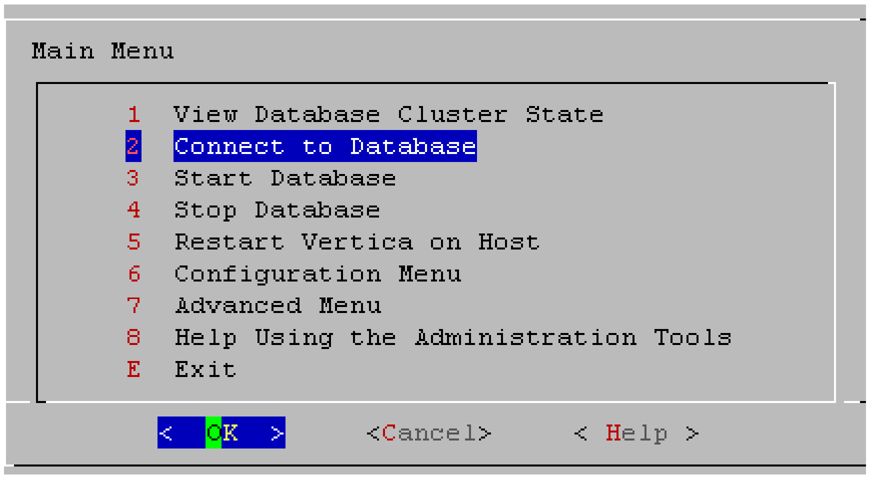
You can use the **Administration Tools** to connect to a database using vsql on any node in the cluster.

1. Log in as any user that does not have root privileges. (HP Vertica does not allow users with root privileges to connect to a database for security reasons).

2. Run the Administration Tools.

   ```
   /opt/vertica/bin/admintools
   ```

3. On the Main Menu, select Connect to Database.



```
Main Menu

        1  View Database Cluster State
        2  Connect to Database
        3  Start Database
        4  Stop Database
        5  Restart Vertica on Host
        6  Configuration Menu
        7  Advanced Menu
        8  Help Using the Administration Tools
        E  Exit

         <  OK  >      <Cancel>     < Help >
```

4. Supply the database password if asked:

```
Password:
```

When you create a new user with the CREATE USER command, you can configure the password or leave it empty. You cannot bypass the password if the user was created with a password configured. You can change a user's password using the ALTER USER command.

5. The Administration Tools connect to the database and transfer control to **vsql**.

```
Welcome to vsql, the Vertica Analytic Database interactive terminal.
Type:  \h or \? for help with vsql commands
       \g or terminate with semicolon to execute query
       \q to quit

=>
```

Note: See Meta-Commands for the various commands you can run while connected to the database through the Administration Tools.

Connecting From the Command Line

You can use vsql from the command line to connect to a database from any Linux machine, including those not part of the cluster. Copy `/opt/vertica/bin/vsql` to your machine.

Syntax

```
/opt/vertica/bin/vsql [ option...] [ dbname [ username ] ]
```

Parameters

option	One or more of the vsql Command Line Options
dbname	The name of the target database
username	The name of the user to connect as

Notes

- If the database is password protected, you must specify the `-w` or `--password` command line option.
- The default *dbname* and *username* is your Linux user name.
- If the connection cannot be made for any reason (for example, insufficient privileges, server is not running on the targeted host, etc.), vsql returns an error and terminates.
- vsql returns the following informational messages:
  - 0 to the shell if it finished normally
  - 1 if a fatal error of its own (out of memory, file not found) occurs
  - 2 if the connection to the server went bad and the session was not interactive
  - 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set
- Unrecognized words in the command line might be interpreted as database or user names.

Example

The following example redirects vsql output and error messages into an output file called retail_queries.out and captures any error messages:

```
$ vsql --echo-all < retail_queries.sql > retail_queries.out 2>&1
```

Command Line Options

This section contains the command-line options for vsql:

`-? --help` displays help about vsql command line arguments and exits.

`-a --echo-all` prints all input lines to standard output as they are read. This is more useful for script processing than interactive mode. It is equivalent to setting the variable ECHO to `all`.

`-A --no-align` switches to unaligned output mode. (The default output mode is aligned.)

`-B SERVER:PORT,SERVER:PORT,...` sets connection backup server/port. Comma separate multiple hosts. (default: not set).

`-c command --command command` runs one command and exits. This is useful in shell scripts. The command must be either a command string that can be completely parsed by the server (it contains no vsql specific features), or a single meta-command. In other words, you cannot mix SQL and vsql meta-commands. To achieve that, you can pipe the string into vsql like this:

```
echo "\\timing\\\\select * from t" | ../Linux64/bin/vsql
Timing is on.
 i | c | v
---+---+---
(0 rows)
```

Note: If you use double quotes with echo, you must double the backslashes.

`-C` Enables connection load balancing (default: not enabled)

`-d dbname --dbname dbname` specifies the name of the database to connect to. This is equivalent to specifying *dbname* as the first non-option argument on the command line.

`-e --echo-queries` copies all SQL commands sent to the server to standard output as well. This is equivalent to setting the variable ECHO to `queries`.

`-E` displays queries generated by internal commands.

`-f filename --file filename` uses the file *filename* as the source of commands instead of reading commands interactively. After the file is processed, vsql terminates. This is in many ways equivalent to the internal command `\i`.

If *filename* is - (hyphen), the standard input is read.

Using this option is subtly different from writing `vsql <` *filename*. In general, both do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option reduces the start-up overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output that you would have gotten had you entered everything by hand.

Using f filename to Read Data Piped into vsql

To read data piped into vsql from a data file:

1. Create the following:
   - A named pipe. For example, to create a named pipe called pipe1: `mkfifo pipe1`
   - A data file. The data file in this example is called *data_file*.
   - The command file that selects the table into which you want to copy data, copies the data from the pipe file (pipe1), and removes the pipe file. The command file in this example is called *command_line*.
2. From the command line, run a command that pipes the data file (data_file) into the appropriate table through vsql. The following example pipes the data file into public.shipping_dimension in the VMart database:

   ```
   cat data_file > pipe1 | vsql -f 'command_line'
   ```

   Example data_file:

   ```
   110|EXPRESS|SEA|FEDEX111|EXPRESS|HAND CARRY|MSC
   112|OVERNIGHT|COURIER|USPS
   ```

   Example command_line file:

   ```
   SELECT * FROM public.shipping_dimension;\set dir `pwd`/
   \set file '''':dir'pipe1'''
   COPY public.shipping_dimension FROM :file delimiter '|';
   SELECT * FROM public.shipping_dimension;
   --Remove the pipe1
   \! rm pipe1
   ```

`-F separator --field-separator separator` specifies the field separator for unaligned output (default: "|") (-P fieldsep=). (See `-A --no-align`.) This is equivalent to `\pset fieldsep` or `\f` .

`-h hostname --host hostname` specifies the host name of the machine on which the server is running.

Notes about -h hostname:

- If you are using client authentication with a connection method of either "gss" or" "krb5" (Kerberos), the -h hostname option is required.
- If you are using client authentication with a "local" connection type specified, do not use `-h hostname` if you want to match the client authentication entry.

`-H --html` turns on HTML tabular output. This is equivalent to `\pset format html` or the `\H` command.

`-k KRB SERVICE` provides the service name portion of the Kerberos principal (default: `vertica`). `-k` is equivalent to the drivers' `KerberosServiceName` connection string.

`-K KRB HOST` provides the instance or host name portion of the Kerberos principal. `-K` is equivalent to the drivers' `KerberosHostName` connection string.

`-l --list` returns all available databases, then exits. Other non-connection options are ignored. This command is similar to the internal command `\list`.

`-n` disables command line editing.

`-o filename --output filename` writes all query output into file *filename*. This is equivalent to the command `\o`.

`-p port --port port` specifies the TCP port or the local socket file extension on which the server is listening for connections. Defaults to port 5433.

`-P assignment --pset assignment` lets you specify printing options in the style of `\pset` on the command line. Note that you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

`-q --quiet` specifies that vsql do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this appears. This is useful with the `-c` option. Within vsql you can also set the QUIET variable to achieve the same effect.

`-R separator --record-separator separator` uses *separator* as the record separator. This is equivalent to the `\pset recordsep` command.

`-s --single-step` runs in single-step mode for debugging scripts. Forces vsql to prompt before each statement is sent to the database and allows you to cancel execution.

`-S --single-line` runs in single-line mode where a newline terminates a SQL command, like the semicolon does.

Note: This mode is provided for those who insist on it, but you are not necessarily encouraged to use it, particularly if you mix SQL and meta-commands on a line. The order of execution might not always be clear to the inexperienced user.

`-t --tuples-only` disables printing of column names, result row count footers, and so on. This is equivalent to the `\t` command.

`-T table_options --table-attr table_options` allows you to specify options to be placed within the HTML `table` tag. See `\pset` for details.

`-U username --username username` connects to the database as the user *username* instead of the default.

`-v assignment --set assignment --variable assignment` performs a variable assignment, like the `\set` internal command.

Note: You must separate name and value, if any, by an equal sign on the command line.

To unset a variable, omit the equal sign. To set a variable without a value, use the equal sign but omit the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes can get overwritten later.

`-V --version` prints the vsql version and exits.

`-w password` specifies the password for a database user.

> **Note:** Using this command line option displays the database password in plain text on the screen. Use it with care, particularly if you are connecting as the database administrator.

`-W --password` forces vsql to prompt for a password before connecting to a database.

The password is not displayed on the screen. This option remains set for the entire session, even if you change the database connection with the meta-command `\connect`.

`-x --expanded` enables extended table formatting mode. This is equivalent to the command `\ x`.

`-X, --no-vsqlrc` prevents the start-up file from being read (the system-wide `vsqlrc` file or the user's `~/.vsqlrc` file).

Connecting From a Non-Cluster Host

You can use the HP Vertica vsql executable image on a non-cluster Linux host to connect to an HP Vertica database.

- On Red Hat 5.0 64-bit and SUSE 10/11 64-bit, you can install the client driver RPM, which includes the vsql executable. See Installing the Client RPM on Red Hat and SUSE for details.

- If the non-cluster host is running the same version of Linux as the cluster, copy the image file to the remote system. For example:

  ```
  $ scp host01:/opt/vertica/bin/vsql .$ ./vsql
  ```

- If the non-cluster host is running a different version of Linux than your cluster hosts, and that operating system is not Red Hat version 5 64-bit or SUSE 10/11 64-bit, you must install the HP Vertica server RPM in order to get vsql. Download the appropriate rpm package from the Download tab of the myVertica portal then log into the non-cluster host as root and install the rpm package using the command:

  ```
  # rpm -Uvh filename
  ```

  In the above command, *filename* is package you downloaded. Note that you do not have to run the `install_HP Vertica` script on the non-cluster host in order to use vsql.

Notes

- Use the same Command Line Options that you would on a cluster host.

- You cannot run vsql on a Cygwin bash shell (Windows). Use ssh to connect to a cluster host, then run vsql.

Meta-Commands

Anything you enter in vsql that begins with an unquoted backslash is a vsql meta-command that is processed by vsql itself. These commands help make vsql more useful for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a vsql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace into an argument you can quote it with a single quote. To include a single quote into such an argument, precede it by a backslash. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\`*digits*, `\0`*digits*, and `\0x`*digits* (the character with the given decimal, octal, or hexadecimal code).

If an unquoted argument begins with a colon (`:`), it is taken as a vsql variable and the value of the variable is used as the argument instead.

Arguments that are enclosed in backquotes (`` ` ``) are taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) is taken as the argument value. The above escape sequences also apply in backquotes.

Some commands take a SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (") protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, `FOO"BAR"BAZ` is interpreted as `fooBARbaz`, and `"A weird"" name"` becomes `A weird" name`.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and vsql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

! [COMMAND]

`\! [ COMMAND ]` executes a command in a Linux shell (passing arguments as entered) or starts an interactive shell.

?

`\?` displays help information about the meta-commands. Works the same as \h .

```
=> \?
General
   \c[onnect] [DBNAME|- [USER]]
                  connect to new database (currently "VMart")
   \cd [DIR]      change the current working directory
   \q             quit vsql
   \set [NAME [VALUE]]
                  set internal variable, or list all if no parameters
   \timing        toggle timing of commands (currently off)
   \unset NAME    unset (delete) internal variable
   \! [COMMAND]   execute command in shell or start interactive shell
   \password [USER]
                  change user's password
Query Buffer
   \e [FILE]      edit the query buffer (or file) with external editor
   \g             send query buffer to server
   \g FILE        send query buffer to server and results to file
   \g | COMMAND   send query buffer to server and pipe results to command
   \p             show the contents of the query buffer
   \r             reset (clear) the query buffer
   \s [FILE]      display history or save it to file
   \w FILE        write query buffer to file
Input/Output
   \echo [STRING] write string to standard output
   \i FILE        execute commands from file
   \o FILE        send all query results to file
   \o | COMMAND   pipe all query results to command
   \o             close query-results file or pipe
   \qecho [STRING]
                  write string to query output stream (see \o)
Informational
   \d [PATTERN]   describe tables (list tables if no argument is supplied)
   \df [PATTERN]  list functions
   \dj [PATTERN]  list projections
   \dn [PATTERN]  list schemas
   \dp [PATTERN]  list table access privileges
   \ds [PATTERN]  list sequences
   \dS [PATTERN]  list system tables
   \dt [PATTERN]  list tables
   \dtv [PATTERN] list tables and views
   \dT [PATTERN]  list data types
   \du [PATTERN]  list users
   \dv [PATTERN]  list views
   \l             list all databases
   \z [PATTERN]   list table access privileges (same as \dp)
Formatting
   \a             toggle between unaligned and aligned output mode
   \b             toggle beep on command completion
   \C [STRING]    set table title, or unset if none
   \f [STRING]    show or set field separator for unaligned query output
   \H             toggle HTML output mode (currently off)
   \pset NAME [VALUE]
                  set table output option
                  (NAME := {format|border|expanded|fieldsep|footer|null|
                  recordsep|tuples_only|title|tableattr|pager})
   \t             show only rows (currently off)
   \T [STRING]    set HTML <table> tag attributes, or unset if none
   \x             toggle expanded output (currently off)
```

a

\a toggles output format alignment. This command is kept for backwards compatibility. See \pset for a more general solution.

`\a` is similar to the command line option -A --no-align, which only disables alignment.

b

`\b` toggles beep on command completion.

c (or \connect) [Dbname [Username]]

`\c (or \connect) [ dbname [ username ] ]` establishes a connection to a new database and/or under a user name. The previous connection is closed. If dbname is - the current database name is assumed.

If username is omitted the current user name is assumed.

As a special rule, `\connect` without any arguments connects to the default database as the default user (as you would have gotten by starting vsql without any arguments).

If the connection attempt fails (wrong user name, access denied, etc.), the previous connection is kept if and only if vsql is in interactive mode. When executing a non-interactive script, processing immediately stops with an error. This distinction that avoids typos and a prevent scripts from accidentally acting on the wrong database.

C [STRING]

`\C [ STRING ]` sets the title of any tables being printed as the result of a query or unsets any such title. This command is equivalent to `\pset title` *title*. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

cd [DIR]

`\cd [ DIR ]` changes the current working directory to *directory*. Without argument, changes to the current user's home directory.

To print your current working directory, use \! pwd. For example:

```
=> \!pwd
/home/dbadmin
```

The \d [PATTERN] Meta-Commands

This section describes the various \d meta-commands

All `\d` meta-commands take an optional pattern (asterisk [`*`] or question mark [`?`]) and return only the records that match that pattern.

The `?` argument is useful if you can't remember if a table name uses an underscore or a dash:

```
=> \dn v?internal
   List of schemas
```

```
    Name    |  Owner
------------+---------
 v_internal | dbadmin
(1 row)
```

The output from the `\d` metacommands places double quotes around non-alphanumeric table names and table names that are keywords, such as in the following example.

```
=> CREATE TABLE my_keywords.precision(x numeric (4,2));
CREATE TABLE
=> \d
                     List of tables
    Schema    |         Name          | Kind  |  Owner
--------------+-----------------------+-------+---------
 my_keywords  | "precision"           | table | dbadmin
```

Double quotes are optional when you use a `\d` command with pattern matching.

d [PATTERN]

The `\d` meta-command lists all tables in the database and returns their schema, table name, kind (e.g., table), and owner.

If you use `\d [ PATTERN ]` and provide the schema name or table name (or wildcard or ? characters) as the pattern argument, the result shows more detailed information about the tables:

- Schema name
- Table name
- Column name
- Column data type
- Data type size
- Default column value
- Whether the column accepts null values or has a `NOT NULL` constraint
- Whether there is a primary key or foreign key constraint

To view information about system tables, you must include the `V_MONITOR` or `V_CATALOG` as the pattern argument; for example:

```
\d v_catalog.types   -- information on the types table in v_catalog schema
\d v_catalog.*       -- information on all table columns in v_catalog schema
```

The following output is the result of all tables in the vmart schema, which is in the PUBLIC schema.

```
VMart=> \d
                     List of tables
    Schema    |         Name          | Kind  |  Owner
--------------+-----------------------+-------+---------
 online_sales | call_center_dimension | table | dbadmin
 online_sales | online_page_dimension | table | dbadmin
 online_sales | online_sales_fact     | table | dbadmin
 public       | customer_dimension    | table | dbadmin
 public       | date_dimension        | table | dbadmin
 public       | employee_dimension    | table | dbadmin
 public       | inventory_fact        | table | dbadmin
 public       | product_dimension     | table | dbadmin
 public       | promotion_dimension   | table | dbadmin
 public       | shipping_dimension    | table | dbadmin
 public       | vendor_dimension      | table | dbadmin
 public       | warehouse_dimension   | table | dbadmin
 store        | store_dimension       | table | dbadmin
 store        | store_orders_fact     | table | dbadmin
 store        | store_sales_fact      | table | dbadmin
(15 rows)
```

This example returns information on the `inventory_fact` table in the VMart database:

```
VMart=> \x
Expanded display is on.
VMart=> \d inventory_fact
List of Fields by Tables
-[ RECORD 1 ]------------------------------------------
Schema      | public
Table       | inventory_fact
Column      | date_key
Type        | int
Size        | 8
Default     |
Not Null    | t
Primary Key | f
Foreign Key | public.date_dimension(date_key)
-[ RECORD 2 ]------------------------------------------
Schema      | public
Table       | inventory_fact
Column      | product_key
Type        | int

Size        | 8
Default     |
Not Null    | t
Primary Key | f
Foreign Key | public.product_dimension(product_key)
-[ RECORD 3 ]------------------------------------------
Schema      | public
Table       | inventory_fact
Column      | product_version
Type        | int
Size        | 8
Default     |
Not Null    | t
```

```
Primary Key | f
Foreign Key | public.product_dimension(product_version)
-[ RECORD 4 ]-----------------------------------------
Schema      | public
Table       | inventory_fact
Column      | warehouse_key
Type        | int
Size        | 8
Default     |
Not Null    | t
Primary Key | f
Foreign Key | public.warehouse_dimension(warehouse_key)
-[ RECORD 5 ]-----------------------------------------
Schema      | public
Table       | inventory_fact
Column      | qty_in_stock
Type        | int
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
```

Use the question mark [?] argument to replace a single character. For example, the ? argument replaces the last character in the user-created SubQ1 and SubQ2 tables, so the command returns information about both:

```
=> \d SubQ?

                                List of Fields by Tables
Schema | Table | Column | Type | Size | Default | Not Null | Primary Key | Foreign Key
--------+-------+--------+------+------+---------+----------+-------------+-------------
 public | SubQ1 | a      | int  |    8 |         | f        | f           |
 public | SubQ1 | b      | int  |    8 |         | f        | f           |
 public | SubQ1 | c      | int  |    8 |         | f        | f           |
 public | SubQ2 | x      | int  |    8 |         | f        | f           |
 public | SubQ2 | y      | int  |    8 |         | f        | f           |
 public | SubQ2 | z      | int  |    8 |         | f        | f           |
(6 rows)
```

If you run the \d command and provide both the schema and table name, output includes columns for tables that match the pattern

```
VMart=> \x
Expanded display is on.
VMart=> \d v_catalog.types
List of Fields by Tables
-[ RECORD 1 ]--------------------
Schema      | v_catalog
Table       | types
Column      | column_size
Type        | int
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 2 ]--------------------
Schema      | v_catalog
Table       | types
Column      | creation_parameters
Type        | varchar(128)
Size        | 128
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 3 ]--------------------
Schema      | v_catalog
Table       | types
Column      | epoch
Type        | int
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 4 ]--------------------
Schema      | v_catalog
Table       | types
Column      | interval_mask
Type        | int
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 5 ]--------------------
Schema      | v_catalog
Table       | types
Column      | max_scale
Type        | int
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 6 ]--------------------
Schema      | v_catalog
Table       | types
Column      | min_scale
Type        | int
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 7 ]--------------------
Schema      | v_catalog
Table       | types
Column      | odbc_subtype
Type        | int
```

```
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 8 ]--------------------
Schema      | v_catalog
Table       | types
Column      | odbc_type
Type        | int
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 9 ]--------------------
Schema      | v_catalog
Table       | types
Column      | type_id
Type        | int
Size        | 8
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
-[ RECORD 10 ]-------------------
Schema      | v_catalog
Table       | types
Column      | type_name
Type        | varchar(128)
Size        | 128
Default     |
Not Null    | f
Primary Key | f
Foreign Key |
```

To view all tables in a schema, use the wildcard character. The following command would return all system tables in the `V_CATALOG` schema:

```
=> \d v_catalog.*
```

Df [PATTERN]

The `\df [ PATTERN ]` meta-command returns all function names, the function return data type, and the function argument data type. Also returns the procedure names and arguments for all procedures that are available to the user.

```
vmartdb=> \df
                          List of functions
  procedure_name | procedure_return_type | procedure_argument_types
-----------------+-----------------------+---------------------------
 abs             | Float                 | Float
 abs             | Integer               | Integer
 abs            | Interval              | Interval
 abs            | Interval              | Interval
 abs            | Numeric               | Numeric
 acos           | Float                 | Float
 add_location   | Varchar               | Varchar
 add_location   | Varchar               | Varchar, Varchar, Varchar
...
 width_bucket   | Integer               | Float, Float, Float, Integer
 width_bucket   | Integer               | Interval, Interval, Interval, Integer
 width_bucket   | Integer               | Interval, Interval, Interval, Integer
 width_bucket   | Integer               | Timestamp, Timestamp, Timestamp, Integer
...
```

The following example uses the wildcard character to search for all functions that begin with as:

```
vmartdb=> \df as*
                         List of functions
 procedure_name | procedure_return_type | procedure_argument_types
----------------+-----------------------+--------------------------
 ascii          | Integer               | Varchar
 asin           | Float                 | Float
(2 rows)
```

Dj [PATTERN]

The `\dj [ PATTERN ]` meta-command returns all projections showing the schema, projection name, owner, and node:

```
vmartdb=> \dj
                               List of projections
    Schema    |             Name              |  Owner  |        Node
--------------+-------------------------------+---------+--------------------
 public       | product_dimension_node0001     | dbadmin | v_wmartdb_node0001
 public       | product_dimension_node0002     | dbadmin | v_wmartdb_node0002
 public       | product_dimension_node0003     | dbadmin | v_wmartdb_node0003
 online_sales | call_center_dimension_node0001 | dbadmin | v_wmartdb_node0001
 online_sales | call_center_dimension_node0002 | dbadmin | v_wmartdb_node0002
 online_sales | call_center_dimension_node0003 | dbadmin | v_wmartdb_node0003
...
```

If you supply a projection name as an argument, the system returns fewer records:

```
vmartdb=> \dj call_center_dimension_n*
                             List of projections
    Schema    |              Name             |  Owner  |        Node
--------------+-------------------------------+---------+--------------------
 online_sales | call_center_dimension_node0001 | dbadmin | v_wmartdb_node0001
 online_sales | call_center_dimension_node0002 | dbadmin | v_wmartdb_node0002
 online_sales | call_center_dimension_node0003 | dbadmin | v_wmartdb_node0003
(3 rows)
```

Dn [PATTERN]

The `\dn [ PATTERN ]` meta-command returns the schema names and schema owner.

```
vmartdb=> \dn
    List of schemas
     Name     |  Owner
--------------+---------
 v_internal   | dbadmin
 v_catalog    | dbadmin
 v_monitor    | dbadmin
 public       | dbadmin
 store        | dbadmin
 online_sales | dbadmin
(6 rows)
```

The following command returns all schemas that begin with the letter v:

```
=> \dn v*
   List of schemas
    Name    |  Owner
------------+---------
 v_internal | dbadmin
 v_catalog  | dbadmin
 v_monitor  | dbadmin
(3 rows)
```

Dp [PATTERN]

The `\dp [ PATTERN ]` meta-command returns the grantee, grantor, privileges, schema, and name for all table access privileges in each schema:

```
vmartdb=> \dp
       Access privileges for database "vmartdb"
 Grantee | Grantor | Privileges | Schema |    Name
---------+---------+------------+--------+------------
         | dbadmin | USAGE      |        | public
         | dbadmin | USAGE      |        | v_internal
         | dbadmin | USAGE      |        | v_catalog
         | dbadmin | USAGE      |        | v_monitor
(4 rows)
```

Note: \dp is the same as `\z` .

ds [PATTERN]

The `\ds [ PATTERN ]` meta-command (lowercase s) returns a list of sequences and their parameters.

The following series of commands creates a sequence called my_seq and uses the vsql command to display its parameters:

```
=> CREATE SEQUENCE my_seq MAXVALUE 5000 START 150;
CREATE SEQUENCE

=> \ds
                                List of Sequences
 Schema | Sequence | CurrentValue | IncrementBy | Minimum | Maximum | AllowCycle
--------+----------+--------------+-------------+---------+---------+------------
 public | my_seq   |          149 |           1 |       1 |    5000 | f
(1 row)
```

Note: You can return additional information about sequences by issuing SELECT * FROM SEQUENCES, as described in the SQL Reference Manual.

dS [PATTERN]

The `\dS [ PATTERN ]` meta-command (uppercase S) returns all system table (monitoring API) names from the `V_CATALOG` and `V_MONITOR` schemas.

Tip: You can get identical results running this query: `SELECT * FROM system_tables;`

If you specify a schema name, you can view results for tables in that schema only; however, you must use the wildcard character. For example:

```
=> \dS v_catalog.*
```

You can also run the `\dS` command with a table argument to return information for that table only:

```
vmartdb=> \dS columns
                        List of tables
  Schema   |  Name   |  Kind  |       Description        | Comment
-----------+---------+--------+--------------------------+---------
 v_catalog | columns | system | Table column information |
(1 row)
```

If you provide the schema name with the table name, the output returns information about the table:

dt [PATTERN]

The `\dt [ PATTERN ]` meta-command (lowercase t) is identical to \d and returns all tables in the database—unless a table name is specified—in which case the command lists only the schema, name, kind and owner for the specified table (or tables if wildcards used).

```
vmartdb=> \dt inventory_fact
```

```
             List of tables
 Schema |      Name      | Kind  |  Owner
--------+----------------+-------+---------
 public | inventory_fact | table | dbadmin
(1 row)
```

The following command returns all table names that begin with "`st`":

```
vmartdb=> \dt st*

              List of tables
 Schema |       Name        | Kind  |  Owner
--------+-------------------+-------+---------
 store  | store_dimension   | table | dbadmin
 store  | store_orders_fact | table | dbadmin
 store  | store_sales_fact  | table | dbadmin

(3 rows)
```

dT [PATTERN]

The `\dT [ PATTERN ]` meta-command (uppercase T) lists all supported data types.

```
vmartdb=> \dT
List of data types
  type_name
-------------
 Binary
 Boolean
 Char
 Date
 Float
 Integer
 Interval Day
 Interval Day to Hour
 Interval Day to Minute
 Interval Day to Second
 Interval Hour
 Interval Hour to Minute
 Interval Hour to Second
 Interval Minute
 Interval Minute to Second
 Interval Month
 Interval Second
 Interval Year
 Interval Year to Month
 Numeric
 Time
 TimeTz
 Timestamp
 TimestampTz
 Varbinary
 Varchar
(26 rows)
```

Dtv [PATTERN]

The `\dtv [ PATTERN ]` meta-command lists all tables and views, returning the schema, table or view name, kind (table of view), and owner.

```
vmartdb=> \dtv
                     List of tables
    Schema    |         Name          | Kind  |  Owner
--------------+-----------------------+-------+---------
 online_sales | call_center_dimension | table | release
 online_sales | online_page_dimension | table | release
 online_sales | online_sales_fact     | table | release
 public       | customer_dimension    | table | release
 public       | date_dimension        | table | release
 public       | employee_dimension    | table | release
 public       | inventory_fact        | table | release
 public       | product_dimension     | table | release
 public       | promotion_dimension   | table | release
 public       | shipping_dimension    | table | release
 public       | vendor_dimension      | table | release
 public       | warehouse_dimension   | table | release
 store        | store_dimension       | table | release
 store        | store_orders_fact     | table | release
 store        | store_sales_fact      | table | release
(15 rows)
```

Du [PATTERN]

The `\du [ PATTERN ]` meta-command returns all database users and attributes, such as if user is a **superuser**.

```
vmartdb=> \du
      List of users
 User name | Is Superuser
-----------+--------------
 dbadmin   | t
(1 row)
```

Dv [PATTERN]

The `\dv [ PATTERN ]` meta-command returns the schema name, view name, and view owner.

The following example defines a view using the SEQUENCES system table:

```
vmartdb=> CREATE VIEW my_seqview AS (SELECT * FROM sequences);
CREATE VIEW
```

```
vmartdb=> \dv
         List of views
 Schema |    Name    |  Owner
--------+------------+---------
 public | my_seqview | dbadmin
(1 row)
```

If a view name is provided as an argument, the result shows the schema, view name, and the following for all columns within the view's result set: schema name, view name, column name, column data type, and data type size.

```
vmartdb=> \dv my_seqview
                      List of View Fields
 Schema |    View    |       Column        |     Type     | Size
--------+------------+---------------------+--------------+------
 public | my_seqview | sequence_schema     | varchar(128) |  128
 public | my_seqview | sequence_name       | varchar(128) |  128
 public | my_seqview | owner_name          | varchar(128) |  128
 public | my_seqview | identity_table_name | varchar(128) |  128
 public | my_seqview | session_cache_count | int          |    8
 public | my_seqview | allow_cycle         | boolean      |    1
 public | my_seqview | output_ordered      | boolean      |    1
 public | my_seqview | increment_by        | int          |    8
 public | my_seqview | minimum             | int          |    8
 public | my_seqview | maximum             | int          |    8
 public | my_seqview | current_value       | int          |    8
 public | my_seqview | sequence_schema_id  | int          |    8
 public | my_seqview | sequence_id         | int          |    8
 public | my_seqview | owner_id            | int          |    8
 public | my_seqview | identity_table_id   | int          |    8
(15 rows)
```

e \edit [FILE]

`\e \edit [ FILE ]` edits the query buffer (or specified file) with an external editor. When the editor exits, its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of vsql, where the whole buffer up to the first semicolon is treated as a single line. (Thus you cannot make scripts this way. Use `\i` for that.) If there is no semicolon, vsql waits for one to be entered (it does not execute the query buffer).

Tip: vsql searches the environment variables VSQL_EDITOR, EDITOR, and VISUAL (in that order) for an editor to use. If all of them are unset, vi is used on Linux systems, notepad.exe on Windows systems.

echo [STRING]

`\echo [ STRING ]` writes the string to standard output

Tip: If you use the \o command to redirect your query output you might want to use \qecho instead of this command.

f [String]

`\f [ string ]` sets the field separator for unaligned query output. The default is the vertical bar (|). See also `\pset` for a generic way of setting output options.

g

The `\g` meta-command sends the query in the input buffer (see \p) to the server. With no arguments, it displays the results in the standard way.

`\g FILE` sends the query input buffer to the server, and writes the results to FILE.

`\g | COMMAND` sends the query buffer to the server, and pipes the results to a shell COMMAND.

See Also

- o

H

`\H` toggles HTML query output format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

h \help

`\h \help` displays help information about the meta-commands. Works the same as \? .

```
=> \h
General
   \c[onnect] [DBNAME|- [USER]]
                  connect to new database (currently "VMart")
   \cd [DIR]      change the current working directory
   \q             quit vsql
   \set [NAME [VALUE]]
                  set internal variable, or list all if no parameters
   \timing        toggle timing of commands (currently off)
   \unset NAME    unset (delete) internal variable
   \! [COMMAND]   execute command in shell or start interactive shell
   \password [USER]
                  change user's password
Query Buffer
   \e [FILE]      edit the query buffer (or file) with external editor
   \g             send query buffer to server
   \g FILE        send query buffer to server and results to file
   \g | COMMAND   send query buffer to server and pipe results to command
   \p             show the contents of the query buffer
   \r             reset (clear) the query buffer
   \s [FILE]      display history or save it to file
   \w FILE        write query buffer to file
Input/Output
   \echo [STRING] write string to standard output
   \i FILE        execute commands from file
   \o FILE        send all query results to file
   \o | COMMAND   pipe all query results to command
   \o             close query-results file or pipe
   \qecho [STRING]
                  write string to query output stream (see \o)
Informational
   \d [PATTERN]   describe tables (list tables if no argument is supplied)
   \df [PATTERN]  list functions
   \dj [PATTERN]  list projections
   \dn [PATTERN]  list schemas
   \dp [PATTERN]  list table access privileges
   \ds [PATTERN]  list sequences
   \dS [PATTERN]  list system tables
   \dt [PATTERN]  list tables
   \dtv [PATTERN] list tables and views
   \dT [PATTERN]  list data types
   \du [PATTERN]  list users
   \dv [PATTERN]  list views
   \l             list all databases
   \z [PATTERN]   list table access privileges (same as \dp)
Formatting
   \a             toggle between unaligned and aligned output mode
   \b             toggle beep on command completion
   \C [STRING]    set table title, or unset if none
   \f [STRING]    show or set field separator for unaligned query output
   \H             toggle HTML output mode (currently off)
   \pset NAME [VALUE]
                  set table output option
                  (NAME := {format|border|expanded|fieldsep|footer|null|
                  recordsep|tuples_only|title|tableattr|pager})
   \t             show only rows (currently off)
   \T [STRING]    set HTML <table> tag attributes, or unset if none
   \x             toggle expanded output (currently off)
```

i FILE

`\i filename` command reads input from the file *filename* and executes it as though it had been typed on the keyboard.

Note: To see the lines on the screen as they are read, set the variable ECHO to all.

l

`\l` provides a list of databases and their owners.

```
vmartdb=> \l
  List of databases
  name   | user_name
---------+-----------
 vmartdb | dbadmin
(1 row)
```

Locale

The vsql `\locale` command displays the current locale setting or lets you set a new locale for the session.

This command does not alter the default locale for all database sessions. To change the default for all sessions, set the `DefaultSessionLocale` configuration parameter.

Viewing the Current Locale Setting

To view the current locale setting, use the vsql command \locale, as follows:

```
=> \locale
en_US@collation=binary
```

Overriding the Default Local for a Session

To override the default local for a specific session, use the vsql command \locale <ICU-locale-identifier>. The session locale setting applies to any subsequent commands issued in the session.

For example:

```
\locale en_GBINFO:  Locale: 'en_GB'
INFO:    English (United Kingdom)
INFO:  Short form: 'LEN'
```

You can also use the short form of an ICU locale identifier:

```
\locale LENINFO:  Locale: 'en'
INFO:    English
INFO:  Short form: 'LEN'
```

Notes

The server locale settings impact only the collation behavior for server-side query processing. The client application is responsible for ensuring that the correct locale is set in order to display the characters correctly. Below are the best practices recommended by HP to ensure predictable results:

- The locale setting in the terminal emulator for vsql (POSIX) should be set to be equivalent to session locale setting on server side (ICU) so data is collated correctly on the server and displayed correctly on the client.

- The vsql locale should be set using the POSIX LANG environment variable in terminal emulator. Refer to the documentation of your terminal emulator for how to set locale.
- Server session locale should be set using the set as described in Specify the Default Locale for the Database.
- Note that all input data for vsql should be in UTF-8 and all output data is encoded in UTF-8
- Non UTF-8 encodings and associated locale values are not supported.

o

The `\o` meta-command is used to control where **vsql** directs its query output. The output can be written to a file, piped to a shell command, or sent to the standard output.

`\o FILE` sends all subsequent query output to FILE.

`\o | COMMAND` pipes all subsequent query output to a shell COMMAND.

`\o` with no argument closes any open file or pipe, and switches back to normal query result output.

Notes

- Query results includes all tables, command responses, and notices obtained from the database server.
- To intersperse text output with query results, use \qecho.

See Also

- g

p

`\p` prints the current query buffer to the standard output. For example:

```
=> \p
CREATE VIEW my_seqview AS (SELECT * FROM sequences);
```

Password [USER]

`\password` starts the password change process. Users can only change their own passwords. The command prompts the user for the old password, a new password, and then the new password again to confirm.

A superuser can change the password of another user by supplying the username. A superuser is not prompted for the old password, either when changing his or her own password, or when changing another user's password.

Note: If you want to cancel the password change process, press ENTER until you return the to vsql prompt.

pset NAME [VALUE]

`\pset NAME [ VALUE ]` sets options affecting the output of query result tables. NAME describes which option to set, as illustrated in the following table. The parameters of VALUE depend thereon.

It is an error to call `\pset` without arguments

Adjustable printing options are:

`format`	Sets the output format to one of `unaligned`, `aligned`, `html`, or `latex`. Unique abbreviations are allowed. (That would mean one letter is enough.) "Unaligned" writes all columns of a row on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs (tab- separated, comma-separated). "Aligned" mode is the standard, human-readable, nicely formatted text output that is default. The "HTML" and "LaTeX" modes put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)
`border`	The second argument must be a number. In general, the higher the number the more borders and lines the tables have, but this depends on the particular format. In HTML mode, this translates directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.
`expanded`	Toggles between regular and expanded format. When expanded format is enabled, all output has two columns with the column name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode. Expanded mode is supported by all four output formats. `\x` is the same as `\pset expanded`.

`fieldsep`	Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a vertical bar).
`footer`	Toggles the display of the default footer `(x rows)`.
`null`	The second argument is a string that is printed whenever a column is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null '(null)'`.
`recordsep`	Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.
`tuples_only (or t)`	Toggles between tuples only and full display. Full display might show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.
`title [ text ]`	Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.
`tableattr (or T)[ text ]`	Allows you to specify any attributes to be placed inside the HTML `table` tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`.
`pager`	Controls use of a pager for query and vsql help output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as `more`) is used. When the pager is off, the pager is not used. When the pager is on, the pager is used only when appropriate; that is, the output is to a terminal and does not fit on the screen. (vsql does not do a perfect job of estimating when to use the pager.) `\pset pager` turns the pager on and off. Pager can also be set to `always`, which causes the pager to be always used.

See illustrations on how these different formats look in the Examples section.

Tip: There are various shortcut commands for \pset. See \a , \C , \H , \t , \T , and \ x .

q

`\q` quits the vsql program.

Qecho [STRING]

`\qecho [ STRING ]` is identical to `\echo` except that the output is written to the query output stream, as set by \o .

r

`\r` resets (clears) the query buffer.

For example, run the \p meta-command to see what is in the query buffer:

```
=> \p
CREATE VIEW my_seqview AS (SELECT * FROM sequences);
```

Now reset the query buffer:

```
=> \r
Query buffer reset (cleared).
```

If you reissue the command to see what's in the query buffer, you can see it is now empty:

```
=> \p
Query buffer is empty.
```

s [FILE]

`\s [ FILE ]` prints or saves the command line history to *filename*. If a filename is not specified, \s writes the history to the standard output. This option is only available if vsql is configured to use the GNU Readline library.

set [NAME [VALUE [...]]]

`\set [ name [ value [ ... ] ] ]` sets the internal variable *name* to *value* or, if more than one value is given, to the concatenation of all of values. If no second argument is given, the variable is set with no value.

If no argument is provided, \set lists all internal variables; for example:

```
=> \set
VERSION = 'Vertica Analytic Database v6.0.0-0'
AUTOCOMMIT = 'off'
VERBOSITY = 'default'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
ROWS_AT_A_TIME = '1000'
DBNAME = 'VMartDB'
USER = 'dbadmin'
```

```
HOST = '<host_ip_address>'
PORT = '5433'
LOCALE = 'en_US@collation=binary'
HISTSIZE = '500'
```

Notes

- Valid variable names are case sensitive and can contain characters, digits, and underscores. vsql treats several variables as special, which are described in Variables.
- The `\set` parameter `ROWS_AT_A_TIME` defaults to 1000. It retrieves results as blocks of rows of that size. The column formatting for the first block is used for all blocks, so in later blocks some entries could overflow. See \timing for examples.
- When formatting results, HP Vertica buffers ROWS_AT_A_TIME rows in memory to calculate the maximum column widths. It is possible that rows after this initial fetch are not properly aligned if any of the field values are longer than those see in the first ROWS_AT_A_TIME rows. ROWS_AT_A_TIME can be \unset to guarantee perfect alignment, but this requires re-buffering the entire result set in memory and may cause vsql to fail if the result set is too big.
- To unset a variable, use the \unset command.

Using Backquotes to Read System Variables

In vsql, the contents of backquotes are passed to the system shell to be interpreted (the same behavior as many UNIX shells). This is particularly useful in setting internal vsql variables, since you may want to access UNIX system variables (such as HOME or TMPDIR) rather than hard-code values.

For example, if you want to set an internal variable to the full path for a file in your UNIX user directory, you could use backquotes to get the content of the system HOME variable, which is the full path to your user directory:

```
=> \set inputfile `echo $HOME`/myinput.txt=> \echo :inputfile
/home/dbadmin/myinput.txt
```

The contents of the backquotes are replaced with the results of running the contents in a system shell interpreter. In this case, the `echo $HOME` command returns the contents of the HOME system variable.

t

`\t` toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

T [STRING]

`\T [ STRING ]` specifies attributes to be placed within the `table` tag in HTML tabular output mode. This command is equivalent to `\pset tableattr` *table_options*.

Timing

`\timing` toggles the timing of commands (currently off). The meta-command displays how long each SQL statement takes, in milliseconds, and reports both the time required to fetch the first block of rows from the server and the total until the last block is formatted.

Example

```
=> \o /dev/null=> SELECT * FROM fact LIMIT 100000;
Time: First fetch (1000 rows): 22.054 ms. All rows formatted: 235.056 ms
```

Note that the database retrieved the first 1000 rows in 22 ms and completed retrieving and formatting all rows in 235 ms.

See Also

- set [NAME [VALUE [...]]]

Unset [NAME]

`\unset [ NAME ]` unsets (deletes) the internal variable *name* that was set using the \set meta-command.

w [FILE]

`\w [ FILE ]` outputs the current query buffer to the file *filename.*

x

\x toggles extended table formatting mode. Is equivalent to `\pset expanded`.

Note: There is no space between the backslash and the x.

z

\z lists table access privileges (grantee, grantor, privilege, and name) for all table access privileges in each schema. Is the same as \dp

Variables

vsql provides variable substitution features similar to common Linux command shells. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the vsql meta-command `\set`:

```
=> \set fact dim
```

sets the variable `fact` to the value `dim`. To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
=> \echo :fact dim
```

Note: The arguments of \set are subject to the same substitution rules as with other commands. For example, \set dim :fact is a valid way to copy a variable.

If you call `\set` without a second argument, the variable is set, with an empty string as value. To unset (or delete) a variable, use the command `\unset`.

vsql's internal variable names can consist of letters, numbers, and underscores in any order and any number. Some of these variables are treated specially by vsql. They indicate certain option settings that can be changed at run time by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes.

SQL Interpolation

An additional useful feature of vsql variables is that you can substitute ("interpolate") them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (`:`).

```
=> \set fact 'my_table'
=> SELECT * FROM :fact;
```

would then query the table `my_table`. The value of the variable is copied literally (except for backquoted strings, see below), so it can even contain unbalanced quotes or backslash commands. Make sure that it makes sense where you put it. Variable interpolation is not performed into quoted SQL entities.

Note: The one exception to variable values being copied literally is strings in backquotes (``). The contents of backquoted strings are passed to a system shell, and replaced with the shell's output. See the set metacommand topic for details.

AUTOCOMMIT

When AUTOCOMMIT is set 'on', each SQL command is automatically committed upon successful completion; for example:

```
\ set  AUTOCOMMIT on
```

To postpone COMMIT in this mode, set the value as off.

```
\set AUTOCOMMIT off
```

If AUTOCOMMIT is empty or defined as off, SQL commands are not committed unless you explicitly issue COMMIT.

Notes

- AUTOCOMMIT is off by default.
- AUTOCOMMIT must be in uppercase, but the values, on or off, are case insensitive.
- In autocommit-off mode, you must explicitly abandon any failed transaction by entering ABORT or ROLLBACK.
- If you exit the session without committing, your work is rolled back.
- Validation on vsql variables is done when they are run, not when they are set.
- The COPY statement, by default, commits on completion, so it does not matter which AUTOCOMMIT mode you use, unless you issue COPY NO COMMIT.
- To tell if AUTOCOMMIT is on or off, issue the set command:

  ```
  $ \set...
  AUTOCOMMIT = 'off'
  ...
  ```

- AUTOCOMMIT is off if a `SELECT * FROM LOCKS` shows locks from the statement you just ran.

  ```
  $ \set AUTOCOMMIT off
  $ \set
  ...
  AUTOCOMMIT = 'off'
  ...
  SELECT COUNT(*) FROM customer_dimension;
   count
  -------
   50000
  ```

```
(1 row)
SELECT node_names, object_name, lock_mode, lock_scope
FROM LOCKS;
 node_names |      object_name        | lock_mode | lock_scope
------------+-------------------------+-----------+-------------
 site01     | Table:customer_dimension | S         | TRANSACTION
(1 row)
```

DBNAME

The name of the database to which you are currently connected. DBNAME is set every time you connect to a database (including program startup), but it can be unset.

ECHO

If set to `all`, all lines entered from the keyboard or from a script are written to the standard output before they are parsed or run.

To select this behavior on program start-up, use the switch `-a` . If set to `queries`, vsql merely prints all queries as they are sent to the server. The switch for this is `-e` .

ECHO_HIDDEN

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the HP Vertica internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch `-E` .)

If you set the variable to the value `noexec`, the queries are just shown but are not actually sent to the server and run.

ENCODING

The current client character set encoding.

HISTCONTROL

If this variable is set to `ignorespace`, lines that begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to any other value than those previously mentioned, all lines read in interactive mode are saved on the history list.

Source: Bash.

HISTSIZE

The number of commands to store in the command history. The default value is 500.

Source: Bash.

HOST

The database server host you are currently connected to. This is set every time you connect to a database (including program startup), but can be unset.

IGNOREEOF

If unset, sending an EOF character (usually Control+D) to an interactive session of vsql terminates the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

Source: Bash.

ON_ERROR_STOP

By default, if a script command results in an error, for example, because of a malformed command or invalid data format, processing continues. If you set `ON_ERROR_STOP` to `‘on’` in a script and an error occurs during processing, the script terminates immediately.

If you set `ON_ERROR_STOP` to `‘on’` in a script, run the script from Linux using `vsql -f <`*`filename`*`>`, and an error occurs, vsql returns an error code 3 to Linux to indicate that the error occurred in a script.

To enable `ON_ERROR_STOP`:

```
=> \set ON_ERROR_STOP on
```

To disable `ON_ERROR_STOP`:

```
=> \set ON_ERROR_STOP off
```

PORT

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

PROMPT1 PROMPT2 PROMPT3

These specify what the prompts vsql issues look like. See Prompting for details.

QUIET

This variable is equivalent to the command line option `-q`. It is probably not too useful in interactive mode.

SINGLELINE

This variable is equivalent to the command line option `-S`.

SINGLESTEP

This variable is equivalent to the command line option `-s`.

USER

The database user you are currently connected as. This is set every time you connect to a database (including program startup), but can be unset.

VERBOSITY

This variable can be set to the values `default`, `verbose`, or `terse` to control the verbosity of error reports.

VSQL_HOME

By default, the vsql program reads configuration files from the user's home directory. In cases where this is not desirable, the configuration file location can be overridden by setting the VSQL_ HOME environment variable in a way that does not require modifying a shared resource.

In the following example, vsql reads configuration information out of /tmp/jsmith rather than out of ~.

```
# Make an alternate configuration file in /tmp/jsmith
mkdir -p /tmp/jsmith
echo "\\echo Using VSQLRC in tmp/jsmith" > /tmp/jsmith/.vsqlrc
# Note that nothing is echoed when invoked normally
vsql
# Note that the .vsqlrc is read and the following is
# displayed before the vsql prompt
#
# Using VSQLRC in tmp/jsmith
VSQL_HOME=/tmp/jsmith vsql
```

Prompting

The prompts vsql issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when vsql requests a new command. Prompt 2 is issued when more input is expected during command input because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run a SQL `COPY` command and you are expected to type in the row values on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (%) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`	The full host name (with domain name) of the database server, or [local] if the connection is over a socket, or [local:/dir/name], if the socket is not at the compiled in default location.
`%m`	The host name of the database server, truncated at the first dot, or [local].
`%>`	The port number at which the database server is listening.
`%n`	The database session user name.
`%/`	The name of the current database.
`%~`	Like %/, but the output is ~ (tilde) if the database is your default database.
`%#`	If the session user is a database superuser, then a #, otherwise a >. (The expansion of this value might change during a database session as the result of the command SET SESSION AUTHORIZATION.)
`%R`	In prompt 1 normally =, but ^ if in single-line mode, and ! if the session is disconnected from the database (which can happen if \connect fails). In prompt 2 the sequence is replaced by -, *, a single quote, a double quote, or a dollar sign, depending on whether vsql expects more input because the command wasn't terminated yet, because you are inside a /* ... */ comment, or because you are inside a quoted or dollar-escaped string. In prompt 3 the sequence doesn't produce anything.
`%x`	Transaction status: an empty string when not in a transaction block, or * when in a transaction block, or ! when in a failed transaction block, or ? when the transaction state is indeterminate (for example, because there is no connection).
`%digits`	The character with the indicated numeric code is substituted. If digits starts with 0x the rest of the characters are interpreted as hexadecimal; otherwise if the first digit is 0 the digits are interpreted as octal; otherwise the digits are read as a decimal number.
`%:name:`	The value of the vsql variable name. See the section Variables for details.
``%`command` ``	The output of command, similar to ordinary "back- tick" substitution.
`%[ ... %]`	Prompts may contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for the line editing features of Readline to work properly, these non-printing control characters must be designated as invisible by surrounding them with %[and %]. Multiple pairs of these may occur within the prompt. The following example results in a boldfaced (1;) yellow-on-black (33;40) prompt on VT100-compatible, color-capable terminals: `testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%#%[%033[0m%] '` To insert a percent sign into your prompt, write %%. The default prompts are '%/%R%# ' for prompts 1 and 2, and '>> ' for prompt 3. **Note:** See the specification for terminal control sequences (applicable to gnome-terminal and xterm).

Command Line Editing

vsql supports the tecla library for convenient line editing and retrieval.

The command history is automatically saved when vsql exits and is reloaded when vsql starts up. Tab-completion is also supported, although the completion logic makes no claim to be a SQL parser. If for some reason you do not like the tab completion, you can turn it off by putting this in a file named `.teclarc` in your home directory:

```
bind ^I
```

Read the tecla documentation for further details.

Notes

The vsql implementation of the tecla library deviates from the tecla documentation as follows:

- Recalling Previously Typed Lines

  Under pure tecla, all new lines are appended to a list of historical input lines maintained within the GetLine resource object. In vsql, only different, non-empty lines are appended to the list of historical input lines.

- History Files

  tecla has no standard name for the history file. In vsql, the file name is called ~/.vsql_hist.

- International Character Sets (Meta keys and locales)

  In vsql, 8-bit meta characters are no longer supported. Make sure that meta characters send an escape by setting their EightBitInput X resource to False. You can do this in one of the following ways:

  - Edit the ~/.Xdefaults file by adding the following line:

    ```
    XTerm*EightBitInput: False
    ```

  - Start an xterm with an -xrm '`*EightBitInput: False`' command-line argument.

- Key Bindings:

- The following key bindings are specific to vsql:

  - *Insert* switches between insert mode (the default) and overwrite mode.

  - *Delete* deletes the character to the right of the cursor.

  - *Home* moves the cursor to the front of the line.

- *End* moves the cursor to the end of the line.
- ^R Performs a history backwards search.

vsql Environment Variables

The following environment variables can be set to automatically use the defined properties each time you start vsql:

- **PAGER -** If the query results do not fit on the screen, they are piped through this command. Typical values are `more` or `less`. The default is platform-dependent. The use of the pager can be disabled by using the \pset command.
- **VSQL_DATABASE -** Database to connect to. For example, `VMart`.
- **VSQL_HOST -** Host name or IP address of the HP Vertica node.
- **VSQL_PORT -** Port to use for the connection.
- **VSQL_USER -** Username to use for the connection.
- **VSQL_PASSWORD -** Password to use for the connection.
- **VSQL_EDITOR, EDITOR and VISUAL -** Editor used by the \e command. The variables are examined in the order listed; the first that is set is used.
- **SHELL -** Command run by the \! command.
- **TMPDIR -** Directory for storing temporary files. The default is platform-dependant. On Unix-like systems the default is `/tmp`.

Locales

The default terminal emulator under Linux is gnome-terminal, although xterm can also be used.

HP recommends that you use gnome-terminal with **vsql** in UTF-8 mode, which is its default.

To Change Settings on Linux

1. From the tabs at the top of the vsql screen, select Terminal.
2. Click **Set Character Encoding**.
3. Select **Unicode (UTF-8)**.

Note: This works well for standard keyboards. xterm has a similar UTF-8 option.

To Change Settings on Windows Using PuTTy

1. Right click the vsql screen title bar and select **Change Settings**.
2. Click **Window** and click **Translation**.
3. Select **UTF-8** in the drop-down menu on the right.

Notes

- vsql has no way of knowing how you have set your terminal emulator options.
- The tecla library is prepared to do POSIX-type translations from a local encoding to UTF-8 on interactive input, using the POSIX LANG, etc., environment variables. This could be useful to international users who have a non-UTF-8 keyboard. See the tecla documentation for details.

  HP recommends the following (or whatever other .UTF-8 locale setting you find appropriate):

  ```
  export LANG=en_US.UTF-8
  ```

- The vsql \locale command invokes and tracks the server SET LOCALE TO command, described in the SQL Reference Manual. vsql itself currently does nothing with this locale setting, but rather treats its input (from files or from tecla), all its output, and all its interactions with the server as UTF-8. vsql ignores the POSIX locale variables, except for any "automatic" uses in `printf`, and so on.

Files

Before starting up, vsql attempts to read and execute commands from the system-wide `vsqlrc` file and the user's `~/.vsqlrc` file. The command-line history is stored in the file `~/.vsql_history.`

Tip: If you want to save your old history file, open another terminal window and save a copy to a different file name.

Exporting Data Using vsql

You can use **vsql** for simple data exports tasks by changing its output format options so the output is suitable for importing into other systems (tab delimited or comma-separated files, for example). These options can be set either from within an interactive vsql session, or through command-line arguments to the vsql command (making the export process suitable for automation through scripting). After you have set vsql's options so it outputs the data in a format your target system can read, you run a query and capture the result in a text file.

The following table lists the meta-commands and command-line options that are useful for changing the format of vsql's output.

Description	Meta-command	Command-line Option
Disable padding used to align output.	\a	-A or --no-align
Show only tuples, disabling column headings and row counts.	\t	-t or --tuples-only
Set the field separator character.	\pset fieldsep	-F or --field-separator
Send output to a file.	\o	-o or --output
Specify a SQL statement to execute.	N/A	-c or --command

The following example demonstrates disabling padding and column headers in the output, and setting a field separator to dump a table to a tab-separated text file within an interactive session.

```
=> SELECT * FROM my_table;
 a |   b   | c
---+-------+---
 a | one   | 1
 b | two   | 2
 c | three | 3
 d | four  | 4
 e | five  | 5
(5 rows)
=> \a
Output format is unaligned.
=> \t
Showing only tuples.
=> \pset fieldsep '\t'
Field separator is "    ".
=> \o dumpfile.txt
=> select * from my_table;
=> \o
=> \! cat dumpfile.txt
a       one     1
b       two     2
c       three   3
d       four    4
e       five    5
```

Note: You could encounter issues with empty strings being converted to NULLs or the reverse using this technique. You can prevent any confusion by explicitly setting null values to output a unique string such as NULLNULLNULL (for example, `\pset null 'NULLNULLNULL'`). Then, on the import end, convert the unique string back to a null value. For example, if you are copying the file back into an HP Vertica database, you would give the argument `NULL 'NULLNULLNULL'` to the COPY statement.

When logged into one of the database nodes, you can create the same output file directly from the command line by passing the right parameters to vsql:

```
$ vsql -U username -F $'\t' -At -o dumpfile.txt -c "SELECT * FROM my_table;"
```

```
Password:
$ cat dumpfile.txt
a       one     1
b       two     2
c       three   3
d       four    4
e       five    5
```

Note: `$'...'` is a BASH-specific string format that interprets backslash escapes, so it will pass a literal tab character to the vsql command as the argument for the -F parameter. Shells other than BASH may have other string literal syntax.

If you want to convert null values to a unique string as mentioned earlier, you can add the argument `-P null='NULLNULLNULL'` (or whatever unique string you choose).

By adding the `-w` vsql command-line option to the example command line, you could use the command within a batch script to automate the data export. However, the script would contain the database password as plain text. If you take this approach, you should prevent unauthorized access to the batch script, and also have the script use a database user account that has limited access.

Copying Data Using vsql

You can use vsql to copy data between two HP Vertica databases. This technique is similar to the technique explained in Exporting Data Using vsql, except instead of having vsql save data to a file for export, you pipe one vsql's output to the input of another vsql command that runs a COPY statement from STDIN. This technique can also work for other databases or applications that accept data from an input stream.

Note: The following technique only works for individual tables. To copy an entire database to another cluster, see Copying a Database to Another Cluster in the Administrator's Guide.

The easiest way to copy using vsql is to log in to a node of the target database, then issue a vsql command that connects to the source HP Vertica database to dump the data you want. For example, the following command copies the store.store_sales_fact table from the vmart database on node testdb01 to the vmart database on the node you are logged into:

```
vsql -U username -w passwd -h testdb01 -d vmart -At -c "SELECT * from store.store_sales_f
act"  \
| vsql -U username -w passwd -d vmart -c "COPY store.store_sales_fact FROM STDIN DELIMITE
R '|';"
```

Note: The above example copies the data only, not the table design. The target table for the data copy must already exist in the target database. You can export the design of the table using EXPORT_OBJECTS or EXPORT_CATALOG.

Monitoring Progress (optional)

You may want some way of monitoring progress when copying large amounts of data between HP Vertica databases. One way of monitoring the progress of the copy operation is to use a utility such as Pipe Viewer that pipes its input directly to its output while displaying the amount and speed of data it passes along. Pipe Viewer can even display a progress bar if you give it the total number of bytes or lines you expect to be processed. You can get the number of lines to be processed by running a separate vsql command that executes a SELECT COUNT query.

Note: Pipe Viewer isn't a standard Linux or Solaris command, so you will need download and install it yourself. See the Pipe Viewer page for download packages and instructions. HP does not support Pipe Viewer. Install and use it at your own risk.

The following command demonstrates how you can use Pipe Viewer to monitor the progress of the copy shown in the prior example. The command is complicated by the need to get the number of rows that will be copied, which is done using a separate vsql command within a BASH backquote string, which executes the strings contents and inserts the output of the command into the command line. This vsql command just counts the number of rows in the store.store_sales_fact table.

```
vsql -U username -w passwd -h testdb01 -d vmart -At -c "SELECT * from store.store_sales_f
act"  \
| pv -lpetr -s `vsql -U username -w passwd -h testdb01 -d vmart -At -c "SELECT COUNT (*)
FROM store.store_sales_fact;"` \
| vsql -U username -w passwd -d vmart -c "COPY store.store_sales_fact FROM STDIN DELIMITE
R '|';"
```

While running, the above command displays a progress bar that looks like this:

```
0:00:39 [12.6M/s] [==============================>                              ] 50% ETA 0
0:00:40
```

Output Formatting Examples

The first example shows how to spread a command over several lines of input. Notice the changing prompt:

```
=> CREATE TABLE my_table (
-> first integer not null default 0,
-> second char(10));
CREATE TABLE
```

Assume you have filled the table with data and want to take a look at it:

```
testdb=> SELECT * FROM my_table;
 first | second
```

```
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

You can display tables in different ways by using the `\pset` command:

```
testdb=> \pset border 2
 Border style is 2.
testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

=> \pset border 0
 Border style is 0.
=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

=> \pset border 1 Border style is 1.
=> \pset format unaligned
Output format is unaligned.
=> \pset fieldsep ','
Field separator is ",".
=> \pset tuples_only
Showing only tuples.
=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
=> \a \t \ x Output format is aligned.
Tuples only is off.
Expanded display is on.
=> SELECT * FROM my_table;
first  | 1
second | one
-------+-----------
first  | 2
```

```
second | two
-------+-----------
first  | 3
second | three
-------+-----------
first  | 4
second | four
```

Writing Queries

Queries are database operations that retrieve data from one or more tables or views. In HP Vertica, the top-level `SELECT` statement is the query, and a query nested within another SQL statement is called a subquery.

HP Vertica is designed to run the same SQL standard queries that run on other databases. However, there are some differences between HP Vertica queries and queries used in other relational database management systems.

The HP Vertica transaction model is different from the SQL standard in a way that has a profound effect on query performance. You can:

- Run a query on a static snapshot of the database from any specific date and time. Doing so avoids holding locks or blocking other database operations.
- Use a subset of the standard SQL isolation levels and access modes (read/write or read-only) for a user **session**.

In HP Vertica, the primary structure of a SQL query is its statement. Each statement ends with a semicolon, and you can write multiple queries separated by semicolons; for example:

```
=> CREATE TABLE t1( ..., date_col date NOT NULL, ...);
=> CREATE TABLE t2(..., state VARCHAR NOT NULL, ...);
```

Multiple Instances of Dimension Tables in the FROM Clause

The same dimension table can appear multiple times in a query's `FROM` clause, using different aliases. For example:

```
SELECT * FROM   fact, dimension d1, dimension d2
WHERE  fact.fk = d1.pk
       AND
       fact.name = d2.name;
```

Historical (Snapshot) Queries

HP Vertica supports querying historical data for individual `SELECT` statements.

Syntax

```
[ AT EPOCH LATEST ] | [ AT TIME 'timestamp' ] SELECT ...
```

Parameters

`AT EPOCH LATEST`	Queries all committed data in the database up to, but not including, the current **epoch**.
`AT TIME '`*`timestamp`*`'`	Queries all committed data in the database up to the time stamp specified. `AT TIME 'timestamp'` queries are resolved to the next epoch boundary before being evaluated.

Historical queries, also known as **snapshot** queries, are useful because they access data in past epochs only. Historical queries do not need to hold table locks or block write operations because they do not return the absolute latest data.

Historical queries behave in the same manner regardless of transaction isolation level. Historical queries observe only committed data, even excluding updates made by the current transaction, unless those updates are to a temporary table.

Note: You do not need to use historical queries for temporary tables because temp tables do not require locks. Their content is private to the transaction and valid only for the length of the transaction.

Be aware that there is only one snapshot of the **logical schema**. This means that any changes you make to the schema are reflected across all epochs. If, for example, you add a new column to a table and you specify a default value for the column, all historical epochs display the new column and its default value.

See Also

- Transactions

Temporary Tables

You can use the `CREATE TEMPORARY TABLE` statement to implement certain queries using multiple steps:

1. Create one or more temporary tables.
2. Execute queries and store the result sets in the temporary tables.
3. Execute the main query using the temporary tables as if they were a normal part of the **logical schema**.

See `CREATE TEMPORARY TABLE` in the SQL Reference Manual for details.

SQL Queries

All DML (Data Manipulation Language) statements can contain queries. This section introduces some of the query types in HP Vertica, with additional details in later sections.

Note: Many of the examples in this chapter use the VMart schema. For information about other HP Vertica-supplied queries, see the Getting Started Guide.

Simple Queries

Simple queries contain a query against one table. Minimal effort is required to process the following query, which looks for product keys and SKU numbers in the product table:

```
=> SELECT product_key, sku_number FROM public.product_dimension;
product_key  | sku_number
-------------+-----------
43           | SKU-#129
87           | SKU-#250
42           | SKU-#125
49           | SKU-#154
37           | SKU-#107
36           | SKU-#106
86           | SKU-#248
41           | SKU-#121
88           | SKU-#257
40           | SKU-#120
(10 rows)
```

Joins

Joins use a relational operator that combines information from two or more tables. The query's `ON` clause specifies how tables are combined, such as by matching foreign keys to primary keys. In the following example, the query requests the names of stores with transactions greater than 70 by joining the store key ID from the store schema's sales fact and sales tables:

```
=> SELECT store_name, COUNT(*) FROM store.store_sales_fact
   JOIN store.store_dimension ON store.store_sales_fact.store_key = store.store_dimension
.store_key
   GROUP BY store_name HAVING COUNT(*) > 70 ORDER BY store_name;

 store_name | count
------------+-------
 Store49    |    72
 Store83    |    78
(2 rows)
```

For more detailed information, see Joins. See also the Multicolumn subqueries section in Subquery Examples.

Cross Joins

Also known as the Cartesian product, a cross join is the result of joining every record in one table with every record in another table. A cross join occurs when there is no join key between tables to restrict records. The following query, for example, returns all instances of vendor and store names in the vendor and store tables:

```
=> SELECT vendor_name, store_name FROM public.vendor_dimension
    CROSS JOIN store.store_dimension;
vendor_name         | store_name
--------------------+------------
Deal Warehouse      | Store41
Deal Warehouse      | Store12
Deal Warehouse      | Store46
Deal Warehouse      | Store50
Deal Warehouse      | Store15
Deal Warehouse      | Store48
Deal Warehouse      | Store39
Sundry Wholesale    | Store41
Sundry Wholesale    | Store12
Sundry Wholesale    | Store46
Sundry Wholesale    | Store50
Sundry Wholesale    | Store15
Sundry Wholesale    | Store48
Sundry Wholesale    | Store39
Market Discounters  | Store41
Market Discounters  | Store12
Market Discounters  | Store46
Market Discounters  | Store50
Market Discounters  | Store15
Market Discounters  | Store48
Market Discounters  | Store39
Market Suppliers    | Store41
Market Suppliers    | Store12
Market Suppliers    | Store46
Market Suppliers    | Store50
Market Suppliers    | Store15
Market Suppliers    | Store48
Market Suppliers    | Store39
...                 | ...
(4000 rows)
```

This example's output is truncated because this particular cross join returned several thousand rows. See also Cross Joins.

Subqueries

A subquery is a query nested within another query. In the following example, we want a list of all products containing the highest fat content. The inner query (subquery) returns the product containing the highest fat content among all food products to the outer query block (containing query). The outer query then uses that information to return the names of the products containing the highest fat content.

```
=> SELECT product_description, fat_content FROM public.product_dimension
   WHERE fat_content IN
     (SELECT MAX(fat_content) FROM public.product_dimension
      WHERE category_description = 'Food' AND department_description = 'Bakery')
   LIMIT 10;
         product_description         | fat_content
-------------------------------------+-------------
 Brand #59110 hotdog buns            |          90
 Brand #58107 english muffins        |          90
 Brand #57135 english muffins        |          90
 Brand #54870 cinnamon buns          |          90
 Brand #53690 english muffins        |          90
 Brand #53096 bagels                 |          90
 Brand #50678 chocolate chip cookies |          90
 Brand #49269 wheat bread            |          90
 Brand #47156 coffee cake            |          90
 Brand #43844 corn muffins           |          90
(10 rows)
```

For more information, see Subqueries.

Sorting Queries

Use the `ORDER BY` clause to order the rows that a query returns.

Special Note About Query Results

You could get different results running certain queries on one machine or another for the following reasons:

- Partitioning on a `FLOAT` type could return **nondeterministic** results because of the precision, especially when the numbers are close to one another, such as results from the `RADIANS()` function, which has a very small range of output.

  To get **deterministic** results, use `NUMERIC` if you must partition by data that is not an `INTEGER` type.

- Most analytics (with analytic aggregations, such as `MIN()/MAX()/SUM()/COUNT()/AVG()` as exceptions) rely on a unique order of input data to get deterministic result. If the analytic window-order clause cannot resolve ties in the data, results could be different each time you run the query.

  For example, in the following query, the analytic `ORDER BY` does not include the first column in the query, `promotion_key`. So for a tie of `AVG(RADIANS(cost_dollar_amount)), product_version`, the same `promotion_key` could have different positions within the analytic partition, resulting in a different `NTILE()` number. Thus, `DISTINCT` could also have a different result:

  ```
  => SELECT COUNT(*) FROM
        (SELECT DISTINCT SIN(FLOOR(MAX(store.store_sales_fact.promotion_key))),
     NTILE(79) OVER(PARTITION BY AVG (RADIANS
  ```

```
      (store.store_sales_fact.cost_dollar_amount ))
   ORDER BY store.store_sales_fact.product_version)
   FROM store.store_sales_fact
   GROUP BY store.store_sales_fact.product_version,
         store.store_sales_fact.sales_dollar_amount ) AS store;
 count
-------
  1425
(1 row)
```

If you add `MAX(promotion_key)` to analytic `ORDER BY`, the results are the same on any machine:

```
=> SELECT COUNT(*) FROM (SELECT DISTINCT MAX(store.store_sales_fact.promotion_key),
    NTILE(79) OVER(PARTITION BY MAX(store.store_sales_fact.cost_dollar_amount)
   ORDER BY store.store_sales_fact.product_version,
   MAX(store.store_sales_fact.promotion_key))
   FROM store.store_sales_fact
   GROUP BY store.store_sales_fact.product_version,
     store.store_sales_fact.sales_dollar_amount) AS store;
```

Subqueries

Subqueries provide a great deal of flexibility to SQL statements by letting you perform in one step what, otherwise, would require several steps. For example, instead of having to write separate queries to answer multiple-part questions, you can write a subquery.

A subquery is a SELECT statement within another SELECT statement. The inner statement is the subquery, and the outer statement is the containing statement (often referred to in HP Vertica as the outer query block).

Like any query, a subquery returns records from a table that could consist of a single column and record, a single column with multiple records, or multiple columns and records. Queries can be noncorrelated or correlated. You can even use them to update or delete records in a table based on values that are stored in other database tables.

Notes

- Many examples in this section use the VMart example database.
- Be sure to read Subquery Restrictions.

Subqueries Used in Search Conditions

Subqueries are used as search conditions in order to filter results. They specify the conditions for the rows returned from the containing query's select-list, a query expression, or the subquery itself. The operation evaluates to TRUE, FALSE, or UNKNOWN (NULL).

Syntax

```
< search_condition > {
    [ { AND | OR [ NOT ] } { < predicate > | ( < search_condition > ) } ]
   } [ ,... ]
< predicate >
     { expression comparison-operator expression
     ... | string-expression [ NOT ] { LIKE | ILIKE | LIKEB | ILIKEB } string-expression

     ... | expression IS [ NOT ] NULL
     ... | expression [ NOT ] IN ( subquery | expression [ ,...n ] )
     ... | expression comparison-operator [ ANY | SOME ] ( subquery )
     ... | expression comparison-operator ALL ( subquery )

     ... | expression OR ( subquery )
     ... | [ NOT ] EXISTS ( subquery )
     ... | [ NOT ] IN ( subquery )
     }
```

Parameters

<search-condition>	Specifies the search conditions for the rows returned from one of the: • containing query's select-list • a query expression • a subquery If the subquery is used with an UPDATE or DELETE statement, UPDATE specifies the rows to update and DELETE specifies the rows to delete.

`{ AND | OR | NOT }`	Keywords that specify the logical operators that combine conditions, or in the case of NOT, negate conditions. • AND — Combines two conditions and evaluates to TRUE when both of the conditions are TRUE. • OR — Combines two conditions and evaluates to TRUE when either condition is TRUE. • NOT — Negates the Boolean expression specified by the predicate.
<predicate>	Is an expression that returns TRUE, FALSE, or UNKNOWN (NULL).
expression	Can be a column name, a constant, a function, a scalar subquery, or a combination of column names, constants, and functions connected by operators or subqueries.
`comparison-operator`	Test conditions between expressions: • < tests the condition of one expression being less than the other. • > tests the condition of one expression being greater than the other. • <= tests the condition of one expression being less than or equal to the other expression. • >= tests the condition of one expression being greater than or equal to the other expression. • = tests the equality between two expressions. • <=> tests equality like the = operator, but it returns TRUE instead of UNKNOWN if both operands are UNKNOWN and FALSE instead of UNKNOWN if one operand is UNKNOWN. • <> and != test the condition of two expressions not equal to one another.
string_expression	Is a character string with optional wildcard (*) characters.

`[ NOT ] { LIKE | ILIKE | LIKEB | ILIKEB }`	Indicates that the character string following the predicate is to be used (or not used) for pattern matching.
`IS [ NOT ] NULL`	Searches for values that are null or are not null.
`ALL`	Is used with a comparison operator and a subquery. Returns TRUE for the lefthand predicate if all values returned by the subquery satisfy the comparison operation, or FALSE if not all values satisfy the comparison or if the subquery returns no rows to the outer query block.
`ANY | SOME`	ANY and SOME are synonyms and are used with a comparison operator and a subquery. Either returns TRUE for the lefthand predicate if any value returned by the subquery satisfies the comparison operation, or FALSE if no values in the subquery satisfy the comparison or if the subquery returns no rows to the outer query block. Otherwise, the expression is UNKNOWN.
`[ NOT ] EXISTS`	Used with a subquery to test for the existence of records that the subquery returns.
`[ NOT ] IN`	Searches for an expression on the basis of an expression's exclusion or inclusion from a list. The list of values is enclosed in parentheses and can be a subquery or a set of constants.

Logical Operators AND and OR

The AND and OR logical operators combine two conditions. AND evaluates to TRUE when both of the conditions joined by the AND keyword are matched, and OR evaluates to TRUE when either condition joined by OR is matched.

OR Subqueries (complex expressions)

HP Vertica supports subqueries in more complex expressions using OR; for example:

- More than one subquery in the conjunct expression:

```
(SELECT MAX(b) FROM t1) + SELECT (MAX FROM t2) a IN (SELECT a FROM t1) OR b IN (SELECT
x FROM t2)
```

- An OR clause in the conjunct expression involves at least one subquery:

```
a IN (SELECT a FROM t1) OR b IN (SELECT x FROM t2) a IN (SELECT a from t1) OR b = 5
a = (SELECT MAX FROM t2) OR b = 5
```

- One subquery is present but it is part of a another expression:

```
x IN (SELECT a FROM t1) = (x = (SELECT MAX FROM t2) (x IN (SELECT a FROM t1) IS NULL
```

How AND Queries Are Evaluated

HP Vertica treats expressions separated by AND (conjunctive) operators individually. For example if the WHERE clause were:

```
  WHERE (a IN (SELECT a FROM t1) OR b IN (SELECT x FROM t2)) AND (c IN (SELECT a FROM t
1))
```

the query would be interpreted as two conjunct expressions:

1. `(a IN (SELECT a FROM t1) OR b IN (SELECT x FROM t2))`
2. `(c IN (SELECT a FROM t1))`

The first expression is considered a complex subquery, whereas the second expression is not.

Examples

The following list shows some of the ways you can filter complex conditions in the WHERE clause:

- OR expression between a subquery and a non-subquery condition:

```
=> SELECT x FROM t WHERE x > (SELECT SUM(DISTINCT x) FROM t GROUP BY y) OR x < 9;
```

- OR expression between two subqueries:

```
=> SELECT * FROM t WHERE x=(SELECT x FROM t) OR EXISTS(SELECT x FROM tt);
```

- Subquery expression:

```
=> SELECT * FROM t WHERE x=(SELECT x FROM t)+1 OR x<>(SELECT x FROM t)+1;
```

- OR expression with [NOT] IN subqueries:

```
=> SELECT * FROM t WHERE NOT (EXISTS (SELECT x FROM t)) OR x >9;
```

- OR expression with IS [NOT] NULL subqueries:

```
=> SELECT * FROM t WHERE (SELECT * FROM t)IS NULL OR (SELECT * FROM tt)IS NULL;
```

- OR expression with boolean column and subquery that returns Boolean data type:

```
=> SELECT * FROM t2 WHERE x = (SELECT x FROM t2) OR x;
```

Note: To return TRUE, the argument of OR must be a Boolean data type.

- OR expression in the CASE statement:

```
=> SELECT * FROM t WHERE CASE WHEN x=1 THEN x > (SELECT * FROM t)
       OR x < (SELECT * FROM t2) END ;
```

- Analytic function, NULL-handling function, string function, math function, and so on:

```
=> SELECT x FROM t WHERE x > (SELECT COALESCE (x,y) FROM t GROUP BY x,y) OR
       x < 9;
```

- In user-defined functions (assuming `f()` is one):

```
=> SELECT * FROM t WHERE x > 5 OR x = (SELECT f(x) FROM t);
```

- Use of parentheses at different places to restructure the queries:

```
=> SELECT x FROM t WHERE (x = (SELECT x FROM t) AND y = (SELECT y FROM t))
       OR (SELECT x FROM t) =1;
```

- Multicolumn subqueries:

```
=> SELECT * FROM t WHERE (x,y) = (SELECT x,y FROM t) OR x > 5;
```

- Constant/NULL on lefthand side of subquery:

```
=> SELECT * FROM t WHERE x > 5 OR 5 = (SELECT x FROM t);
```

See Also

- Subquery Restrictions

In Place of an Expression

Subqueries that return a single value (unlike a list of values returned by IN subqueries) can be used just about anywhere an expression is allowed in SQL. It can be a column name, a constant, a function, a scalar subquery, or a combination of column names, constants, and functions connected by operators or subqueries.

For example:

```
=> SELECT c1 FROM t1 WHERE c1 = ANY (SELECT c1 FROM t2) ORDER BY c1;
=> SELECT c1 FROM t1 WHERE COALESCE((t1.c1 > ANY (SELECT c1 FROM t2)), TRUE);
=> SELECT c1 FROM t1 GROUP BY c1 HAVING
     COALESCE((t1.c1 <> ALL (SELECT c1 FROM t2)), TRUE);
```

Multi-column expressions are also supported:

```
=> SELECT c1 FROM t1 WHERE (t1.c1, t1.c2) = ALL (SELECT c1, c2 FROM t2);
=> SELECT c1 FROM t1 WHERE (t1.c1, t1.c2) <> ANY (SELECT c1, c2 FROM t2);
```

HP Vertica returns an error on queries where more than one row would be returned by any subquery used as an expression:

```
=> SELECT c1 FROM t1 WHERE c1 = (SELECT c1 FROM t2) ORDER BY c1;
   ERROR:  more than one row returned by a subquery used as an expression
```

See Also

- Subquery Restrictions

Comparison Operators

HP Vertica supports Boolean subquery expressions in the WHERE clause with any of the following operators: (>, <, >=, <=, =, <>, <=>).

WHERE clause subqueries filter results and take the following form:

```
SELECT <column, ...> FROM <table>
WHERE <condition> (SELECT <column, ...> FROM <table> WHERE <condition>);
```

These conditions are available for all data types where comparison makes sense. All Comparison Operators are binary operators that return values of TRUE, FALSE, or UNKNOWN (NULL).

Expressions that correlate to just one outer table in the outer query block are supported, and these correlated expressions can be comparison operators.

The following subquery scenarios are supported:

```
SELECT * FROM T1 WHERE T1.x =  (SELECT MAX(c1) FROM T2);
SELECT * FROM T1 WHERE T1.x >= (SELECT MAX(c1) FROM T2 WHERE T1.y = T2.c2);
SELECT * FROM T1 WHERE T1.x <= (SELECT MAX(c1) FROM T2 WHERE T1.y = T2.c2);
```

See Also

- Subquery Restrictions

LIKE Pattern Matching

HP Vertica supports `LIKE` pattern-matching conditions in subqueries and take the following form:

string-expression `[ NOT ] { LIKE | ILIKE | LIKEB | ILIKEB }` *string-expression*

The following command searches for customers whose company name starts with "Ev" and returns the total count:

```
=> SELECT COUNT(*) FROM customer_dimension WHERE customer_name LIKE
      (SELECT 'Ev%' FROM customer_dimension LIMIT 1);
 count
-------
   153
(1 row)
```

HP Vertica also supports single-row subqueries as the pattern argument for LIKEB and ILIKEB predicates; for example:

```
=> SELECT * FROM t1 WHERE t1.x LIKEB (SELECT t2.x FROM t2);
```

The following symbols are substitutes for the LIKE keywords:

```
~~    LIKE
~#    LIKEB
~~*   ILIKE
~#*   ILIKEB
!~~   NOT LIKE
!~#   NOT LIKEB
!~~*  NOT ILIKE
!~#*  NOT IILIKEB
```

Note: The `ESCAPE` keyword is not valid for the above symbols.

See LIKE-predicate in the SQL Reference Manual for additional examples.

ANY (SOME) and ALL

Normally, you use operators like equal and greater-than only on subqueries that return one row. With ANY and ALL, however, comparisons can be made on subqueries that return multiple rows. The ANY and ALL keywords let you specify whether *any* or *all* of the subquery values, respectively, match the specified condition.

These subqueries take the following form:

```
expression comparison-operator { ANY | SOME } ( subquery )expression comparison-operator ALL (
subquery )
```

Notes

- The keyword SOME is an alias for ANY.
- IN is equivalent to = ANY.
- NOT IN is equivalent to <> ALL.

ANY Subqueries

Subqueries that use the ANY keyword yield a Boolean result when *any* value retrieved in the subquery matches the value of the lefthand expression.

Expression	Returns
`> ANY(1,10,100)`	Returns TRUE for any value > 1 (greater than at least one value or greater than the minimum value)
`< ANY(1,10,100)`	Returns TRUE for any value < 100 (less than at least one value or less than the maximum value)
`= ANY(1,10,100)`	Returns TRUE for any value = 1 or 10 or 100 (equals *any* of the values)

ANY Subquery Examples

- An ANY subquery within an expression. Note that the second statement uses the SOME keyword:

  ```
  => SELECT c1 FROM t1 WHERE COALESCE((t1.c1 > ANY (SELECT c1 FROM t2)), TRUE);
  => SELECT c1 FROM t1 WHERE COALESCE((t1.c1 > SOME (SELECT c1 FROM t2)), TRUE);
  ```

- ANY noncorrelated subqueries without aggregates:

```
=> SELECT c1 FROM t1 WHERE c1 = ANY (SELECT c1 FROM t2) ORDER BY c1;
```

Note that omitting the ANY keyword returns an error because more than one row would be returned by the subquery used as an expression:

```
=> SELECT c1 FROM t1 WHERE c1 = (SELECT c1 FROM t2) ORDER BY c1;
```

- ANY noncorrelated subqueries with aggregates:

```
=> SELECT c1 FROM t1 WHERE c1 <> ANY (SELECT MAX(c1) FROM t2) ORDER BY c1;
=> SELECT c1 FROM t1 GROUP BY c1 HAVING c1 <> ANY (SELECT MAX(c1) FROM t2)
    ORDER BY c1;
```

- ANY noncorrelated subqueries with aggregates and a GROUP BY clause:

```
=> SELECT c1 FROM t1 WHERE c1 <> ANY (SELECT MAX(c1) FROM t2 GROUP BY c2)
     ORDER BY c1;
```

- ANY noncorrelated subqueries with a GROUP BY clause:

```
=> SELECT c1 FROM t1 WHERE c1 <=> ANY (SELECT c1 FROM t2 GROUP BY c1)
     ORDER BY c1;
```

- ANY correlated subqueries with no aggregates or GROUP BY clause:

```
=> SELECT c1 FROM t1 WHERE c1 >= ANY (SELECT c1 FROM t2 WHERE t2.c2 = t1.c2)
     ORDER BY c1;
```

ALL Subqueries

Subqueries that use the ALL keyword yield a Boolean result when *all* values retrieved in the subquery match the specified condition of the lefthand expression.

Expression	Returns
`> ALL(1,10,100)`	Returns the expression value > 100 (greater than the maximum value)
`< ALL(1,10,100)`	Returns the expression value < 1 (less than the minimum value)

ALL Subquery Examples

Following are some examples of ALL (subquery):

- ALL noncorrelated subqueries without aggregates:

```
=> SELECT c1 FROM t1 WHERE c1 >= ALL (SELECT c1 FROM t2) ORDER BY c1;
```

- ALL noncorrelated subqueries with aggregates:

```
=> SELECT c1 FROM t1 WHERE c1 = ALL (SELECT MAX(c1) FROM t2) ORDER BY c1;
=> SELECT c1 FROM t1 GROUP BY c1 HAVING c1 <> ALL (SELECT MAX(c1) FROM t2)
     ORDER BY c1;
```

- ALL noncorrelated subqueries with aggregates and a GROUP BY clause:

```
=> SELECT c1 FROM t1 WHERE c1 <= ALL (SELECT MAX(c1) FROM t2 GROUP BY c2)
     ORDER BY c1;
```

- ALL noncorrelated subqueries with a GROUP BY clause:

```
=> SELECT c1 FROM t1 WHERE c1 <> ALL (SELECT c1 FROM t2 GROUP BY c1)
     ORDER BY c1;
```

See Also

- Subquery Restrictions

EXISTS and NOT EXISTS

The EXISTS predicate is one of the most common predicates used to build conditions that use noncorrelated and correlated subqueries. Use EXISTS to identify the existence of a relationship without regard for the quantity. For example, EXISTS returns true if the subquery returns any rows, and NOT EXISTS returns true if the subquery returns no rows.

[NOT] EXISTS subqueries take the following form:

```
expression [ NOT ] EXISTS ( subquery )
```

The `EXISTS` condition is considered to be met if the subquery returns at least one row. Since the result depends only on whether any records are returned, and not on the contents of those records, the output list of the subquery is normally uninteresting. A common coding convention is to write all `EXISTS` tests as follows:

```
EXISTS (SELECT 1 WHERE ...)
```

In the above fragment, SELECT 1 returns the value 1 for every record in the query. If the query returns, for example, five records, it returns 5 ones. The system doesn't care about the real values in those records; it just wants to know if a row is returned.

Alternatively, a subquery’s select list that uses EXISTS might consist of the asterisk (*). You do not need to specify column names, because the query tests for the existence or nonexistence of records that meet the conditions specified in the subquery.

```
EXISTS (SELECT * WHERE ...)
```

Notes

- If EXISTS (subquery) returns at least 1 row, the result is TRUE.
- If EXISTS (subquery) returns no rows, the result is FALSE.
- If NOT EXISTS (subquery) returns at least 1 row, the result is FALSE.
- If NOT EXISTS (subquery) returns no rows, the result is TRUE.

Examples

The following query retrieves the list of all the customers who purchased anything from any of the stores amounting to more than 550 dollars:

```
=> SELECT customer_key, customer_name, customer_state
   FROM public.customer_dimension WHERE EXISTS
     (SELECT 1 FROM store.store_sales_fact
      WHERE customer_key = public.customer_dimension.customer_key
      AND sales_dollar_amount > 550)
   AND customer_state = 'MA' ORDER BY customer_key;
 customer_key |   customer_name    | customer_state
--------------+--------------------+----------------
        14818 | William X. Nielson | MA
        18705 | James J. Goldberg  | MA
        30231 | Sarah N. McCabe    | MA
        48353 | Mark L. Brown      | MA
(4 rows)
```

Whether you use EXISTS or IN subqueries depends on which predicates you select in outer and inner query blocks. For example, to get a list of all the orders placed by all stores on January 2, 2003 for vendors with records in the vendor table:

```
=> SELECT store_key, order_number, date_ordered
   FROM store.store_orders_fact WHERE EXISTS
     (SELECT 1 FROM public.vendor_dimension
      WHERE public.vendor_dimension.vendor_key = store.store_orders_fact.vendor_key)
   AND date_ordered = '2003-01-02';
 store_key | order_number | date_ordered
-----------+--------------+--------------
        37 |         2559 | 2003-01-02
        16 |          552 | 2003-01-02
        35 |         1156 | 2003-01-02
        13 |         3885 | 2003-01-02
        25 |          554 | 2003-01-02
        21 |         2687 | 2003-01-02
        49 |         3251 | 2003-01-02
        19 |         2922 | 2003-01-02
        26 |         1329 | 2003-01-02
        40 |         1183 | 2003-01-02
(10 rows)
```

The above query looks for existence of the vendor and date ordered. To return a particular value, rather than simple existence, the query looks for orders placed by the vendor who got the best deal on January 4, 2004:

```
=> SELECT store_key, order_number, date_ordered
   FROM store.store_orders_fact ord, public.vendor_dimension vd
   WHERE ord.vendor_key = vd.vendor_key AND vd.deal_size IN
      (SELECT MAX(deal_size) FROM public.vendor_dimension)
    AND date_ordered = '2004-01-04';
 store_key | order_number | date_ordered
-----------+--------------+--------------
       166 |        36008 | 2004-01-04
       113 |        66017 | 2004-01-04
       198 |        75716 | 2004-01-04
        27 |       150241 | 2004-01-04
       148 |       182207 | 2004-01-04
         9 |       188567 | 2004-01-04
        45 |       202416 | 2004-01-04
        24 |       250295 | 2004-01-04
       121 |       251417 | 2004-01-04
(9 rows)
```

See Also

- Subquery Restrictions

IN and NOT IN

While you cannot equate a single value to a set of values, you can check to see if a single value is found within that set of values. Use the `IN` clause for multiple-record, single-column subqueries. After the subquery returns results introduced by IN or NOT IN, the outer query uses them to return the final result.

[NOT] IN subqueries take the following form:

`{ expression [ NOT ] IN ( subquery )| expression [ NOT ] IN ( expression ) }`

There is no limit to the number of parameters passed to the `IN` clause of the `SELECT` statement; for example:

```
=> SELECT * FROM tablename WHERE column IN (a, b, c, d, e, ...);
```

HP Vertica also supports queries where two or more outer expressions refer to different inner expressions:

```
=> SELECT * FROM A WHERE (A.x,A.x) IN (SELECT B.x, B.y FROM B);
```

Examples

The following query uses the VMart schema to illustrate the use of outer expressions referring to different inner expressions:

```
=> SELECT product_description, product_price FROM product_dimension
   WHERE (product_dimension.product_key, product_dimension.product_key) IN
      (SELECT store.store_orders_fact.order_number,
         store.store_orders_fact.quantity_ordered
       FROM store.store_orders_fact);
 product_description         | product_price
-----------------------------+---------------
 Brand #72 box of candy      |           326
 Brand #71 vanilla ice cream |           270
(2 rows)
```

To find all products supplied by stores in MA, first create the inner query and run it to ensure that it works as desired. The following query returns all stores located in MA:

```
=> SELECT store_key FROM store.store_dimension WHERE store_state = 'MA';
 store_key
-----------
        13
        31
(2 rows)
```

Then create the outer or main query that specifies all distinct products that were sold in stores located in MA. This statement combines the inner and outer queries using the IN predicate:

```
=> SELECT DISTINCT s.product_key, p.product_description
   FROM store.store_sales_fact s, public.product_dimension p
   WHERE s.product_key = p.product_key
       AND s.product_version = p.product_version
       AND s.store_key IN
            (SELECT store_key
             FROM store.store_dimension
             WHERE store_state = 'MA')
   ORDER BY s.product_key;
 product_key |          product_description
-------------+-------------------------------------
           1 | Brand #1 white bread
           1 | Brand #4 vegetable soup
           3 | Brand #9 wheelchair
           5 | Brand #15 cheddar cheese
           5 | Brand #19 bleach
           7 | Brand #22 canned green beans
           7 | Brand #23 canned tomatoes
           8 | Brand #24 champagne
           8 | Brand #25 chicken nuggets
```

```
          11 | Brand #32 sausage
         ...    ...
(281 rows)
```

When using NOT IN, the subquery returns a list of zero or more values in the outer query where the comparison column does not match any of the values returned from the subquery. Using the previous example, NOT IN returns all the products that are not supplied from MA.

Notes

HP Vertica supports multicolumn NOT IN subqueries in which the columns are not marked NOT NULL. Previously, marking the columns NOT NULL was a requirement; now, if one of the columns is found to contain a a NULL value during query execution, HP Vertica returns a run-time error.

Similarly, IN subqueries nested within another expression are not supported if any of the column values are NULL. For example, if in the following statement column x from either table contained a NULL value, Vertica would return a run-time error:

```
=> SELECT * FROM t1 WHERE (x IN (SELECT x FROM t2)) IS FALSE;
   ERROR: NULL value found in a column used by a subquery
```

See Also

- Subquery Restrictions
- IN-predicate

Subqueries in the SELECT List

Subqueries can occur in the select list of the containing query. The results from the following statement are ordered by the first column (customer_name). You could also write `ORDER BY 2` and specify that the results be ordered by the select-list subquery.

```
=> SELECT c.customer_name, (SELECT AVG(annual_income) FROM customer_dimension
    WHERE deal_size = c.deal_size) AVG_SAL_DEAL FROM customer_dimension c
     ORDER BY 1;
 customer_name | AVG_SAL_DEAL
---------------+--------------
 Goldstar      |       603429
 Metatech      |       628086
 Metadata      |       666728
 Foodstar      |       695962
 Verihope      |       715683
 Veridata      |       868252
 Bettercare    |       879156
 Foodgen       |       958954
 Virtacom      |       991551
 Inicorp       |      1098835
```

```
...
```

Notes

- Scalar subqueries in the select-list return a single row/column value. These subqueries use Boolean comparison operators: =, >, <, <>, <=, >=.

  If the query is correlated, it returns NULL if the correlation results in 0 rows. If the query returns more than one row, the query errors out at run time and HP Vertica displays an error message that the scalar subquery must only return 1 row.

- Subquery expressions such as [NOT] IN, [NOT] EXISTS, ANY/SOME, or ALL always return a single Boolean value that evaluates to TRUE, FALSE, or UNKNOWN; the subquery itself can have many rows. Most of these queries can be correlated or noncorrelated.

  Note: ALL subqueries cannot be correlated.

- Subqueries in the ORDER BY and GROUP BY clauses are supported; for example, the following statement says to order by the first column, which is the select-list subquery:

  ```
  SELECT (SELECT MAX(x) FROM t2 WHERE y=t1.b) FROM t1 ORDER BY 1;
  ```

See Also

- Subquery Restrictions

WITH Clauses in SELECT

While not strictly subqueries, `WITH` clauses are concomitant queries within a larger, primary query. HP Vertica evaluates each `WITH` clause only once while executing the primary query. You can reference the results of any evaluated `WITH` clause during the primary query transaction, as if the results existed in a temporary table using the `WITH` query name. Each WITH clause query must have a unique name. Attempting to use same-name aliases for WITH clause query names causes an error.

You can also use the results of a previously evaluated `WITH` clause in any subsequent `WITH` clause or select statement. Combining `WITH` clause results in another WITH query lets you successively use the results of evaluated `WITH` clauses (the limit of evaluated `WITH` clauses is undefined). `WITH` clauses do not support INSERT, DELETE, and UPDATE statements, and you cannot use them recursively.

Using WITH Clauses

This example illustrates the use of two WITH clauses evaluated before the primary query, `regional_sales` and `top_regions`. Following are the evaluation and usage steps:

1. The `regional_sales` clause evaluates its `select` statement from table `orders`. After evaluation, each of the columns selected in this `WITH` clause can be queried or used as part of a later `WITH` clause.

2. The `top_regions` clause is evaluated using values from results obtained in the `regional_sales` clause.

3. The primary query is then evaluated using the results obtained in the top_regions clause, and accessed as if top_regions is a temporary table.

```
-- Begin WITH clauses,
-- First WITH clause,regional_sales
WITH
   regional_sales AS (
      SELECT region, SUM(amount) AS total_sales
      FROM orders
      GROUP BY region),
-- Second WITH clause top_regions
   top_regions AS (
       SELECT region
       FROM regional_sales
       WHERE total_sales > (SELECT SUM (total_sales)/10 FROM regional_sales) )
-- End defining WITH clause statement
-- Begin main primary query
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

See Also

- Subquery Restrictions
- WITH Clause

Noncorrelated and Correlated Subqueries

A class of queries is evaluated by running the subquery once and then substituting the resulting value or values into the `WHERE` clause of the outer query. These self-contained queries are called **noncorrelated** (simple) subqueries; you can run them by themselves and inspect the results independent of their containing statements. A **correlated** subquery, however, is dependent on its containing statement, from which it references one or more columns.

See the following table for examples of the two subquery types:

Noncorrelated (simple)	Correlated
`SELECT name, street, city, stateFROM addresses as ADD WHERE state IN (SELECT state FROM states);`	`SELECT name, street, city, state FROM addresses as ADD WHERE EXISTS (SELECT * FROM states as ST WHERE ST.state = ADD.state);`
The subquery (SELECT state FROM states) is independent from the containing query. It is evaluated first and its results are passed to the outer query block.	The subquery needs values from the state column in containing query, and results are then passed to the outer query block. The subquery is evaluated for every record of the outer block because the column is being used in the subquery.

The difference between noncorrelated (simple) subqueries and correlated subqueries is that in simple subqueries, the containing (outer) query only has to take action on the results from the subquery (inner query). In a correlated subquery, the outer query block provides values for subquery to use in its evaluation.

Notes

- You can use an outer join to obtain the same effect as a correlated subquery.
- Arbitrary uncorrelated queries are permitted in the WHERE clause as single-row expressions; for example:

  ```
  => SELECT COUNT(*) FROM SubQ1 WHERE SubQ1.a = (SELECT y from SubQ2);
  ```

- Noncorrelated queries in the HAVING clause as single-row expressions are permitted; for example:

  ```
  => SELECT COUNT(*) FROM SubQ1 GROUP BY SubQ1.a   HAVING SubQ1.a = (SubQ1.a & (SELECT y
  from SubQ2));
  ```

See Also

- Subquery Restrictions

Flattening FROM Clause Subqueries and Views

FROM clause subquery are always evaluated before their containing query. For example, in the following statement, the HP Vertica optimizer must evaluate all records in table `t1` before it can evaluate the records in table `t0`:

```
=> SELECT * FROM (SELECT a, MAX(a) AS max FROM (SELECT * FROM t1) AS t0 GROUP BY a);
```

In an optimization called subquery flattening, some `FROM` clause subqueries are *flattened* into the containing query, improving the performance of subquery-containing queries.

Using the above query, HP Vertica can internally flatten it as follows:

```
=> SELECT * FROM (SELECT a, MAX(a) FROM t1 GROUP BY a) AS t0;
```

Both queries return the same results, but the flattened query runs more quickly.

The optimizer will choose the flattening plan if the subquery or view does not contain the following:

- Aggregates
- Analytics
- An outer join (left, right or full)
- GROUP BY, ORDER BY, or HAVING clause
- DISTINCT keyword
- LIMIT or OFFSET clause
- UNION, EXCEPT, or INTERSECT clause
- EXISTS subquery

To see if a FROM clause subquery has been flattened, inspect the query plan, as described in EXPLAIN in the SQL Reference Manual. Typically, the number of value expression nodes (`ValExpNode`) decreases after flattening.

Flattening Views

When you specify a **view** in a `FROM` clause query, HP Vertica first replaces the view name with the view definition query, creating further opportunities for subquery flattening. This process is called view flattening and works the same way as subquery flattening. See Implementing Views in the Administrator's Guide for additional details about views.

Examples

If you have a predicate that applies to a view or subquery, the flattening operation can allow for optimizations by evaluating the predicates before the flattening takes place. In this example, without flattening, HP Vertica must first evaluate the subquery, and only then can the predicate `WHERE x > 10` be applied. In the flattened subquery, HP Vertica applies the predicate before evaluating the subquery, thus reducing the amount of work for the optimizer because it returns only the records `WHERE x > 10` to the containing query.

Assume that view v1 is defined as follows:

```
=> CREATE VIEW v1 AS SELECT * FROM a;
```

You enter the following query:

```
=> SELECT * FROM v1 JOIN b ON x=y WHERE x > 10;
```

HP Vertica internally transforms the above query as follows:

```
=> SELECT * FROM (SELECT * FROM a) AS t1 JOIN b ON x=y WHERE x > 10;
```

And the flattening mechanism gives you the following:

```
=> SELECT * FROM a JOIN b ON x=y WHERE x > 10;
```

The following example is how HP Vertica transforms `FROM` clause subqueries within a `WHERE` clause `IN` subquery:

- Original query: `SELECT * FROM a WHERE b IN (SELECT b FROM (SELECT * FROM t2) AS D WHERE x=1;`
- Flattened query: `SELECT * FROM a WHERE b IN (SELECT b FROM t2) AS D WHERE x=1;`

See Also

- Subquery Restrictions

Subqueries in UPDATE and DELETE Statements

You can nest subqueries within UPDATE and DELETE statements.

UPDATE Subqueries

If you want to update records in a table based on values that are stored in other database tables, you can nest a subquery within an `UPDATE` statement. See also UPDATE in the SQL Reference Manual.

Syntax

```
UPDATE [[db-name.]schema.]table SET column = { expression | DEFAULT } [, ...]
[ FROM from-list ]
[ WHERE Clause ]
```

Semantics	`UPDATE` changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need to be specified in the `SET` clause. Columns that are not explicitly modified retain their previous values.
Outputs	On successful completion, an update operation returns a count, which represents the number of rows updated. A count of 0 is not an error; it means that no rows matched the condition.

Performance Tip

To be eligible for `DELETE` optimization, all target table columns referenced in a `DELETE` or `UPDATE` statement's WHERE clause must be in the projection definition.

For example, the following simple schema has two tables and three projections:

```
CREATE TABLE tb1 (a INT, b INT, c INT, d INT);
CREATE TABLE tb2 (g INT, h INT, i INT, j INT);
```

The first projection references all columns in `tb1` and sorts on column `a`:

```
CREATE PROJECTION tb1_p AS SELECT a, b, c, d FROM tb1 ORDER BY a;
```

The buddy projection references and sorts on column `a` in `tb1`:

```
CREATE PROJECTION tb1_p_2 AS SELECT a FROM tb1 ORDER BY a;
```

This projection references all columns in `tb2` and sorts on column `i`:

```
CREATE PROJECTION tb2_p AS SELECT g, h, i, j FROM tb2 ORDER BY i;
```

Consider the following DML statement, which references `tb1.a` in its `WHERE` clause. Since both projections on `tb1` contain column `a`, both are eligible for the optimized `DELETE`:

```
DELETE FROM tb1 WHERE tb1.a IN (SELECT tb2.i FROM tb2);
```

Restrictions

Optimized `DELETE`s are not supported under the following conditions:

- With pre-join projections on nodes that are down
- With replicated and pre-join projections if subqueries reference the target table. For example, the following syntax is not supported:

  ```
  DELETE FROM tb1 WHERE tb1.a IN (SELECT e FROM tb2, tb2 WHERE tb2.e = tb1.e);
  ```

- With subqueries that do not return multiple rows. For example, the following syntax is not supported:

  ```
  DELETE FROM tb1 WHERE tb1.a = (SELECT k from tb2);
  ```

Notes and Restrictions

- The table specified in the `UPDATE` list cannot also appear in the FROM list (no self joins); for example, the following is not allowed:

  ```
  => BEGIN;
  => UPDATE result_table
     SET address='new' || r2.address
     FROM result_table r2
     WHERE r2.cust_id = result_table.cust_id + 10;
     ERROR:  Self joins in UPDATE statements are not allowed
     DETAIL:  Target relation result_table also appears in the FROM list
  ```

- If more than one row in a table to be updated matches the `WHERE` predicate, HP Vertica returns an error specifying which row had more than one match.

UPDATE Example

The following series of commands illustrate the use of subqueries in `UPDATE` statements; they all use the following simple schema:

```
=> CREATE TABLE result_table(
       cust_id INTEGER,
       address VARCHAR(2000));
```

Enter some customer data:

```
=> COPY result_table FROM stdin delimiter ',' DIRECT;
   20, Lincoln Street
   30, Booth Hill Road
   30, Beach Avenue
   40, Mt. Vernon Street
   50, Hillside Avenue
   \.
```

Query the table you just created:

```
=> SELECT * FROM result_table;
 cust_id |      address
---------+--------------------
      20 |  Lincoln Street
      30 |  Beach Avenue
      30 |  Booth Hill Road
      40 |  Mt. Vernon Street
      50 |  Hillside Avenue
(5 rows)
```

Create a second table called `new_addresses:`

```
=> CREATE TABLE new_addresses(
     new_cust_id integer,
     new_address VARCHAR(200));
```

Enter some customer data.

Note: The following `COPY` statement creates an entry for a customer ID with a value of 60, which does not have a matching value in the `result_table` table:

```
=> COPY new_addresses FROM stdin delimiter ',' DIRECT;
   20, Infinite Loop
   30, Loop Infinite
   60, New Addresses
   \.
```

Query the `new_addresses` table:

```
=> SELECT * FROM new_addresses;
 new_cust_id |  new_address
-------------+---------------
          20 |  Infinite Loop
          30 |  Loop Infinite
          60 |  New Addresses
(3 rows)
```

Commit the changes:

```
=> COMMIT;
```

In the following example, a noncorrelated subquery is used to change the address record in `results_table` to 'New Address' when the query finds a customer ID match in both tables:

```
=> UPDATE result_table
   SET address='New Address'
   WHERE cust_id IN (SELECT new_cust_id FROM new_addresses);
```

The output returns the expected count indicating that three rows were updated:

```
 OUTPUT
--------
      3
(1 row)
```

Now query the `result_table` table to see the changes for matching customer ID 20 and 30. Addresses for customer ID 40 and 50 are not updated:

```
=> SELECT * FROM result_table;
 cust_id |     address
```

```
---------+------------------
      20 | New Address
      30 | New Address
      30 | New Address
      40 | Mt. Vernon Street
      50 | Hillside Avenue
(5 rows)
```

To preserve your original data, issue the `ROLLBACK` command:

```
=> ROLLBACK;
```

In the following example, a correlated subquery is used to replace all `address` records in the `results_table` with the `new_address` record from the `new_addresses` table when the query finds match on the customer ID in both tables:

```
=> UPDATE result_table
   SET address=new_addresses.new_address
   FROM new_addresses
   WHERE cust_id = new_addresses.new_cust_id;
```

Again, the output returns the expected count indicating that three rows were updated:

```
 OUTPUT
--------
      3
(1 row)
```

Now query the `result_table` table to see the changes for customer ID 20 and 30. Addresses for customer ID 40 and 50 are not updated, and customer ID 60 is omitted because there is no match:

```
=> SELECT * FROM result_table;
 cust_id |     address
---------+------------------
      20 | Infinite Loop
      30 | Loop Infinite
      30 | Loop Infinite
      40 | Mt. Vernon Street
      50 | Hillside Avenue
(5 rows)
```

DELETE Subqueries

If you want to delete records in a table based on values that are stored in other database tables, you can nest a subquery within a `DELETE` statement. See also DELETE in the SQL Reference Manual.

Syntax

```
DELETE FROM [[db-name.]schema.]table WHERE Clause
```

Semantics	The `DELETE` operation deletes rows that satisfy the WHERE clause from the specified table. If the `WHERE` clause is absent, all table rows are deleted. The result is a valid, even though the statement leaves an empty table.
Outputs	On successful completion, a `DELETE` operation returns a count, which represents the number of rows deleted. A count of 0 is not an error; it means that no rows matched the condition.

DELETE Example

The following series of commands illustrate the use of subqueries in `DELETE` statements; they all use the following simple schema:

```
=> CREATE TABLE t (a INTEGER);
=> CREATE TABLE  t2 (a INTEGER);
=> INSERT INTO t VALUES (1);
=> INSERT INTO t VALUES (2);
=> INSERT INTO t2 VALUES (1);
=> COMMIT;
```

The following command deletes the expected row from table `t`:

```
=> DELETE FROM t WHERE t.a IN (SELECT t2.a FROM t2);
 OUTPUT
--------
      1
(1 row)
```

Notice that table `t` now has only one row,instead of two:

```
=> SELECT * FROM t;
 a
---
 2
(1 row)
```

To preserve the data for this example, issue the rollback command:

```
=> ROLLBACK;
```

The following command deletes the expected two rows:

```
=> DELETE FROM t WHERE EXISTS (SELECT * FROM t2);
 OUTPUT
--------
      2
(1 row)
```

Now table `t` contains no rows:

```
=> SELECT * FROM t;
 a
---
(0 rows)
```

Roll back to the previous state and verify that you still have two rows:

```
=> ROLLBACK;SELECT * FROM t;
 a
---
 1
 2
(2 rows)
```

The following command uses a correlated subquery to delete all rows in table `t` where `t.a` matches a value of `t2.a`.

```
=> DELETE FROM t WHERE EXISTS (SELECT * FROM t2 WHERE t.a = t2.a);
 OUTPUT
--------
      1
(1 row)
```

Query the table to verify the row was deleted:

```
=> SELECT * FROM t;
 a
---
 2
(1 row)
```

Roll back to the previous state and query the table again:

```
=> ROLLBACK;=> SELECT * FROM t;
 a
---
 1
 2
(2 rows)
```

See Also

- Subquery Restrictions

Subquery Examples

This topic illustrates some of the subqueries you can write. The examples use the VMart example database.

Single-Row Subqueries

Single-row subqueries are used with single-row comparison operators (=, >=, <=, <>, and <=>) and return exactly one row.

For example, the following query retrieves the name and hire date of the oldest employee in the Vmart database:

```
=> SELECT employee_key, employee_first_name, employee_last_name, hire_date
   FROM employee_dimension
   WHERE hire_date = (SELECT MIN(hire_date) FROM employee_dimension);
 employee_key | employee_first_name | employee_last_name | hire_date
--------------+---------------------+--------------------+------------
         2292 | Mary                | Bauer              | 1956-01-11
(1 row)
```

Multiple-Row Subqueries

Multiple-row subqueries return multiple records.

For example, the following IN clause subquery returns the names of the employees making the highest salary in each of the six regions:

```
=> SELECT employee_first_name, employee_last_name, annual_salary, employee_region
    FROM employee_dimension WHERE annual_salary IN
     (SELECT MAX(annual_salary) FROM employee_dimension GROUP BY employee_region)
   ORDER BY annual_salary DESC;
 employee_first_name | employee_last_name | annual_salary |  employee_region
---------------------+--------------------+---------------+-------------------
 Alexandra           | Sanchez            |        992363 | West
 Mark                | Vogel              |        983634 | South
 Tiffany             | Vu                 |        977716 | SouthWest
 Barbara             | Lewis              |        957949 | MidWest
 Sally               | Gauthier           |        927335 | East
 Wendy               | Nielson            |        777037 | NorthWest
(6 rows)
```

Multicolumn Subqueries

Multicolumn subqueries return one or more columns. Sometimes a subquery's result set is evaluated in the containing query in column-to-column and row-to-row comparisons.

Note: Multicolumn subqueries can use the <>, !=, and = operators but not the <, >, <=, >= operators.

You can substitute some multicolumn subqueries with a join, with the reverse being true as well. For example, the following two queries ask for the sales transactions of all products sold online to customers located in Massachusetts and return the same result set. The only difference is the first query is written as a join and the second is written as a subquery.

Join query:	**Subquery:**
`=> SELECT *`	`=> SELECT *`
`   FROM online_sales.online_sales_fact`	`   FROM online_sales.online_sales_fact`
`   INNER JOIN public.customer_dimension`	`   WHERE customer_key IN`
`   USING (customer_key)`	`     (SELECT customer_key`
`   WHERE customer_state = 'MA';`	`      FROM public.customer_dimension`
	`      WHERE customer_state = 'MA');`

The following query returns all employees in each region whose salary is above the average:

```
=> SELECT e.employee_first_name, e.employee_last_name, e.annual_salary,
      e.employee_region, s.average
   FROM employee_dimension e,
     (SELECT employee_region, AVG(annual_salary) AS average
      FROM employee_dimension GROUP BY employee_region) AS s
   WHERE  e.employee_region = s.employee_region AND e.annual_salary > s.average
   ORDER BY annual_salary DESC;
 employee_first_name | employee_last_name | annual_salary | employee_region |     average
---------------------+--------------------+---------------+-----------------+------------
------
 Doug                | Overstreet         |        995533 | East            |  61192.7860
13986
 Matt                | Gauthier           |        988807 | South           | 57337.86389
02996
 Lauren              | Nguyen             |        968625 | West            | 56848.42749
14089
 Jack                | Campbell           |        963914 | West            | 56848.42749
14089
 William             | Martin             |        943477 | NorthWest       | 58928.22761
19403
 Luigi               | Campbell           |        939255 | MidWest         | 59614.91704
54545
 Sarah               | Brown              |        901619 | South           | 57337.86389
02996
 Craig               | Goldberg           |        895836 | East            |  61192.7860
13986
 Sam                 | Vu                 |        889841 | MidWest         | 59614.91704
54545
 Luigi               | Sanchez            |        885078 | MidWest         | 59614.91704
54545
 Michael             | Weaver             |        882685 | South           | 57337.86389
02996
 Doug                | Pavlov             |        881443 | SouthWest       | 57187.25105
48523
 Ruth                | McNulty            |        874897 | East            |  61192.7860
13986
 Luigi               | Dobisz             |        868213 | West            | 56848.42749
14089
 Laura               | Lang               |        865829 | East            |  61192.7860
13986
 ...
```

You can also use the EXCEPT, INTERSECT, and UNION [ALL] keywords in FROM, WHERE, and HAVING clauses.

The following subquery returns information about all Connecticut-based customers who bought items through either stores or online sales channel and whose purchases amounted to more than 500 dollars:

```
=> SELECT DISTINCT customer_key, customer_name FROM public.customer_dimension
   WHERE customer_key IN (SELECT customer_key FROM store.store_sales_fact
      WHERE sales_dollar_amount > 500
      UNION ALL
      SELECT customer_key FROM online_sales.online_sales_fact
      WHERE sales_dollar_amount > 500)
   AND customer_state = 'CT';
 customer_key |  customer_name
--------------+------------------
          200 | Carla Y. Kramer
          733 | Mary Z. Vogel
          931 | Lauren X. Roy
         1533 | James C. Vu
         2948 | Infocare
         4909 | Matt Z. Winkler
         5311 | John Z. Goldberg
         5520 | Laura M. Martin
         5623 | Daniel R. Kramer
         6759 | Daniel Q. Nguyen
 ...
```

HAVING Clause Subqueries

A HAVING clause is used in conjunction with the GROUP BY clause to filter the select-list records that a GROUP BY returns. HAVING clause subqueries must use Boolean comparison operators: =, >, <, <>, <=, >= and take the following form:

```
SELECT <column, ...>
FROM <table>
GROUP BY <expression>
HAVING <expression>
  (SELECT <column, ...>
   FROM <table>
   HAVING <expression>);
```

For example, the following statement uses the VMart database and returns the number of customers who purchased lowfat products. Note that the GROUP BY clause is required because the query uses an aggregate (COUNT).

```
=> SELECT s.product_key, COUNT(s.customer_key) FROM store.store_sales_fact s
   GROUP BY s.product_key HAVING s.product_key IN
     (SELECT product_key FROM product_dimension WHERE diet_type = 'Low Fat');
```

The subquery first returns the product keys for all low-fat products, and the outer query then counts the total number of customers who purchased those products.

```
 product_key | count
 -------------+-------
           15 |     2
           41 |     1
           66 |     1
          106 |     1
          118 |     1
          169 |     1
          181 |     2
          184 |     2
          186 |     2
          211 |     1
          229 |     1
          267 |     1
          289 |     1
          334 |     2
          336 |     1
(15 rows)
```

Subquery Restrictions

The following list summarizes subquery restrictions in HP Vertica.

- Subqueries are not allowed in the defining query of a CREATE PROJECTION statement.
- Subqueries can be used in the select-list, but GROUP BY or aggregate functions are not allowed in the query if the subquery is not part of the GROUP BY clause in the containing query; for example, the following two statement returns an error message:

```
=> SELECT y, (SELECT MAX(a) FROM t1) FROM t2 GROUP BY y;
   ERROR:  subqueries in the SELECT or ORDER BY are not supported if the
   subquery is not part of the GROUP BY
=> SELECT MAX(y), (SELECT MAX(a) FROM t1) FROM t2;
   ERROR:  subqueries in the SELECT or ORDER BY are not supported if the
   query has aggregates and the subquery is not part of the GROUP BY
```

- Subqueries are supported within UPDATE statements with the following exceptions:
  - You cannot use SET column = {expression} to specify a subquery.
  - The table specified in the UPDATE list cannot also appear in the FROM list (no self joins).
- FROM clause subqueries require an alias but tables do not. If the table has no alias, the query must refer to columns inside it as <table>.<column>; however, if the column names are uniquely identified among all tables used by the query, then preceding the column with a table name is not enforced.
- If the ORDER BY clause is inside a FROM clause subquery, rather than in the containing query, the query could return unexpected sort results. This is because HP Vertica data comes from multiple nodes, so sort order cannot be guaranteed unless an ORDER BY clause is specified in

the outer query block. This behavior is compliant with the SQL standard but it might differ from other databases.

- Multicolumn subqueries cannot use the <, >, <=, >= comparison operators. They can use <>, !=, and = operators.
- WHERE and HAVING clause subqueries must use Boolean comparison operators: =, >, <, <>, <=, >=. Those subqueries can be noncorrelated and correlated.
  - [NOT] IN and ANY subqueries nested within another expression are not supported if any of the column values are NULL. In the following statement, for example, if column x from either table t1 or t2 contains a NULL value, HP Vertica returns a run-time error:

    ```
    => SELECT * FROM t1 WHERE (x IN (SELECT x FROM t2)) IS FALSE;   ERROR:  NULL value f
    ound in a column used by a subquery
    ```

- HP Vertica returns an error message during subquery run time on scalar subqueries that return more than one row.
- Aggregates and GROUP BY clauses are allowed in subqueries, as long as those subqueries are not correlated.
- Correlated expressions under ALL and [NOT] IN are not supported.
- Correlated expressions under OR are not supported.
- Multiple correlations are allowed only for subqueries that are joined with an equality predicate (<, >, <=, >=, =, <>, <=>) but IN/NOT IN, EXISTS/NOT EXISTS predicates within correlated subqueries are not allowed:

  ```
  => SELECT t2.x, t2.y, t2.z FROM t2 WHERE t2.z NOT IN
         (SELECT t1.z FROM t1 WHERE t1.x = t2.x);
     ERROR: Correlated subquery with NOT IN is not supported
  ```

- Up to one level of correlated subqueries is allowed in the `WHERE` clause if the subquery references columns in the immediate outer query block. For example, the following query is not supported because the **`t2.x = t3.x`** subquery can only refer to table `t1` in the outer query, making it a correlated expression because `t3.x` is two levels out:

  ```
  => SELECT t3.x, t3.y, t3.z FROM t3 WHERE t3.z IN (
       SELECT t1.z FROM t1 WHERE EXISTS (
          SELECT 'x' FROM t2 WHERE t2.x = t3.x) AND t1.x = t3.x);
  ERROR:  More than one level correlated subqueries are not supported
  ```

  The query is supported if it is rewritten as follows:

```
=> SELECT t3.x, t3.y, t3.z FROM t3 WHERE t3.z IN
       (SELECT t1.z FROM t1 WHERE EXISTS
         (SELECT 'x' FROM t2 WHERE t2.x = t1.x)
   AND t1.x = t3.x);
```

Joins

Queries can combine records from multiple tables, or multiple instances of the same table. A query that combines records from one or more tables is called a join. Joins are allowed in a `SELECT` statement, as well as inside a subquery.

HP Vertica supports the following join types:

- Inner (including natural, cross) joins
- Left, right, and full outer joins
- Optimizations for equality and range joins predicates
- Hash, merge and sort-merge join algorithms.

There are three basic algorithms that perform a join operation: hash, merge, and nested loop:

- A hash join is used to join large data sets. The smaller joined table is used to build a hash table in memory on the join column. The HP Vertica optimizer then scans the larger table and probes the hash table to look for matches. The optimizer chooses a hash join when projections are not sorted on the join columns.
- If both inputs are pre-sorted, the optimizer can skip a sort operation and choose a merge join. The term sort-merge join refers to the case when one or both inputs must be sorted before the merge join. In this case, HP Vertica sorts the inner input but only if the outer input is already sorted on the join keys.
- HP Vertica does not support nested loop joins.

The ANSI Join Syntax

Before the ANSI SQL-92 standard introduced the new join syntax, relations (tables, views, etc.) were named in the `FROM` clause, separated by commas. Join conditions were specified in the `WHERE` clause:

```
=> SELECT * FROM T1, T2 WHERE T1.id = T2.id;
```

The ANSI SQL-92 standard provided more specific join syntax, with join conditions named in the ON clause:

```
=> SELECT * FROM T1
   [ INNER | LEFT OUTER | RIGHT OUTER | FULL OUTER | NATURAL | CROSS ] JOIN T2
   ON T1.id = T2.id
```

See SQL-99 ANSI syntax at BNF Grammar for SQL-99 for additional details.

Although some users continue to use the older join syntax, HP encourages you to use the SQL-92 join syntax whenever possible because of its many advantages:

- SQL-92 outer join syntax is portable across databases; the older syntax was not consistent between databases. (HP Vertica does not support proprietary outer join syntax such as '+' that can be used in some databases.)
- SQL-92 syntax provides greater control over whether predicates are to be evaluated during or after outer joins. This was also not consistent between databases when using the older syntax. See "Join Conditions vs. Filter Conditions" below.
- SQL-92 syntax eliminates ambiguity in the order of evaluating the joins, in cases where more than two tables are joined with outer joins.
- Union joins can be expressed using the SQL-92 syntax, but not in the older syntax.

  Note: HP Vertica does not currently support union joins.

Join Conditions vs. Filter Conditions

If you do not use the SQL-92 syntax, join conditions (predicates that are evaluated during the join) are difficult to distinguish from filter conditions (predicates that are evaluated after the join), and in some cases cannot be expressed at all. With SQL-92, join conditions and filter conditions are separated into two different clauses, the `ON` clause and the `WHERE` clause, respectively, making queries easier to understand.

- **The ON clause** contains relational operators (for example, <, <=, >, >=, <>, =, <=>) or other predicates that specify which records from the left and right input relations to combine, such as by matching foreign keys to primary keys. `ON` can be used with inner, left outer, right outer, and full outer joins. Cross joins and union joins do not use an `ON` clause.

  Inner joins return all pairings of rows from the left and right relations for which the `ON` clause evaluates to TRUE. In a left join, all rows from the left relation in the join are present in the result; any row of the left relation that does not match any rows in the right relation is still present in the result but with nulls in any columns taken from the right relation. Similarly, a right join preserves all rows from the right relation, and a full join retains all rows from both relations.
- **The WHERE clause** is evaluated after the join is performed. It filters records returned by the `FROM` clause, eliminating any records that do not satisfy the `WHERE` clause condition.

HP Vertica automatically converts outer joins to inner joins in cases where it is correct to do so, allowing the optimizer to choose among a wider set of query plans and leading to better performance.

Inner Joins

An inner join combines records from two tables based on a join predicate and requires that each record in the first table has a matching record in the second table. Inner joins, thus, return only those records from both joined tables that satisfy the join condition. Records that contain no matches are not preserved.

Inner joins take the following form:

```
SELECT <column list>
FROM <left joined table>
[INNER] JOIN <right joined table>
ON <join condition>
```

Notes

- Inner joins are are commutative and associative, which means you can specify the tables in any order you want, and the results do not change.
- If you omit the `INNER` keyword, the join is still an inner join, the most commonly used type of join.
- Join conditions that follow the `ON` keyword generally can contain many predicates connected with Boolean `AND`, `OR`, or `NOT` predicates.
- For best performance, do not join on any `LONG VARBINARY` and `LONG VARCHAR` columns.
- You can also use inner join syntax to specify joins for pre-join projections. See Pre-Join Projections and Join Predicates.

- Some SQL-related books and online tutorials refer to a left-joined table as the outer table and a right-joined table as the inner table. The HP documentation often uses the left/right table concept.

Example

In the following example, an inner join produces only the set of records that matches in both T1 and T2 when T1 and T2 have the same data type; all other data is excluded.

```
=> SELECT * FROM T1 INNER JOIN T2 ON (T1.id = T2.id);
```

If a company, for example, wants to know the dates vendors in Utah sold inventory:

```
=> SELECT v.vendor_name, d.date FROM vendor_dimension v
   INNER JOIN date_dimension d ON v.vendor_key = d.date_key
   WHERE vendor_state = 'UT';
   vendor_name    |    date
```

```
------------------+------------
 Frozen Warehouse | 2003-01-07
 Delicious Farm   | 2003-01-26
(2 rows)
```

To clarify, if the vendor dimension table contained a third row that has no corresponding date when a vendor sold inventory, then that row would not be included in the result set. Similarly, if on some date there was no inventory sold by any vendor, those rows would be left out of the result set. If you want to include all rows from one table or the other regardless of whether a match exists, you can specify an outer join.

See Also

- Join Notes and Restrictions

Equi-Joins and Non Equi-Joins

HP Vertica supports any arbitrary join expression with both matching and non-matching column values; for example:

```
SELECT * FROM fact JOIN dim ON fact.x = dim.x;
SELECT * FROM fact JOIN dim ON fact.x > dim.y;
SELECT * FROM fact JOIN dim ON fact.x <= dim.y;
SELECT * FROM fact JOIN dim ON fact.x <> dim.y;
SELECT * FROM fact JOIN dim ON fact.x <=> dim.y;
```

Note: The = and <=> operators generally run the fastest.

Equi-joins are based on equality (matching column values). This equality is indicated with an equal sign (=), which functions as the comparison operator in the `ON` clause using SQL-92 syntax or the `WHERE` clause using older join syntax.

The first example below uses SQL-92 syntax and the `ON` clause to join the online sales table with the call center table using the call center key; the query then returns the sale date key that equals the value 156:

```
=> SELECT sale_date_key, cc_open_date FROM online_sales.online_sales_fact
   INNER JOIN   online_sales.call_center_dimension
   ON (online_sales.online_sales_fact.call_center_key =
    online_sales.call_center_dimension.call_center_key
   AND sale_date_key = 156);
 sale_date_key | cc_open_date
---------------+--------------
           156 | 2005-08-12
(1 row)
```

The second example uses older join syntax and the `WHERE` clause to join the same tables to get the same results:

```
=> SELECT sale_date_key, cc_open_date
    FROM online_sales.online_sales_fact, online_sales.call_center_dimension
   WHERE online_sales.online_sales_fact.call_center_key =
      online_sales.call_center_dimension.call_center_key
   AND sale_date_key = 156;
 sale_date_key | cc_open_date
---------------+--------------
           156 | 2005-08-12
(1 row)
```

HP Vertica also permits tables with compound (multiple-column) primary and foreign keys. For example, to create a pair of tables with multi-column keys:

```
=> CREATE TABLE dimension(pk1 INTEGER NOT NULL, pk2 INTEGER NOT NULL);=> ALTER TABLE dime
nsion ADD PRIMARY KEY (pk1, pk2);
=> CREATE TABLE fact (fk1 INTEGER NOT NULL, fk2 INTEGER NOT NULL);
=> ALTER TABLE fact ADD FOREIGN KEY (fk1, fk2) REFERENCES dimension (pk1, pk2);
```

To join tables using compound keys, you must connect two join predicates with a Boolean AND operator. For example:

```
=> SELECT * FROM fact f JOIN dimension d ON f.fk1 = d.pk1 AND f.fk2 = d.pk2;
```

You can write queries with expressions that contain the <=> operator for NULL=NULL joins.

```
=> SELECT * FROM fact JOIN dim ON fact.x <=> dim.y;
```

The <=> operator performs an equality comparison like the = operator, but it returns true, instead of NULL, if both operands are NULL, and false, instead of NULL, if one operand is NULL.

```
=> SELECT 1 <=> 1, NULL <=> NULL, 1 <=> NULL;
 ?column? | ?column? | ?column?
----------+----------+----------
 t        | t        | f
(1 row)
```

Compare the <=> operator to the = operator:

```
=> SELECT 1 = 1, NULL = NULL, 1 = NULL;
 ?column? | ?column? | ?column?
----------+----------+----------
 t        |          |
(1 row)
```

Note: Writing NULL=NULL joins on primary key/foreign key combinations is not an optimal choice because PK/FK columns are usually defined as NOT NULL.

When composing joins, it helps to know in advance which columns contain null values. An employee's hire date, for example, would not be a good choice because it is unlikely hire date would be omitted. An hourly rate column, however, might work if some employees are paid hourly and some are salaried. If you are unsure about the value of columns in a given table and want to check, type the command:

```
=> SELECT COUNT(*) FROM tablename WHERE columnname IS NULL;
```

Natural Joins

A natural join is just a join with an implicit join predicate. Natural joins can be inner, left outer, right outer, or full outer joins and take the following form:

```
SELECT <column list> FROM <left-joined table>
NATURAL [ INNER | LEFT OUTER | RIGHT OUTER | FULL OUTER ] JOIN <right-joined table>
```

Natural joins are, by default, natural *inner* joins; however, there can also be natural (left/right) outer joins. The primary difference between an inner and natural join is that inner joins have an explicit join condition, whereas the natural join's conditions are formed by matching all pairs of columns in the tables that have the same name and compatible data types, making natural joins equi-joins because join condition are equal between common columns. (If the data types are incompatible, HP Vertica returns an error.)

Note: The data type coercion chart lists the data types that can be cast to other data types. If one data type can be cast to the other, those two data types are compatible.

The following query is a simple natural join between tables T1 and T2 when the T2 column `val` is greater than 5:

```
=> SELECT * FROM T1 NATURAL JOIN T2 WHERE T2.val > 5;
```

The following example shows a natural join between the `store_sales_fact` table and the `product_dimension` table with columns of the same name, `product_key` and `product_version`:

```
=> SELECT product_description, store.store_sales_fact.*
   FROM store.store_sales_fact, public.product_dimension
   WHERE store.store_sales_fact.product_key = public.product_dimension.product_key
   AND store.store_sales_fact.product_version = public.product_dimension.product_version;
```

The following three queries return the same result expressed as a basic query, an inner join, and a natural join. at the table expressions are equivalent only if the common attribute in the `store_sales_fact` table and the `store_dimension` table is `store_key`. If both tables have a column named `store_key`, then the natural join would also have a `store_sales_fact.store_key = store_dimension.store_key` join condition. Since the results are the same in all three instances, they are shown in the first (basic) query only:

```
=> SELECT store_name FROM store.store_sales_fact, store.store_dimension
   WHERE store.store_sales_fact.store_key = store.store_dimension.store_key
   AND store.store_dimension.store_state = 'MA' ORDER BY store_name;
```

```
 store_name
------------
 Store11
 Store128
 Store178
 Store66
 Store8
 Store90
(6 rows)
```

The query written as an inner join:

```
=> SELECT store_name FROM store.store_sales_fact
   INNER JOIN store.store_dimension
   ON (store.store_sales_fact.store_key = store.store_dimension.store_key)
   WHERE store.store_dimension.store_state = 'MA' ORDER BY store_name;
```

In the case of the natural join, the join predicate appears implicitly by comparing all of the columns in both tables that are joined by the same column name. The result set contains only one column representing the pair of equally-named columns.

```
=> SELECT store_name FROM store.store_sales_fact
   NATURAL JOIN store.store_dimension
   WHERE store.store_dimension.store_state = 'MA' ORDER BY store_name;
```

Cross Joins

Cross joins are the simplest joins to write, but they are not usually the fastest to run because they consist of all possible combinations of two tables' records. Cross joins contain no join condition and return what is known as a Cartesian product, where the number of rows in the result set is equal to the number of rows in the first table multiplied by the number of rows in the second table.

The following query returns all possible combinations from the the promotion table and the store sales table:

```
=> SELECT * FROM promotion_dimension CROSS JOIN store.store_sales_fact;
```

Since this example returns over 600 million records, many cross join results can be extremely large and difficult to manage. Cross joins can be useful, however, such as when you want to return a single-row result set.

Tip: Tip: Filter out unwanted records in a cross with `WHERE` clause join predicates:

```
=> SELECT * FROM promotion_dimension p   CROSS JOIN store.store_sales_fact f
   WHERE p.promotion_key LIKE f.promotion_key;
```

For details on what qualifies as a join predicate, see Pre-Join Projections and Join Predicates.

HP recommends that you do not write implicit cross joins (coma-separated tables in the `FROM` clause). These queries could imply accidental omission of a join predicate.

The following statement is an example of an implicit cross join:

```
=> SELECT * FROM promotion_dimension , store.store_sales_fact;
```

If you intend is to run a cross join, write an explicit cross join query using `CROSS JOIN` keywords, such as in the following statement:

```
=> SELECT * FROM promotion_dimension CROSS JOIN  store.store_sales_fact;
```

Examples

The following example creates two small tables and their superprojections and then runs a cross join on the tables:

```
=> CREATE TABLE employee(employee_id INT, employee_fname VARCHAR(50));
=> CREATE TABLE department(dept_id INT, dept_name VARCHAR(50));
=> INSERT INTO employee VALUES (1, 'Andrew');
=> INSERT INTO employee VALUES (2, 'Priya');
=> INSERT INTO employee VALUES (3, 'Michelle');
=> INSERT INTO department VALUES (1, 'Engineering');
=> INSERT INTO department VALUES (2, 'QA');
=> SELECT * FROM employee CROSS JOIN department;
```

In the result set, the cross join retrieves records from the first table and then creates a new row for every row in the 2nd table. It then does the same for the next record in the first table, and so on.

```
 employee_id | employee_name | dept_id | dept_name
-------------+---------------+---------+-----------
           1 | Andrew        |       1 |  Engineering
           2 | Priya         |       1 |  Engineering
           3 | Michelle      |       1 |  Engineering
           1 | Andrew        |       2 |  QA
           2 | Priya         |       2 |  QA
           3 | Michelle      |       2 |  QA
(6 rows)
```

Outer Joins

Outer joins extend the functionality of inner joins by letting you preserve rows of one or both tables that do not have matching rows in the non-preserved table. Outer joins take the following form:

```
SELECT <column list>
FROM <left-joined table>
[ LEFT | RIGHT | FULL ] OUTER JOIN <right-joined table>
ON <join condition>
```

Note: Omitting the keyword `OUTER` from your statements does not affect results of left and right joins. `LEFT OUTER JOIN` and `LEFT JOIN` perform the same operation and return the same results.

Left Outer Joins

A left outer join returns a complete set of records from the left-joined (preserved) table `T1`, with matched records, where available, in the right-joined (non-preserved) table `T2`. Where HP Vertica finds no match, it extends the right side column (`T2`) with null values.

```
=> SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.x = T2.x;
```

To exclude the non-matched values from T2, write the same left outer join, but filter out the records you don't want from the right side by using a `WHERE` clause:

```
=> SELECT * FROM T1 LEFT OUTER JOIN T2
   ON T1.x = T2.x WHERE T2.x IS NOT NULL;
```

The following example uses a left outer join to enrich telephone call detail records with an incomplete numbers dimension. It then filters out results that are known not to be from Massachusetts:

```
=> SELECT COUNT(*) FROM calls LEFT OUTER JOIN numbers
   ON calls.to_phone = numbers.phone WHERE NVL(numbers.state, '') <> 'MA';
```

Right Outer Joins

A right outer join returns a complete set of records from the right-joined (preserved) table, as well as matched values from the left-joined (non-preserved) table. If HP Vertica finds no matching records from the left-joined table (`T1`), `NULL` values appears in the `T1` column for any records with no matching values in `T1`. A right join is, therefore, similar to a left join, except that the treatment of the tables is reversed.

```
=> SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T1.x = T2.x;
```

The above query is equivalent to the following query, where `T1 RIGHT OUTER JOIN T2 = T2 LEFT OUTER JOIN T1`.

```
=> SELECT * FROM T2 LEFT OUTER JOIN T1 ON T2.x = T1.x;
```

The following example identifies customers who have *not* placed an order:

```
=> SELECT customers.customer_id FROM orders RIGHT OUTER JOIN customers
   ON orders.customer_id = customers.customer_id
   GROUP BY customers.customer_id HAVING COUNT(orders.customer_id) = 0;
```

Full Outer Joins

A full outer join returns results for both left and right outer joins. The joined table contains all records from both tables, including nulls (missing matches) from either side of the join. This is useful if you want to see, for example, each employee who is assigned to a particular department and each department that has an employee, but you also want to see all the employees who are not assigned to a particular department, as well as any department that has no employees:

```
=> SELECT employee_last_name, hire_date FROM  employee_dimension emp
   FULL OUTER JOIN department dept ON emp.employee_key = dept.department_key;
```

Notes

HP Vertica also supports joins where the outer (preserved) table or subquery is replicated on more than one node and the inner (non-preserved) table or subquery is segmented across more than one node. For example, in the following query, the fact table, which is almost always segmented, appears on the non-preserved side of the join, and it is allowed:

```
=> SELECT sales_dollar_amount, transaction_type, customer_name
    FROM store.store_sales_fact f RIGHT JOIN customer_dimension d
   ON f.customer_key = d.customer_key;
 sales_dollar_amount | transaction_type | customer_name
---------------------+------------------+---------------
                 252 | purchase         | Inistar
                 363 | purchase         | Inistar
                 510 | purchase         | Inistar
                -276 | return           | Foodcorp
                 252 | purchase         | Foodcorp
                 195 | purchase         | Foodcorp
                 290 | purchase         | Foodcorp
                 222 | purchase         | Foodcorp
                     |                  | Foodgen
                     |                  | Goldcare
(10 rows
```

Range Joins

HP Vertica provides performance optimizations for <, <=, >, >=, and `BETWEEN` predicates in join `ON` clauses. These optimizations are particularly useful when a column from one table is restricted to be in a range specified by two columns of another table.

Key Ranges

Multiple, consecutive key values can map to the same dimension values. Consider, for example, a table of IPv4 addresses and their owners. Because large subnets (ranges) of IP addresses could belong to the same owner, this dimension can be represented as:

```
=> CREATE TABLE ip_owners(
      ip_start INTEGER,
      ip_end INTEGER,
      owner_id INTEGER);
=> CREATE TABLE clicks(
      ip_owners INTEGER,
      dest_ip INTEGER);
```

A query that associates a click stream with its destination can use a join similar to the following, which takes advantage of the range optimization:

```
=> SELECT owner_id, COUNT(*) FROM clicks JOIN ip_owners
   ON clicks.dest_ip BETWEEN ip_start AND ip_end
   GROUP BY owner_id;
```

Slowly-Changing Dimensions

Sometimes there are multiple dimension ranges, each relevant over a different time period. For example, stocks might undergo splits (and reverse splits), and the price or volume of two trades might not be directly comparable without taking this into account. A “split factor” can be defined, which accounts for these events through time:

```
=> CREATE TABLE splits(
      symbol VARCHAR(10),
      start DATE,
      "end" DATE,
      split_factor FLOAT);
```

A join with an optimized range predicate can then be used to match each trade with the effective split factor:

```
=> SELECT trades.symbol, SUM(trades.volume * splits.split_factor)
   FROM trades JOIN splits
   ON trades.symbol = splits.symbol AND trades.tdate between splits.start AND splits.end
   GROUP BY trades.symbol;
```

Notes

- Operators <, <=, >, >=, or `BETWEEN` must appear as top-level conjunctive predicates for range join optimization to be effective, as shown in the following examples:

  The following example query is optimized because `BETWEEN` is the only predicate:

  ```
  => SELECT COUNT(*) FROM fact JOIN dim
      ON fact.point BETWEEN dim.start AND dim.end;
  ```

  This next example uses comparison operators as top-level predicates (within `AND`):

```
=> SELECT COUNT(*) FROM fact JOIN dim
    ON fact.point > dim.start AND fact.point < dim.end;
```

The following is optimized because `BETWEEN` is a top-level predicate (within `AND`):

```
=> SELECT COUNT(*) FROM fact JOIN dim
   ON (fact.point BETWEEN dim.start AND dim.end) AND fact.c <> dim.c;
```

The following query is not optimized because `OR` is the top-level predicate (disjunctive):

```
=> SELECT COUNT(*) FROM fact JOIN dim
   ON (fact.point BETWEEN dim.start AND dim.end) OR dim.end IS NULL;
```

- Expressions are optimized in range join queries in many cases.
- If range columns can have `NULL` values indicating that they are open-ended, it is possible to use range join optimizations by replacing nulls with very large or very small values:

```
=> SELECT COUNT(*) FROM fact JOIN dim
   ON fact.point BETWEEN NVL(dim.start, -1) AND NVL(dim.end, 1000000000000);
```

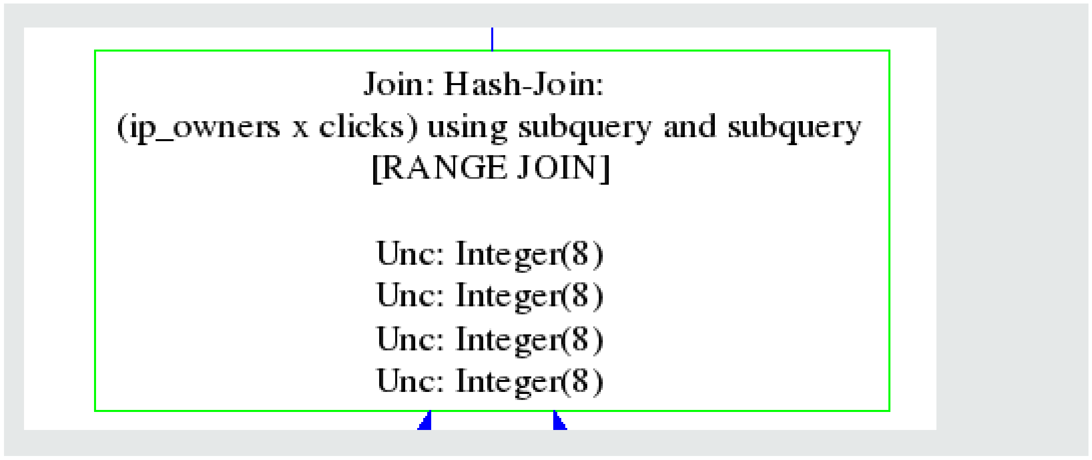
- If there is more than one set of ranging predicates in the same `ON` clause, the order in which the predicates are specified might impact the effectiveness of the optimization:

```
=> SELECT COUNT(*) FROM fact JOIN dim   ON fact.point1 BETWEEN dim.start1 AND dim.end1
   AND fact.point2 BETWEEN dim.start2 AND dim.end2;
```

The optimizer chooses the first range to optimize, so write your queries so that the range you most want optimized appears first in the statement.

- The use of the range join optimization is not directly affected by any characteristics of the physical schema; no schema tuning is required to benefit from the optimization.
- The range join optimization can be applied to joins without any other predicates, and to `HASH` or `MERGE` joins.
- To determine if an optimization is in use, search for `RANGE` in the `EXPLAIN` plan. For example:

```
 => EXPLAIN SELECT owner_id, COUNT(*) FROM clicks JOIN ip_owners
     ON clicks.dest_ip BETWEEN ip_start AND ip_end GROUP BY owner_id;
```



Pre-Join Projections and Join Predicates

HP Vertica can use pre-join projections when queries contain equi-joins between tables that contain all foreign key-primary key (FK-PK) columns in the equality predicates.

If you use **pre-join projections** in queries, the join in the input query becomes an inner join due to FK-PK constraints, so the second predicate in the example that follows (`AND f.id2 = d.id2`) is just extra. HP Vertica runs queries using pre-join projections only if the query contains a superset of the join predicates in the pre-join projection. In the following example, as long as the pre-join projection contains `f.id = d.id`, the pre-join can be used, even with the presence of `f.id2 = d.id2`.

```
=> SELECT * FROM fact f JOIN dim d ON  f.id = d.id AND f.id2 = d.id2;
```

Note: HP Vertica uses a maximum of one pre-join projection per query. More than one pre-join projection might appear in a query plan, but at most, one will have been used to replace the join that would be computed with the precomputed pre-join. Any other pre-join projections are used as regular projections to supply records from a particular table.

Examples

The following is an example of a pre-join projection schema with a single-column constraint called `customer_key`. The first sequence of statements creates a customer table in the public schema and a `store_sales` table in the `store` schema. The dimension table has one primary key, and the fact table has a foreign key that references the dimension table's primary key.

```
=> CREATE TABLE public.customer_dimension (
       customer_key integer,
       annual_income integer,
       largest_bill_amount integer);
=> CREATE TABLE store.store_sales_fact (
       customer_key integer,
       sales_quantity integer,
       sales_dollar_amount integer);
=> ALTER TABLE public.customer_dimension
   ADD CONSTRAINT pk_customer_dimension PRIMARY KEY (customer_key);
=> ALTER TABLE store.store_sales_fact
   ADD CONSTRAINT fk_store_sales_fact FOREIGN KEY (customer_key)
   REFERENCES public.customer_dimension (customer_key);
=> CREATE PROJECTION p1 (
       customer_key,
       annual_income,
       largest_bill_amount)
   AS SELECT * FROM public.customer_dimension UNSEGMENTED ALL NODES;
=> CREATE PROJECTION p2 (
       customer_key,
       sales_quantity,
       sales_dollar_amount)
   AS SELECT * FROM store.store_sales_fact UNSEGMENTED ALL NODES;
```

The following command creates the pre-join projection:

```
=> CREATE PROJECTION pp (
       cust_customer_key,
       cust_annual_income,
       cust_largest_bill_amount,
       fact_customer_key,
       fact_sales_quantity,
       fact_sales_dollar_amount)
   AS SELECT * FROM public.customer_dimension cust, store.store_sales_fact fact
   WHERE cust.customer_key = fact.customer_key ORDER BY cust.customer_key;
```

The pre-join projection contains columns from both tables and has a join predicate between `customer_dimension` and `store_sales_fact` along the FK-PK (primary key-foreign key) constraints defined on the tables.

The following query uses a pre-join projection because the join predicates match the pre-join projection's predicates exactly:

```
=> SELECT COUNT(*) FROM public.customer_dimension INNER JOIN store.store_sales_fact
   ON public.customer_dimension.customer_key = store.store_sales_fact.customer_key;
 count
-------
 10000
(1 row)
```

Join Notes and Restrictions

The following list summarizes the notes and restrictions for joins in HP Vertica:

- Inner joins are are commutative and associative, which means you can specify the tables in any order you want, and the results do not change.

- If you omit the INNER keyword, the join is still an inner join, the most commonly used type of join.
- Join conditions that follow the ON keyword generally can contain many predicates connected with Boolean AND, OR, or NOT predicates.
- For best performance, do not join on any LONG VARBINARY and LONG VARCHAR columns.
- You can also use inner join syntax to specify joins for pre-join projections. See Pre-Join Projections and Join Predicates.

- HP Vertica supports any arbitrary join expression with both matching and non-matching column values; for example:

```
=> SELECT * FROM fact JOIN dim ON fact.x = dim.x;
=> SELECT * FROM fact JOIN dim ON fact.x > dim.y;
=> SELECT * FROM fact JOIN dim ON fact.x <= dim.y;
=> SELECT * FROM fact JOIN dim ON fact.x <> dim.y;
=> SELECT * FROM fact JOIN dim ON fact.x <=> dim.y;
```

- HP Vertica permits joins between tables with compound (multiple-column) primary and foreign keys, as long as you connect the two join predicates with a Boolean AND operator.
- You can write queries with expressions that contain the <=> operator for NULL=NULL joins.

```
=> SELECT * FROM fact JOIN dim ON fact.x <=> dim.y;
```

  The <=> operator performs an equality comparison like the = operator, but it returns true, instead of NULL, if both operands are NULL, and false, instead of NULL, if one operand is NULL.

- HP recommends that you do not write implicit cross joins (such as tables named in the FROM clause separated by commas). Such queries could imply accidental omission of a join predicate. If your intent is to run a cross join, write explicit CROSS JOIN syntax.
- HP Vertica supports joins where the outer (preserved) table or subquery is replicated on more than one node and the inner (non-preserved) table or subquery is segmented across more than one node.
- HP Vertica uses a maximum of one pre-join projection per query. More than one pre-join projection might appear in a query plan, but at most, one will have been used to replace the join that would be computed with the precomputed pre-join. Any other pre-join projections are used as regular projections to supply records from a particular table.

About Running Database Designer Programmatically

If you have been granted the DBDUSER role and have enabled the role, you can access Database Designer functionality programmatically. In previous releases, Database Designer was available only via the Administration Tools. Using the DESIGNER_* command-line functions, you can perform the following Database Designer tasks:

- Create a comprehensive or incremental design.
- Add tables and queries to the design.
- Set the optimization objective to prioritize for query performance or storage footprint.
- Assign a weight to each query.
- Assign the K-safety value to a design.
- Analyze statistics on the design tables.
- Create the script that contains the DDL statements that create the design projections.
- Deploy the database design.
- Specify that all projections in the design be segmented.
- Populate the design.
- Cancel a running design.
- Wait for a running design to complete.
- Deploy a design automatically.
- Drop database objects from one or more completed or terminated designs.

Important: When you grant the DBDUSER role, make sure to associate a resource pool with that user to manage resources during Database Designer runs. Multiple users can run Database Designer concurrently without interfering with each other or using up all the cluster resources. When a user runs Database Designer, either using the Administration Tools or programmatically, its execution is mostly contained by the user's resource pool, but may spill over into some system resource pools for less-intensive tasks.

For detailed information about each function, see Database Designer Functions in the SQL Reference Manual.

When to Run Database Designer Programmatically

Run Database Designer programmatically when you want to:

- Optimize performance on tables you own.
- Create or update a design without the involvement of the superuser.
- Add individual queries and tables, or add data to your design and then rerun Database Designer to update the design based on this new information.
- Customize the design.
- Use recently executed queries to set up your database to run Database Designer automatically on a regular basis.
- Assign each design query a *query weight* that indicates the importance of that query in creating the design. Assign a higher weight to queries that you run frequently so that Database Designer prioritizes those queries in creating the design.

Categories Database Designer Functions

You can run Database Designer functions in vsql:

Setup Functions

This function directs Database Designer to create a new design:

- DESIGNER_CREATE_DESIGN

Configuration Functions

The following functions allow you to specify properties of a particular design:

- DESIGNER_DESIGN_PROJECTION_ENCODINGS
- DESIGNER_SET_DESIGN_KSAFETY
- DESIGNER_SET_OPTIMIZATION_OBJECTIVE
- DESIGNER_SET_DESIGN_TYPE
- DESIGNER_SET_PROPOSED_UNSEGMENTED_PROJECTIONS
- DESIGNER_SET_ANALYZE_CORRELATIONS_MODE

Input Functions

The following functions allow you to add tables and queries to your Database Designer design:

- DESIGNER_ADD_DESIGN_QUERIES
- DESIGNER_ADD_DESIGN_QUERIES_FROM RESULTS
- DESIGNER_ADD_DESIGN_QUERY
- DESIGNER_ADD_DESIGN_TABLES

Invocation Functions

These functions populate the Database Designer workspace and create design and deployment scripts. You can also analyze statistics, deploy the design automatically, and drop the workspace after the deployment:

- DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY
- DESIGNER_WAIT_FOR_DESIGN

Output Functions

The following functions display information about projections and scripts that the Database Designer created:

- DESIGNER_OUTPUT_ALL_DESIGN_PROJECTIONS
- DESIGNER_OUTPUT_DEPLOYMENT_SCRIPT

Cleanup Functions

The following functions cancel any running Database Designer operation or drop a Database Designer design and all its contents:

- DESIGNER_CANCEL_POPULATE_DESIGN
- DESIGNER_DROP_DESIGN
- DESIGNER_DROP_ALL_DESIGNS

Privileges for Running Database Designer Functions

If they have been granted the DBDUSER role, non-DBADMIN users can run Database Designer using the functions described in Categories of Database Designer Functions. Non-DBADMIN users cannot run Database Designer using Administration Tools, even if they have been assigned the DBDUSER role.

To grant the DBDUSER role:

1. The DBADMIN user must grant the DBDUSER role:

   ```
   => GRANT DBDUSER TO <username>;
   ```

   This role persists until the DBADMIN revokes it.

   IMPORTANT: When you grant the DBDUSER role, make sure to associate a resource pool with that user to manage resources during Database Designer runs. Multiple users can run Database Designer concurrently without interfering with each other or using up all the cluster resources. When a user runs Database Designer, either using the Administration Tools or programmatically, its execution is mostly contained by the user's resource pool, but may spill over into some system resource pools for less-intensive tasks.

2. For a user to run the Database Designer functions, one of the following must happen first:

   - The user must enable the DBDUSER role:

     ```
     => SET ROLE DBDUSER;
     ```

   - The superuser must add DBDUSER as the default role:

     ```
     => ALTER USER <username> DEFAULT ROLE DBDUSER;
     ```

DBDUSER Capabilities and Limitations

The DBDUSER role has the following capabilities and limitations:

- A DBDUSER can change K-safety for their own designs, but they cannot change the system K-safety value. The DBDUSER can set the K-safety to a value less than or equal to the system K-safety value, but is limited to a value of 0, 1, or 2.
- A DBDUSER cannot explicitly change the ancient history mark (AHM), even during deployment of their design.

DBDUSER Privileges

When you create a design, you automatically have privileges to manipulate the design. Other tasks may require that the DBDUSER have additional privileges:

To...	DBDUSER must have...
Add tables to a design	• USAGE privilege on the design table schema • OWNER privilege on the design table
Add a single design query to the design	• Privilege to execute the design query
Add a query file to the design	• Read privilege on the storage location that contains the query file • Privilege to execute all the queries in the file
Add queries from the result of a user query to the design	• Privilege to execute the user query • Privilege to execute each design query retrieved from the results of the user query
Create the design and deployment scripts	• WRITE privilege on the storage location of the design script • WRITE privilege on the storage location of the deployment script

Workflow for Running Database Designer Programmatically

The following example shows the steps you take to create a design by running Database Designer programmatically.

Note: Be sure to back up the existing design using the EXPORT_CATALOG (on page 1) function before running the Database Designer functions on an existing schema. You must explicitly back up the current design when using Database Designer to create a new comprehensive design.

Before you run this example, you should have the DBDUSER role, and you should have enabled that role using the SET ROLE DBDUSER command:

1. Create a table in the public schema:

```
=> CREATE TABLE T(
   x INT,
   y INT,
   z INT,
   u INT,
   v INT,
   w INT PRIMARY KEY
   );
```

2. Add data to the table:

```
\! perl -e 'for ($i=0; $i<100000; ++$i)   {printf("%d, %d, %d, %d, %d, %d\n", $i/1000
0, $i/100, $i/10, $i/2, $i, $i);}'
   | vsql -c "COPY T FROM STDIN DELIMITER ',' DIRECT;"
```

3. Create a second table in the public schema:

```
=> CREATE TABLE T2(
   x INT,
   y INT,
   z INT,
   u INT,
   v INT,
   w INT PRIMARY KEY
   );
```

4. Copy the data from table `T1` to table `T2` and commit the changes:

```
=> INSERT /*+DIRECT*/ INTO T2 SELECT * FROM T;
=> COMMIT;
```

5. Create a new design:

```
=> SELECT DESIGNER_CREATE_DESIGN('my_design');
```

   This command creates the following system tables in the V_MONITOR schema:

   - DESIGNS
   - DESIGN_TABLES
   - DEPLOYMENT_PROJECTIONS
   - DEPLOYMENT_PROJECTION_STATEMENTS
   - DESIGN_QUERIES
   - OUTPUT_DEPLOYMENT_STATUS
   - OUTPUT_EVENT_HISTORY

6. Add tables from the public schema to the design :

```
=> SELECT DESIGNER_ADD_DESIGN_TABLES('my_design', 'public.t');
=> SELECT DESIGNER_ADD_DESIGN_TABLES('my_design', 'public.t2');
```

These commands populate the DESIGN_TABLES system table.

7. Create a file named `queries.txt` in `/tmp/examples`, or another directory where you have READ and WRITE privileges. Add the following two queries in that file and save it. Database Designer uses these queries to create the design:

```
SELECT DISTINCT T2.u FROM T JOIN T2 ON T.z=T2.z-1 WHERE T2.u > 0;
SELECT DISTINCT w FROM T;
```

8. Add the queries file to the design and display the results—the numbers of accepted queries, non-design queries, and unoptimizable queries:

```
=> SELECT DESIGNER_ADD_DESIGN_QUERIES
     ('my_design',
     '/tmp/examples/queries.txt',
     'true'
     );
```

The results show that both queries were accepted:

```
Number of accepted queries                     =2
Number of queries referencing non-design tables =0
Number of unsupported queries                  =0
Number of illegal queries                      =0
```

The DESIGNER_ADD_DESIGN_QUERIES function populates the DESIGN_QUERIES system table.

9. Set the design type to **comprehensive**. (This is the default.) A comprehensive design creates an initial or replacement design for all the design tables:

```
=> SELECT DESIGNER_SET_DESIGN_TYPE('my_design', 'comprehensive');
```

10. Set the optimization objective to **query**. This setting creates a design that focuses on faster query performance, which might recommend additional projections. These projections could result in a larger database storage footprint:

```
=> SELECT DESIGNER_SET_OPTIMIZATION_OBJECTIVE('my_design', 'query');
```

11. Create the design and save the design and deployment scripts in `/tmp/examples`, or another

directory where you have READ and WRITE privileges. The following command:

- Analyzes statistics
- Doesn't deploy the design.
- Doesn't drop the design after deployment.
- Stops if it encounters an error.

```
=> SELECT DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY
   ('my_design',
    '/tmp/examples/my_design_projections.sql',
    '/tmp/examples/my_design_deploy.sql',
    'True',
    'False',
    'False',
    'False'
    );
```

This command populates the following system tables:

- DEPLOYMENT_PROJECTION_STATEMENTS
- DEPLOYMENT_PROJECTIONS
- OUTPUT_DEPLOYMENT_STATUS

12. Examine the status of the Database Designer run to see what projections Database Designer recommends. In the `deployment_projection_name` column:

    - `rep` indicates a replicated projection
    - `super` indicates a superprojection

    The `deployment_status` column is `pending` because the design has not yet been deployed.

    For this example, Database Designer recommends four projections:

```
=> \x
Expanded display is on.
=> SELECT * FROM OUTPUT_DEPLOYMENT_STATUS;
-[ RECORD 1 ]--------------+-----------------------------
deployment_id              | 45035996273795970
deployment_projection_id   | 1
deployment_projection_name | T_DBD_1_rep_my_design
deployment_status          | pending
error_message              | N/A
-[ RECORD 2 ]--------------+-----------------------------
```

```
deployment_id               | 45035996273795970
deployment_projection_id    | 2
deployment_projection_name  | T2_DBD_2_rep_my_design
deployment_status           | pending
error_message               | N/A
-[ RECORD 3 ]--------------+-----------------------------
deployment_id               | 45035996273795970
deployment_projection_id    | 3
deployment_projection_name  | T_super
deployment_status           | pending
error_message               | N/A
-[ RECORD 4 ]--------------+-----------------------------
deployment_id               | 45035996273795970
deployment_projection_id    | 4
deployment_projection_name  | T2_super
deployment_status           | pending
error_message               | N/A
```

13. View the script `/tmp/examples/my_design_deploy.sql` to see how these projections are created when you run the deployment script. In this example, the script also assigns the encoding schemes RLE and COMMONDELTA_COMP to columns where appropriate.

14. Deploy the design from the directory where you saved it:

```
=> \i /tmp/examples/my_design_deploy.sql
```

15. Now that the design is deployed, delete the design:

```
=> SELECT DESIGNER_DROP_DESIGN('my_design');
```

Using SQL Analytics

HP Vertica analytics are SQL functions based on the ANSI 99 standard. These functions handle complex analysis and reporting tasks such as:

- Rank the longest-standing customers in a particular state
- Calculate the moving average of retail volume over a specified time
- Find the highest score among all students in the same grade
- Compare the current sales bonus each salesperson received against his or her previous bonus

Analytic functions return aggregate results but they do not group the result set. They return the group value multiple times, once per record.

You can sort these group values, or partitions, using a window `ORDER BY` clause, but the order affects only the function result set, not the entire query result set. This ordering concept is described more fully later.

Notes

HP Vertica supports a full list of analytic functions, including:

- FIRST_VALUE(*arguments*): Allows the selection of the first value of a table or partition without having to use a self-join
- MEDIAN(arguments): Returns the middle value from a set of values
- NTILE(value): Equally divides the data set into a {value} number of subsets (buckets)
- RANK(): Assigns a rank to each row returned from the query with respect to the other ordered rows
- STDDEV(*arguments*): Computes the statistical sample standard deviation of the current row with respect to a group of rows
- AVG(*arguments*): Computes an average of an expression in a group of rows

For additional details, see Analytic Functions in the SQL Reference Manual.

How Analytic Functions Work

Analytic functions take the following form:

```
analytic_function ( arguments ) OVER( analytic_clause  )

        [ window_partition_clause ]
        [ window_order_clause { ASC | DESC }
            { NULLS { FIRST | LAST | AUTO } } ]
        [ window_frame_clause ] )
```

```
{ ROWS | RANGE }
{
  {
      BETWEEN
      { UNBOUNDED PRECEDING
      | CURRENT ROW
      | constant-value { PRECEDING | FOLLOWING }
      }
      AND
      { UNBOUNDED FOLLOWING
      | CURRENT ROW
      | constant-value  { PRECEDING | FOLLOWING }
      }
  }
|
  {
      { UNBOUNDED PRECEDING
      | CURRENT ROW
      | constant-value  PRECEDING
      }
  }
}
```

Evaluation Order

Analytic functions conform to the following phases of execution:

1. Take the input rows.

   Analytic functions are computed after WHERE, GROUP BY, HAVING clause operations, and joins are performed on the query.

2. Group input rows according to the PARTITION BY clause.

   The analytic PARTITION BY clause (called the window_partition_clause) is different from table partition expressions. See Working with Table Partitions in the Administrator's Guide for details.

3. Order rows within groups (partitions) according to ORDER BY clause.

   The analytic ORDER BY clause (called the window_order_clause) is different from the SQL ORDER BY clause. If the query has a final ORDER BY clause (outside the OVER() clause), the final results are ordered according by the SQL ORDER BY clause, not the window_order_ clause. See NULL Placement By Analytic Functions and Designing Tables to Minimize Run-Time Sorting of NULL Values in Analytic Functions in the Programmer's Guide for additional information about sort computation.

4. Compute some function for each row.

Notes

Analytic functions:

- Require the OVER() clause. However, depending on the function, the `window_frame_clause` and `window_order_clause` might not apply. For example, when used with analytic aggregate functions like SUM(*x*), you can use the OVER() clause without supplying any of the windowing clauses; in this case, the aggregate returns the same aggregated value for each row of the result set.
- Are allowed only in the SELECT and ORDER BY clauses.
- Can be used in a subquery or in the parent query but *cannot* be nested; for example, the following query is not allowed:

  ```
  => SELECT MEDIAN(RANK() OVER(ORDER BY sal) OVER()).
  ```

- WHERE, GROUP BY and HAVING operators are technically not part of the analytic function; however, they determine on which rows the analytic functions operate.

Note: Several examples throughout this section refer back to this example's `allsales` table schema.

Example: Calculation of Median Value

A median is a numerical value that separates the higher half of a sample from the lower half. For example, you can retrieve the median of a finite list of numbers by arranging all observations from lowest value to highest value and then picking the middle one.

If there is an even number of observations, then there is no single middle value; the median is then defined to be the mean (average) of the two middle values.

```
CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT);
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;
```

The following analytic query returns the median value from the allsales table. Note that the median value is reported for every row in the result set:

```
=> SELECT name, sales, MEDIAN(sales) OVER () AS
   MEDIAN FROM allsales;
 name | sales | median
------+-------+--------
 G    |    10 |     20
 C    |    15 |     20
 D    |    20 |     20
 B    |    20 |     20
 F    |    40 |     20
```

```
 E    |    50 |     20
 A    |    60 |     20
(7 rows)
```

Additional MEDIAN() examples are in Window Partitioning.

See Also

- Analytic Functions

Analytic Functions Versus Aggregate Functions

Like aggregate functions, analytic functions return aggregate results, but analytics do not group the result set. Instead, they return the group value multiple times with each record, allowing further analysis.

Analytic queries also generally run faster and use fewer resources than aggregate queries.

Analytic functions	Aggregate functions
Return the same number of rows as the input	Return a single summary value
The groups of rows on which an analytic function operates are defined by window partitioning and window frame clauses	The groups of rows on which an aggregate function operates are defined by the SQL GROUP BY clause

Example

This examples illustrate the difference between aggregate functions and their analytic counterpart using table `employees` defined in the below sample schema:

```
CREATE TABLE employees(emp_no INT, dept_no INT);
INSERT INTO employees VALUES(1, 10);
INSERT INTO employees VALUES(2, 30);
INSERT INTO employees VALUES(3, 30);
INSERT INTO employees VALUES(4, 10);
INSERT INTO employees VALUES(5, 30);
INSERT INTO employees VALUES(6, 20);
INSERT INTO employees VALUES(7, 20);
INSERT INTO employees VALUES(8, 20);
INSERT INTO employees VALUES(9, 20);
INSERT INTO employees VALUES(10, 20);
INSERT INTO employees VALUES(11, 20);
COMMIT;
```

Table `employees`:

```
SELECT * FROM employees ORDER BY emp_no;
 emp_no | dept_no
--------+---------
      1 |      10
      2 |      30
      3 |      30
      4 |      10
      5 |      30
      6 |      20
      7 |      20
      8 |      20
      9 |      20
     10 |      20
     11 |      20
(11 rows)
```

Both queries below ask for the number of employees are in each department:

Aggregate query/result	**Analytics query/result**

`SELECT dept_no, COUNT(*)`
`    AS emp_count`
`FROM employees`
`GROUP BY dept_no ORDER BY`
`1;`
	`SELECT emp_no, dept_no, COUNT(*)   OVER(PARTITION BY dept_no`
	`       ORDER BY emp_no)`
	`  AS emp_count FROM employees;`

` dept_no | emp_count`
`---------+-----------`
`      10 |         2`
`      20 |         6`
`      30 |         3`
`(3 rows)`
	` emp_no | dept_no | emp_count`
	`--------+---------+-----------`
	`      1 |      10 |         1`
	`      4 |      10 |         2`
	`------------------------------`
	`      6 |      20 |         1`
	`      7 |      20 |         2`
	`      8 |      20 |         3`
	`      9 |      20 |         4`
	`     10 |      20 |         5`
	`     11 |      20 |         6`
	`------------------------------`
	`      2 |      30 |         1`
	`      3 |      30 |         2`
	`      5 |      30 |         3`
	`(11 rows)`

Aggregate function `COUNT()` returns one row per department for the number of employees in that department.	The analytic function `COUNT()` returns a count of the number of employees in each department, as well as which employee is in each department. Within each partition, the results are sorted on the emp_no column, which is specified in the OVER order by clause.

If you wanted to add the employee number to the above aggregate query, you would add the emp_no column to the GROUP BY clause. For results, you would get emp_count=1 for each row—unless the data contained employees with the same emp_no value. For example:

```
SELECT dept_no, emp_no, COUNT(*)
   AS emp_count
FROM employees
GROUP BY dept_no, emp_no ORDER BY 1, 2;
 dept_no | emp_no | emp_count
---------+--------+-----------
      10 |      1 |         1
      10 |      4 |         1
      20 |      6 |         1
      20 |      7 |         1
      20 |      8 |         1
      20 |      9 |         1
      20 |     10 |         1
      20 |     11 |         1
      30 |      2 |         1
      30 |      3 |         1
      30 |      5 |         1
(11 rows)
```

See Also

- Analytic Query Examples

The Window OVER() Clause

The OVER() clause contains what is called a window. The window defines partitioning, ordering, and framing for an analytic function—important elements that determine what data the analytic function takes as input with respect to the current row. The analytic function then operates on a query result set, which are the rows that are returned after the FROM, WHERE, GROUP BY, and HAVING clauses have been evaluated.

You can also the OVER() clause with certain analytic functions to define a moving window of data for every row within a partition.

When used with analytic aggregate functions, OVER() does not require any of the windowing clauses; in this case, the aggregate returns the same aggregated value for each row of the result set.

The `OVER()` clause must follow the analytic function, as in the following syntax:

```
ANALYTIC_FUNCTION ( arguments )
   OVER( window_partition_clause
        window_order_clause
        window_frame_clause )
```

Window Partitioning

Window partitioning is optional. When specified, the window_partition_clause divides the rows in the input based on user-provided expressions, such as aggregation functions like SUM(*x*). Window partitioning is similar to the GROUP BY clause except that it returns only one result row per input row. If you omit the window_partition_clause, all input rows are treated as a single partition.

The analytic function is computed per partition and starts over again (resets) at the beginning of each subsequent partition. The `window_partition_clause` is specified within the `OVER()` clause.

Syntax

```
OVER( window_partition_clause
      window_order_clause
      window_frame_clause )
```

Examples

The examples in this topic use the allsales schema defined in How Analytic Functions Work.

```
CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT);
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;
```

Median of Sales Within Each State

The following query uses the analytic window_partition_clause to calculate the median of sales within each state. The analytic function is computed per partition and starts over again at the beginning of the next partition.

```
=> SELECT state, name, sales, MEDIAN(sales)
      OVER (PARTITION BY state) AS median from allsales;
```

Results are grouped into partitions for MA (35) and NY (20) under the median column.

```
 state | name | sales | median
-------+------+-------+--------
 NY    | C    |    15 |     20
 NY    | B    |    20 |     20
 NY    | F    |    40 |     20
-------------------------------
 MA    | G    |    10 |     35
 MA    | D    |    20 |     35
 MA    | E    |    50 |     35
 MA    | A    |    60 |     35
(7 rows)
```

Median of Sales Among All States

This query calculates the median of total sales among states. When you use OVER() with no parameters, there is one partition, the entire input:

```
=> SELECT state, sum(sales), median(SUM(sales))
      OVER () AS median FROM allsales GROUP BY state;
 state | sum | median
-------+-----+--------
 NY    |  75 |  107.5
 MA    | 140 |  107.5
(2 rows)
```

Sales Larger Than Median (evaluation order)

Remember that analytic functions are evaluated *after* all other clauses except the query's final SQL ORDER BY clause. So if you were to write a query like the following, which asks for all rows with sales larger than the median, HP Vertica would return an error because the WHERE clause is applied before the analytic function and m does not yet exist

```
=> SELECT name, sales,  MEDIAN(sales) OVER () AS m
   FROM allsales WHERE sales > m;
   ERROR 2624:  Column "m" does not exist
```

You can work around this by having the '`WHERE sales > m`' predicate complete in a subquery:

```
=> SELECT * FROM
   (SELECT name, sales, MEDIAN(sales) OVER () AS m FROM allsales) sq
   WHERE sales > m;
 name | sales | m
------+-------+----
 F    |    40 | 20
 E    |    50 | 20
 A    |    60 | 20
(3 rows)
```

For additional examples, see Analytic Query Examples.

Window Ordering

Window ordering sorts the rows specified by the OVER() clause and specifies whether data is sorted in ascending or descending order as well as the placement of null values; for example: ORDER BY expr_list [ASC | DESC] [NULLS { FIRST | LAST | AUTO]. The ordering of the data affects the results.

Using ORDER BY in an OVER clause changes the default window to RANGE UNBOUNDED PRECEDING AND CURRENT ROW, which is described in Window Framing.

The following table shows the default null placement, with bold clauses to indicate what is implicit:

Ordering	Null placement
ORDER BY column1	ORDER BY a **ASC NULLS LAST**
ORDER BY column1 ASC	ORDER BY a **ASC NULLS LAST**
ORDER BY column1 DESC	ORDER BY a **DESC NULLS FIRST**

Because the window_order_clause is different from a query's final ORDER BY clause, window ordering might not guarantee the final result order; it specifies only the order within a window result set, supplying the ordered set of rows to the window_frame_clause (if present), to the analytic function, or to both. Use the SQL ORDER BY clause to guarantee ordering of the final result set. (See also NULL Placement By Analytic Functions and Designing Tables to Minimize Run-Time Sorting of NULL Values in Analytic Functions.)

Syntax

```
OVER( window_partition_clause        window_order_clause
      window_frame_clause )
```

Examples

The below examples use the the allsales table schema, defined in How Analytic Functions Work.

```
CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT);
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;
```

Example 1

In this example, the query orders the sales inside each sales partition:

```
SELECT state, sales, name, RANK()OVER (PARTI
TION BY state
  ORDER BY sales) AS RANK
FROM allsales;
 state | sales | name | RANK
-------+-------+------+----------
 MA    |    10 | G    |        1
 MA    |    20 | D    |        2
 MA    |    50 | E    |        3
 MA    |    60 | A    |        4
---------------------------------
 NY    |    15 | C    |        1
 NY    |    20 | B    |        2
 NY    |    40 | F    |        3
(7 rows)
```

Example 2

In this example, the final ORDER BY clause sorts the results by name:

```
SELECT state, sales, name, RANK()OVER (PARTI
TION by state
  ORDER BY sales) AS RANK
FROM allsales ORDER BY name;
 state | sales | name | RANK
-------+-------+------+----------
 MA    |    60 | A    |        4
 NY    |    20 | B    |        2
 NY    |    15 | C    |        1
 MA    |    20 | D    |        2
 MA    |    50 | E    |        3
 NY    |    40 | F    |        3
 MA    |    10 | G    |        1
(7 rows)
```

Window Framing

Window framing represents a unique construct, called a moving window. It defines which values in the partition are evaluated relative to the current row. You specify a window frame type by using the RANGE or ROWS keywords, described in the next topics. Both RANGE and ROWS are respective to the CURRENT ROW, which is the next row for which the analytic function computes results. As the current row advances, the window boundaries are recomputed (move) along with it, determining which rows fall into the current window.

An analytic function with a window frame specification is computed for each row based on the rows that fall into the window relative to that row. If you omit the window_frame_clause, the default window is RANGE UNBOUNDED PRECEDING AND CURRENT ROW.

Syntax

```
OVER( window_partition_clause
      window_order_clause
      window_frame_clause )
```

See window_frame_clause in the SQL Reference Manual for more detailed syntax.

Schema for Examples

The window framing examples that follow use the following `emp` table schema:

```
CREATE TABLE emp(deptno INT, sal INT, empno INT);
INSERT INTO emp VALUES(10,101,1);
INSERT INTO emp VALUES(10,104,4);
INSERT INTO emp VALUES(20,100,11);
INSERT INTO emp VALUES(20,109,7);
INSERT INTO emp VALUES(20,109,6);
INSERT INTO emp VALUES(20,109,8);
INSERT INTO emp VALUES(20,110,10);
INSERT INTO emp VALUES(20,110,9);
INSERT INTO emp VALUES(30,102,2);
INSERT INTO emp VALUES(30,103,3);
INSERT INTO emp VALUES(30,105,5);
COMMIT;


CREATE TABLE emp(deptno INT, sal INT, empno INT);
INSERT INTO emp VALUES(10,101,1);
INSERT INTO emp VALUES(10,104,4);
INSERT INTO emp VALUES(20,100,11);
INSERT INTO emp VALUES(20,109,7);
INSERT INTO emp VALUES(20,109,6);
INSERT INTO emp VALUES(20,109,8);
INSERT INTO emp VALUES(20,110,10);
INSERT INTO emp VALUES(20,110,9);
INSERT INTO emp VALUES(30,102,2);
INSERT INTO emp VALUES(30,103,3);
INSERT INTO emp VALUES(30,105,5);
COMMIT;
```

Windows with a Physical Offset (ROWS)

ROWS specifies the window as a physical offset. Using ROWS, defines a start and end point of a window by the number of rows before or after the current row. The value can be `INTEGER` data type only.

Note: : The value returned by an analytic function with a physical offset could produce **nondeterministic** results unless the ordering expression results in a unique ordering. You might have to specify multiple columns in the window_order_clause to achieve this unique ordering.

Examples

The examples on this page use the emp table schema defined in Window Framing:

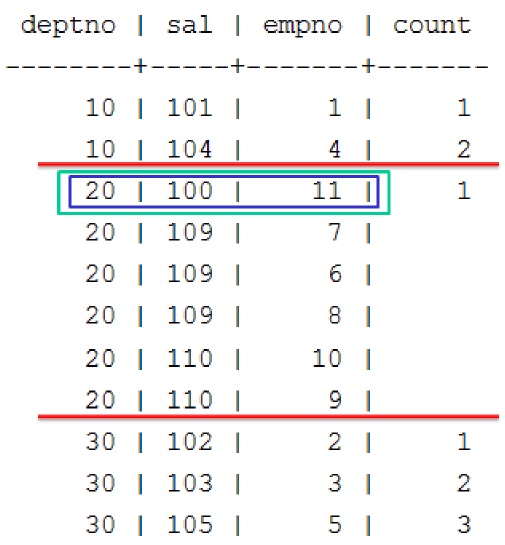
```
CREATE TABLE emp(deptno INT, sal INT, empno INT);
INSERT INTO emp VALUES(10,101,1);
INSERT INTO emp VALUES(10,104,4);
INSERT INTO emp VALUES(20,100,11);
INSERT INTO emp VALUES(20,109,7);
INSERT INTO emp VALUES(20,109,6);
INSERT INTO emp VALUES(20,109,8);
INSERT INTO emp VALUES(20,110,10);
INSERT INTO emp VALUES(20,110,9);
INSERT INTO emp VALUES(30,102,2);
INSERT INTO emp VALUES(30,103,3);
INSERT INTO emp VALUES(30,105,5);
COMMIT;
```

- The red line represents the partition
- The blue box represents the current row
- The green box represents the analytic window relative to the current row.

The following example uses the ROWS-based window for the COUNT() analytic function to return the department number, salary, and employee number with a count. The `window_frame_clause` specifies the rows between the current row and two preceding. Using ROWS in the `window_frame_clause` specifies the window as a physical offset and defines the start- and end-point of a window by the number of rows before and after the current row.

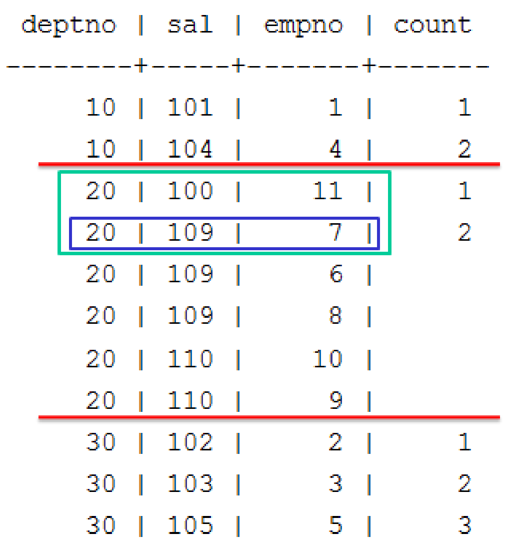
```
SELECT deptno, sal, empno, COUNT(*) OVER (PARTITION BY deptno ORDER BY sal
      ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
AS count FROM emp;
```

Notice that the partition includes department 20, and the current row and window are the same because there are no rows that precede the current row within that partition, even though the query specifies 2 preceding. The value in the count column (1) represents the number of rows in the current window.



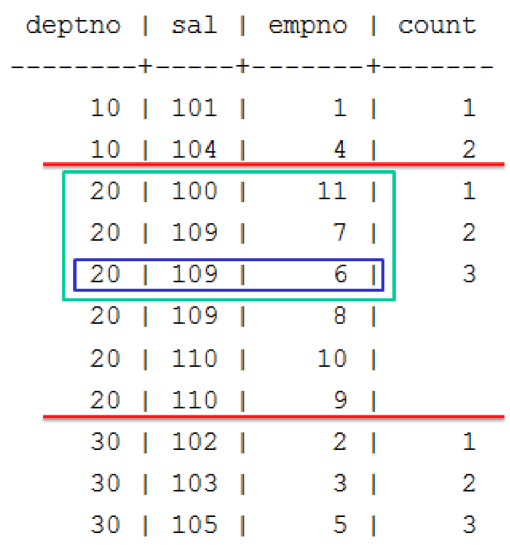
```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |
     20 | 109 |     6 |
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

As the current row moves, the window spans from 1 preceding to the current row, which is as far as it can go within the constraints of the `window_frame_clause`. COUNT() returns the number of rows in the window, even if 2 preceding is specified. In the count column, (2) includes the current row and the row above, which is the maximum the statement of 2 preceding allows within the current partition.



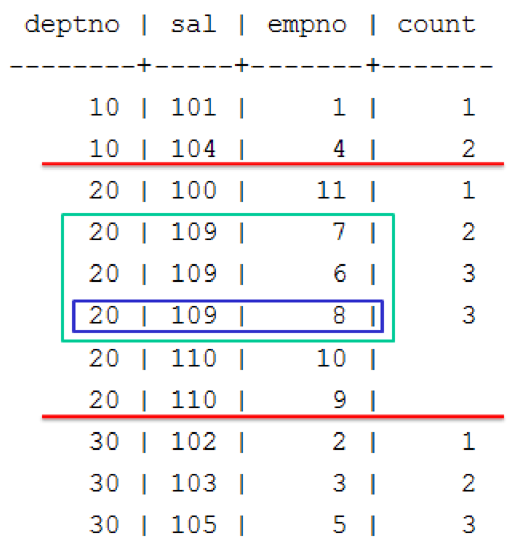
```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     2
     20 | 109 |     6 |
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

The current row moves again, and the window can now span 2 preceding the current row within the partition. The count (3) includes the number of rows in the partition (2 above + current), which is the maximum the statement of 2 preceding allows within the current partition.



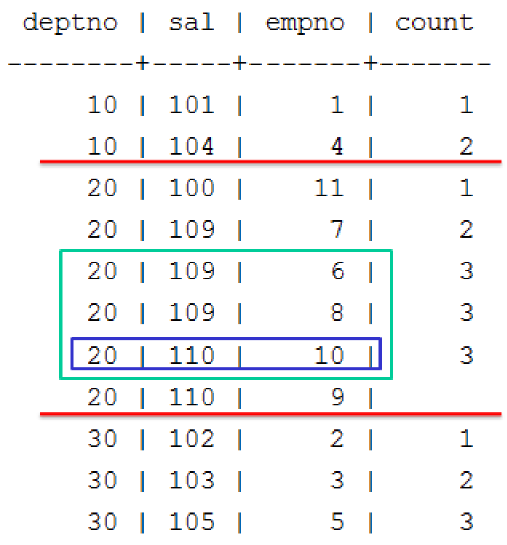
```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     2
     20 | 109 |     6 |     3
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

When the current row moves, the window also moves to maintain 2 preceding and current row. The count of 3 repeats because it represents the number of rows in the window, which has not changed:



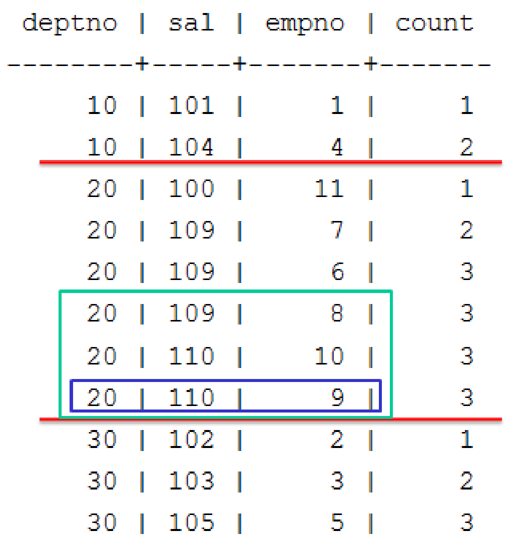
```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     2
     20 | 109 |     6 |     3
     20 | 109 |     8 |     3
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

The current row advances again, and the window is defined by the same window, so the count does not change.



```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     2
     20 | 109 |     6 |     3
     20 | 109 |     8 |     3
     20 | 110 |    10 |     3
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

In this example, the current row advances once more. Notice the current row has reached the end of the `deptno` partition.



```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     2
     20 | 109 |     6 |     3
     20 | 109 |     8 |     3
     20 | 110 |    10 |     3
     20 | 110 |     9 |     3
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

Windows with a Logical Offset (RANGE)

The RANGE keyword specifies the window as a logical offset, such as time. The range value must match the window_order_clause data type, which can be NUMERIC, DATE/TIME, FLOAT or INTEGER.

Note: The value returned by an analytic function with a logical offset is always **deterministic**.

During the analytical computation, rows are excluded or included based on the logical offset, or value (RANGE). relative to the current row, which is always the reference point.

The ORDER BY column (`window_order_clause`) is the column whose value is used to compute the window span.

Only one window_order_clause column is allowed, and the data type must be NUMERIC, DATE/TIME, FLOAT or INTEGER, unless the window specifies one of following frames:

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
- RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

INTERVAL Year to Month can be used in an analytic RANGE window when the ORDER BY column type is TIMESTAMP/TIMESTAMP WITH TIMEZONE, or DATE. TIME/TIME WITH TIMEZONE are not supported.

INTERVAL Day to Second can be used when the ORDER BY column type is TIMESTAMP/TIMESTAMP WITH TIMEZONE, DATE, and TIME/TIME WITH TIMEZONE.

Examples

The examples on this page use the emp table schema defined in Window Framing:

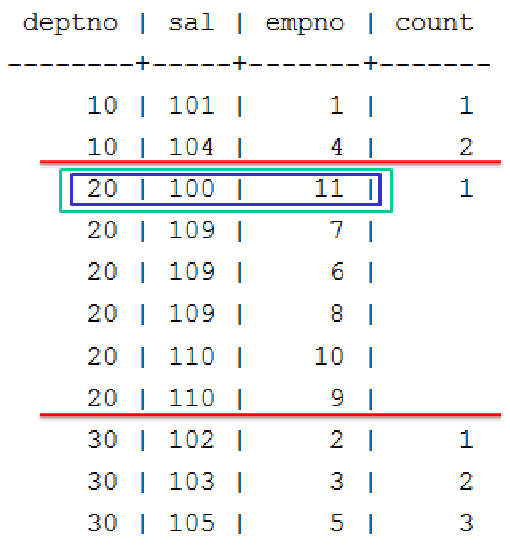
```
CREATE TABLE emp(deptno INT, sal INT, empno INT);
INSERT INTO emp VALUES(10,101,1);
INSERT INTO emp VALUES(10,104,4);
INSERT INTO emp VALUES(20,100,11);
INSERT INTO emp VALUES(20,109,7);
INSERT INTO emp VALUES(20,109,6);
INSERT INTO emp VALUES(20,109,8);
INSERT INTO emp VALUES(20,110,10);
INSERT INTO emp VALUES(20,110,9);
INSERT INTO emp VALUES(30,102,2);
INSERT INTO emp VALUES(30,103,3);
INSERT INTO emp VALUES(30,105,5);
COMMIT;
```

- The red line represents the partition
- The blue box represents the current row
- The green box represents the analytic window relative to the current row.

In the following query, RANGE specifies the window as a logical offset (value-based). The ORDER BY column is the column on which the range is applied.

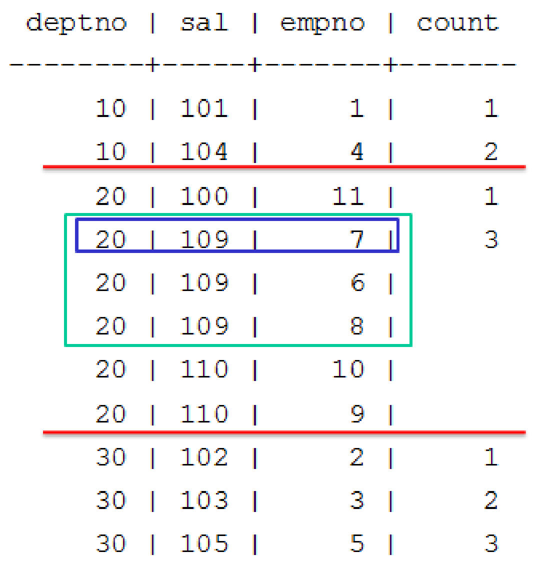
```
SELECT deptno, sal, empno, COUNT(*) OVER (PARTITION BY deptno ORDER BY sal
      RANGE BETWEEN 2 PRECEDING AND CURRENT ROW)
AS COUNT FROM emp;
```

The partition includes department 20, and the current row and window are the same because there are no rows that precede the current row within that partition:



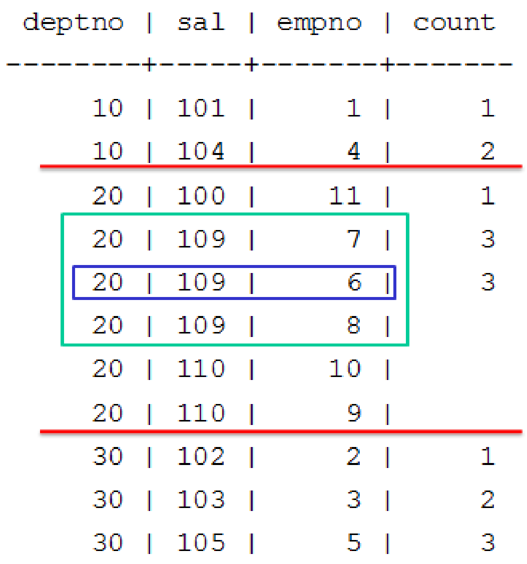
```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |
     20 | 109 |     6 |
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

In the next example, the ORDER BY column value is 109, so 109 - 2 = 107. The window would include all rows whose ORDER BY column values are between 107 and 109 inclusively.



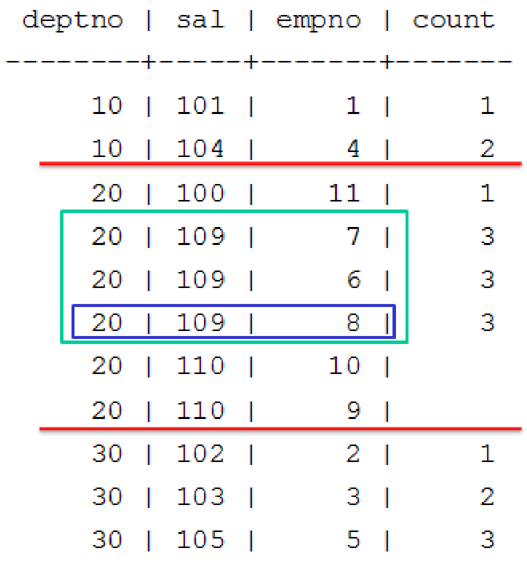
```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     3
     20 | 109 |     6 |
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

Here, the current row advances, and 107-109 are still inclusive.



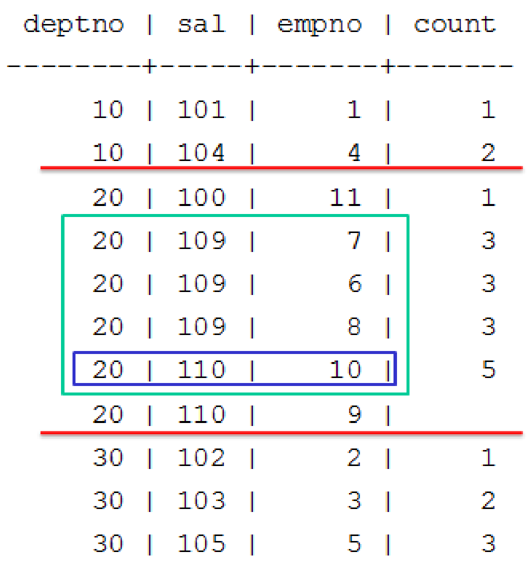
```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     3
     20 | 109 |     6 |     3
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

Though the current row advances again, the window is the same.



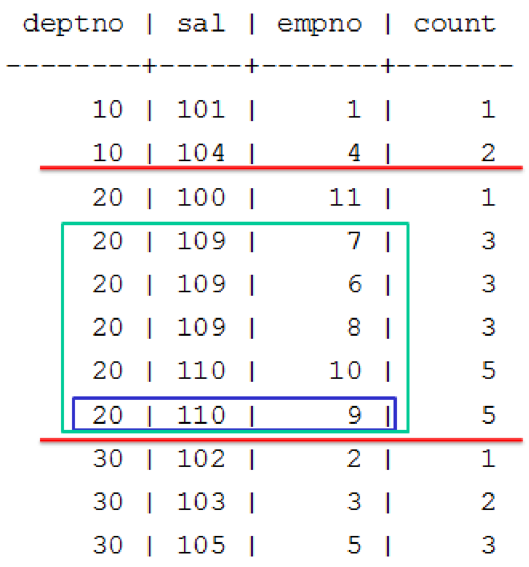
```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     3
     20 | 109 |     6 |     3
     20 | 109 |     8 |     3
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

The current row advances so that the ORDER BY column value becomes 110 (before it was 109). Now the window would include all rows whose ORDER BY column values were between 108 and 110, inclusive, because 110 - 2 = 108.



```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     3
     20 | 109 |     6 |     3
     20 | 109 |     8 |     3
     20 | 110 |    10 |     5
     20 | 110 |     9 |
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

As the current row advances, the window still includes rows for 108-110, inclusive.



```
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     3
     20 | 109 |     6 |     3
     20 | 109 |     8 |     3
     20 | 110 |    10 |     5
     20 | 110 |     9 |     5
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

Remember that a window frame can also be time based, such as the following query:

```
SELECT ts, bid, avg(bid) OVER
   (ORDER BY ts RANGE BETWEEN '40 SECONDS' PRECEDING AND CURRENT ROW)
   FROM ticks WHERE stock = 'VERT'
   GROUP BY bid, ts ORDER BY ts;
```

Reporting Aggregates

Some of the analytic functions that take the window_frame_clause are the reporting aggregates. These functions let you compare a partition's aggregate values with detail rows, taking the place of correlated subqueries or joins.

- AVG()
- COUNT()
- MAX() and MIN()
- SUM()
- STDDEV(), STDDEV_POP(), and STDDEV_SAMP()
- VARIANCE(), VAR_POP(), and VAR_SAMP()

If you use a window aggregate with an empty OVER() clause, the analytic function is used as a reporting function, where the entire input is treated as a single partition.

About Standard Deviation and Variance Functions

With standard deviation functions, a low standard deviation indicates that the data points tend to be very close to the mean, whereas high standard deviation indicates that the data points are spread out over a large range of values.

Standard deviation is often graphed and a distributed standard deviation creates the classic bell curve.

Variance functions measure how far a set of numbers is spread out.

Examples

Think of the window for reporting aggregates as a window defined as UNBOUNDED PRECEDING and UNBOUNDED FOLLOWING. The omission of a window_order_clause makes all rows in the partition also the window (reporting aggregates).

```
SELECT deptno, sal, empno, COUNT(sal)
  OVER (PARTITION BY deptno) AS COUNT FROM emp;
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     2
     10 | 104 |     4 |     2
------------------------------
     20 | 110 |    10 |     6
     20 | 110 |     9 |     6
     20 | 109 |     7 |     6
     20 | 109 |     6 |     6
     20 | 109 |     8 |     6
     20 | 100 |    11 |     6
------------------------------
     30 | 105 |     5 |     3
     30 | 103 |     3 |     3
     30 | 102 |     2 |     3
```

```
(11 rows)
```

If the OVER() clause in the above query contained a `window_order_clause` (for example, ORDER BY sal), it would become a moving window (window aggregate) query with a default window of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW:

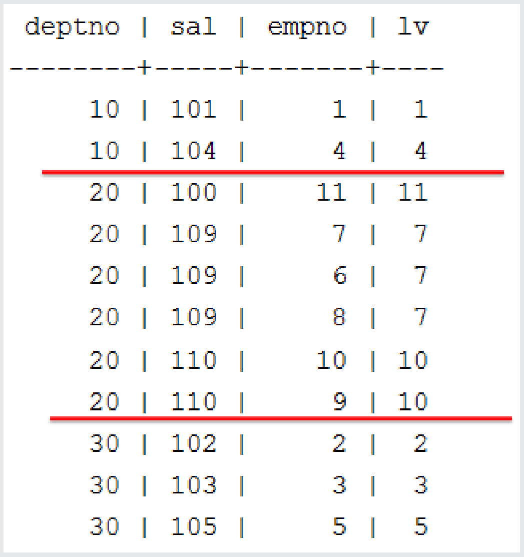
```
SELECT deptno, sal, empno, COUNT(sal)  OVER (PARTITION BY deptno ORDER BY sal) AS COUNT F
ROM emp;
 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
------------------------------
     20 | 100 |    11 |     1
     20 | 109 |     7 |     4
     20 | 109 |     6 |     4
     20 | 109 |     8 |     4
     20 | 110 |    10 |     6
     20 | 110 |     9 |     6
------------------------------
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
(11 rows)
```

What About LAST_VALUE()?

You might wonder why you couldn't just use the LAST_VALUE() analytic function.

For example, for each employee, get the highest salary in the department:

```
SELECT deptno, sal, empno,LAST_VALUE(empno) OVER (PARTITION BY deptno ORDER BY sal) AS lv
FROM emp;
```



```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |  7
     20 | 109 |     8 |  7
     20 | 110 |    10 | 10
     20 | 110 |     9 | 10
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```

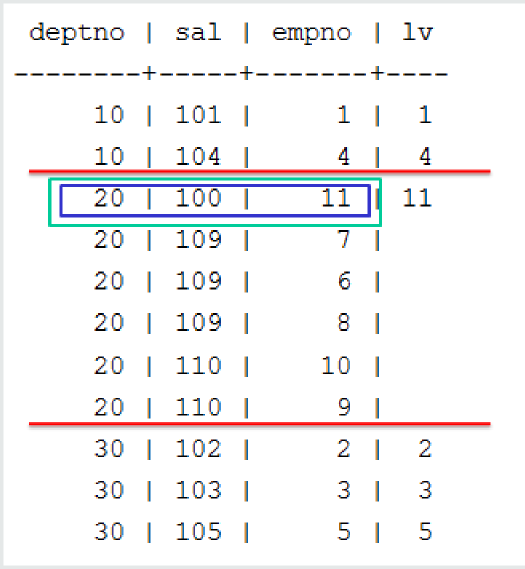
Due to default window semantics, LAST_VALUE does not always return the last value of a partition. If you omit the window_frame_clause from the analytic clause, LAST_VALUE operates on this default window. Results, therefore, can seem non-intuitive because the function does not return the bottom of the current partition. It returns the bottom of the window, which continues to change along with the current input row being processed.

Remember the default window:

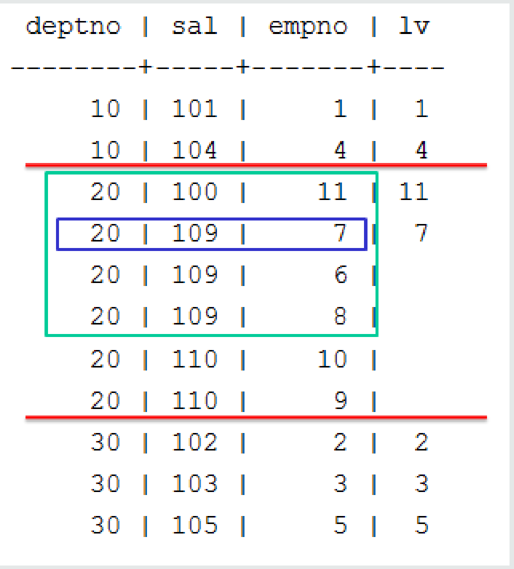
```
OVER (PARTITION BY deptno ORDER BY sal)
```

is the same as:

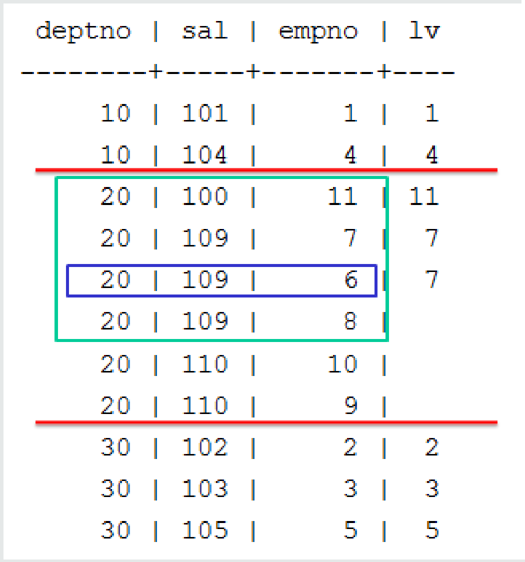
```
OVER(PARTITION BY deptno ORDER BY salROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```



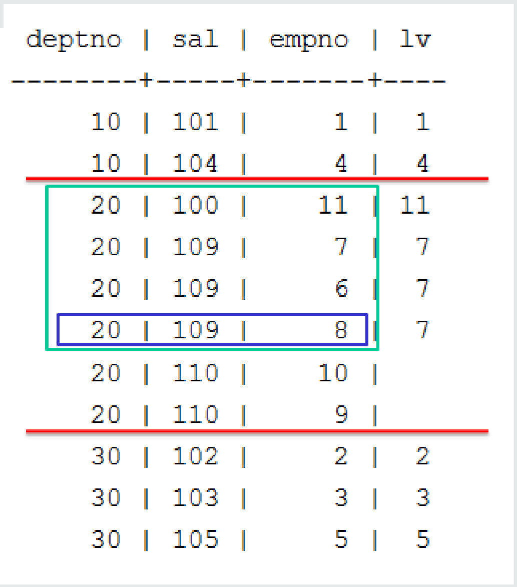
```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |
     20 | 109 |     6 |
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```



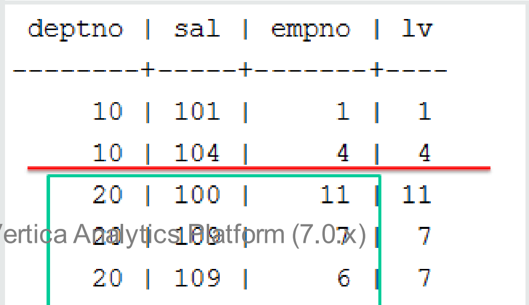
```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```



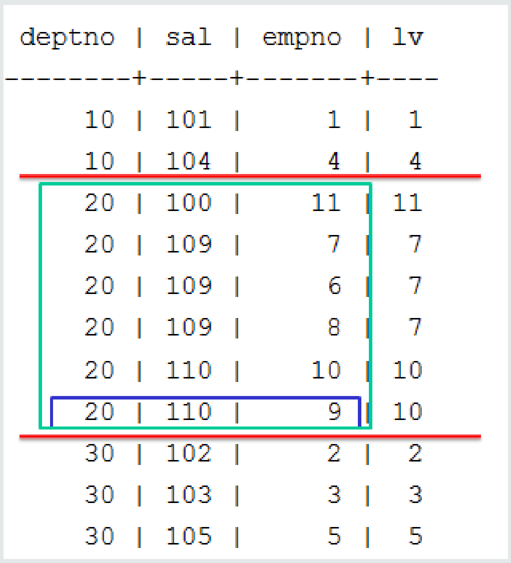
```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |  7
     20 | 109 |     8 |
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```



```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |  7
     20 | 109 |     8 |  7
     20 | 110 |    10 |
     20 | 110 |     9 |
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```



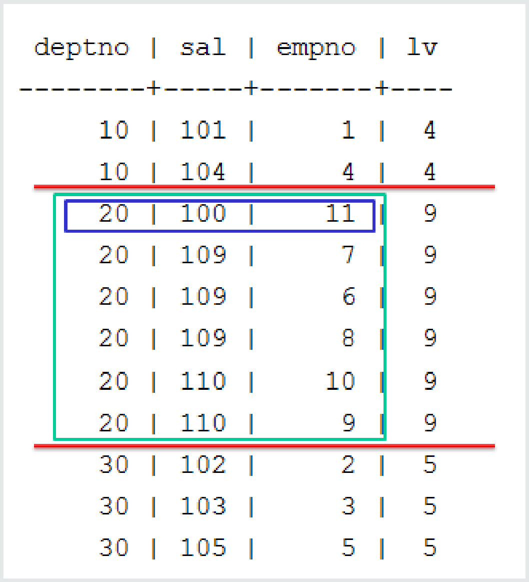
```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |  7
```



```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  1
     10 | 104 |     4 |  4
     20 | 100 |    11 | 11
     20 | 109 |     7 |  7
     20 | 109 |     6 |  7
     20 | 109 |     8 |  7
     20 | 110 |    10 | 10
     20 | 110 |     9 | 10
     30 | 102 |     2 |  2
     30 | 103 |     3 |  3
     30 | 105 |     5 |  5
```

If you want to return the last value of a partition, use UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

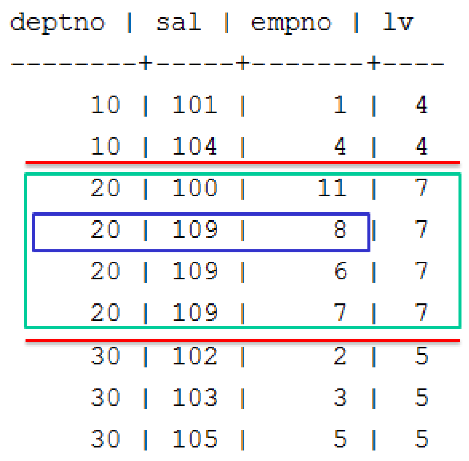
```
SELECT deptno, sal, empno, LAST_VALUE(empno)
OVER (PARTITION BY deptno ORDER BY sal
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
FROM emp;
```



```
 deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  4
     10 | 104 |     4 |  4
     20 | 100 |    11 |  9
     20 | 109 |     7 |  9
     20 | 109 |     6 |  9
     20 | 109 |     8 |  9
     20 | 110 |    10 |  9
     20 | 110 |     9 |  9
     30 | 102 |     2 |  5
     30 | 103 |     3 |  5
     30 | 105 |     5 |  5
```

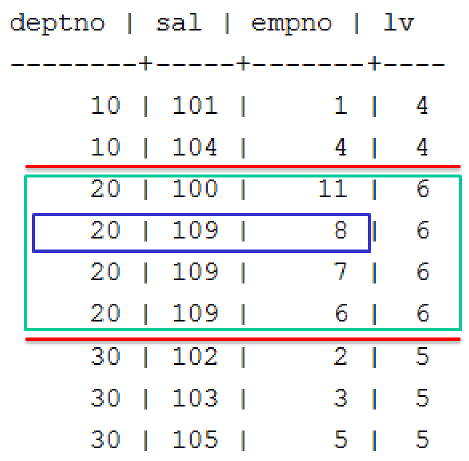
Vertica recommends that you use LAST_VALUE with the window_order_clause to produce **deterministic** results.

In the following example, empno 6, 7, and 8 have the same salary, so they are in adjacent rows. empno 8 appears first in this case but the order is not guaranteed.



```
deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  4
     10 | 104 |     4 |  4
     20 | 100 |    11 |  7
     20 | 109 |     8 |  7
     20 | 109 |     6 |  7
     20 | 109 |     7 |  7
     30 | 102 |     2 |  5
     30 | 103 |     3 |  5
     30 | 105 |     5 |  5
```

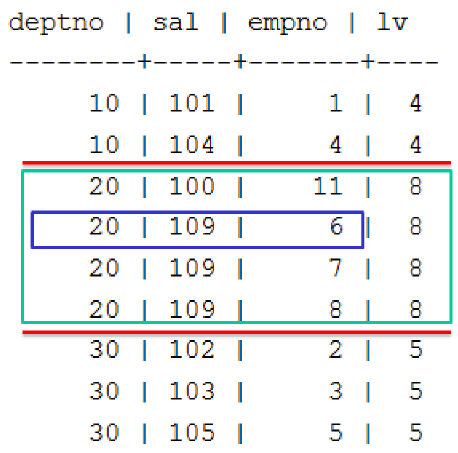
Notice in the output above, the last value is 7, which is the last row from the partition deptno = 20. If the rows have a different order, then the function returns a different value:



```
deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  4
     10 | 104 |     4 |  4
     20 | 100 |    11 |  6
     20 | 109 |     8 |  6
     20 | 109 |     7 |  6
     20 | 109 |     6 |  6
     30 | 102 |     2 |  5
     30 | 103 |     3 |  5
     30 | 105 |     5 |  5
```

Now the last value is 6, which is the last row from the partition deptno = 20. The solution is to add a unique key to the sort order. Even if the order of the query changes, the result will always be the same, and so **deterministic**.

```
SELECT deptno, sal, empno, LAST_VALUE(empno)
OVER (PARTITION BY deptno ORDER BY sal, empno
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as lv
FROM emp;
```



```
deptno | sal | empno | lv
--------+-----+-------+----
     10 | 101 |     1 |  4
     10 | 104 |     4 |  4
     20 | 100 |    11 |  8
     20 | 109 |     6 |  8
     20 | 109 |     7 |  8
     20 | 109 |     8 |  8
     30 | 102 |     2 |  5
     30 | 103 |     3 |  5
     30 | 105 |     5 |  5
```

Notice how the rows are now ordered by `empno`, the last value stays at 8, and it does not matter the order of the query.

Naming Windows

You can use the WINDOW clause to name one or more windows and avoid typing long OVER() clause syntax. The WINDOW clause takes the following form:

```
WINDOW window_name AS ( window_definition_clause );
```

```
[ window_partition_clause ] [ window_order_clause ]
```

The window_definition_clause is described in detail in the SQL Reference Manual.

Example

In the following example, RANK() and DENSE_RANK() use the partitioning and ordering specifications in the window definition for a window named `w`:

```
=> SELECT RANK() OVER w , DENSE_RANK() OVER w
    FROM employee_dimension
   WINDOW w AS (PARTITION BY employee_region ORDER by annual_salary);
```

Though analytic functions can reference a named window to inherit the window_partition_clause, you can define your own window_order_clause; for example:

```
=> SELECT RANK() OVER(w ORDER BY annual_salary ASC) ,        DENSE_RANK() OVER(w ORDER BY
annual_salary DESC)
   FROM employee_dimension
   WINDOW w AS (PARTITION BY employee_region);
```

Notes:

- Each window defined in the window_definition_clause must have a unique name.
- The `window_partition_clause` is defined in the named window specification, not in the `OVER()` clause.
- The `OVER()` clause can specify its own `window_order_clause` only if the `window_definition_clause` did not already define it. For example, if the second example above is rewritten as follows, the system returns an error:

```
=> SELECT RANK() OVER(w ORDER BY annual_salary ASC) ,
   DENSE_RANK() OVER(w ORDER BY annual_salary DESC)
   FROM employee_dimension
   WINDOW w AS (PARTITION BY employee_region ORDER BY annual_salary);
   ERROR:  cannot override ORDER BY clause of window "w"
```

- A window definition cannot contain a `window_frame_clause`.
- Each window defined in the `window_definition_clause` must have a unique name.

  You can reference window names within their scope only. For example, because named window `w1` below is defined before `w2`, `w2` is within the scope of `w1`:

```
=> SELECT RANK() OVER(w1 ORDER BY sal DESC),
   RANK() OVER w2
   FROM EMP
   WINDOW w1 AS (PARTITION BY deptno), w2 AS (w1 ORDER BY sal);
```

Analytic Query Examples

Calculating a Median Value

A median is described as the numerical value separating the higher half of a sample from the lower half. The median of a finite list of numbers can be found by arranging all the observations from lowest value to highest value and picking the middle one.

If there is an even number of observations, then there is no single middle value. The median is then defined to be the mean (average) of the two middle values.

The examples that follow use the allsales table schema, defined in How Analytic Functions Work.

Allsales Table Schema

```
CREATE TABLE allsales(state VARCHAR(20), name VARCHAR(20), sales INT);
INSERT INTO allsales VALUES('MA', 'A', 60);
INSERT INTO allsales VALUES('NY', 'B', 20);
INSERT INTO allsales VALUES('NY', 'C', 15);
INSERT INTO allsales VALUES('MA', 'D', 20);
INSERT INTO allsales VALUES('MA', 'E', 50);
INSERT INTO allsales VALUES('NY', 'F', 40);
INSERT INTO allsales VALUES('MA', 'G', 10);
COMMIT;
```

Table allsales:

```
=> SELECT * FROM allsales;
 state | name | sales
-------+------+-------
 MA    | A    |    60
 NY    | B    |    20
 NY    | C    |    15
 MA    | D    |    20
 MA    | E    |    50
 NY    | F    |    40
 MA    | G    |    10
(7 rows)
```

The following query calculates the median of sales from the allsales table. Note that when you use OVER() with no parameters, the query returns the same aggregated value for each row of the result set:

```
=> SELECT name, sales, MEDIAN(sales) OVER() AS median FROM allsales;
 name | sales | median
------+-------+--------
 G    |    10 |     20
 C    |    15 |     20
 D    |    20 |     20
 B    |    20 |     20
 F    |    40 |     20
 E    |    50 |     20
 A    |    60 |     20
(7 rows)
```

Without analytics, you'd have to write an overly complex query to get the median sales, but performance will suffer, and the query returns only one row:

```
=> SELECT sales MEDIAN FROM
   (
      SELECT a1.name, a1.sales, COUNT(a1.sales) Rank
      FROM allsales a1, allsales a2
      WHERE a1.sales < a2.sales OR
         (a1.sales=a2.sales AND a1.name <= a2.name)
      GROUP BY a1.name, a1.sales
      ORDER BY a1.sales desc
   ) a3
   WHERE Rank =
   (SELECT (COUNT(*)+1) / 2 FROM allsales);
 MEDIAN
--------
     20
(1 row)
```

Getting Price Differential for Two Stocks

The following subquery selects out two stocks of interest. The outer query uses the LAST_VALUE () and OVER() components of analytics, with IGNORE NULLS.

Schema

```
DROP TABLE Ticks CASCADE;


CREATE TABLE Ticks (ts TIMESTAMP, Stock varchar(10), Bid float);
INSERT INTO Ticks VALUES('2011-07-12 10:23:54', 'abc', 10.12);
INSERT INTO Ticks VALUES('2011-07-12 10:23:58', 'abc', 10.34);
INSERT INTO Ticks VALUES('2011-07-12 10:23:59', 'abc', 10.75);
INSERT INTO Ticks VALUES('2011-07-12 10:25:15', 'abc', 11.98);
INSERT INTO Ticks VALUES('2011-07-12 10:25:16', 'abc');
INSERT INTO Ticks VALUES('2011-07-12 10:25:22', 'xyz', 45.16);
INSERT INTO Ticks VALUES('2011-07-12 10:25:27', 'xyz', 49.33);
INSERT INTO Ticks VALUES('2011-07-12 10:31:12', 'xyz', 65.25);
INSERT INTO Ticks VALUES('2011-07-12 10:31:15', 'xyz');

COMMIT;
```

ticks Table

```
SELECT * FROM ticks;
         ts          | stock |  bid
```

```
---------------------+-------+-------
 2011-07-12 10:23:59 | abc   | 10.75
 2011-07-12 10:25:22 | xyz   | 45.16
 2011-07-12 10:23:58 | abc   | 10.34
 2011-07-12 10:25:27 | xyz   | 49.33
 2011-07-12 10:23:54 | abc   | 10.12
 2011-07-12 10:31:15 | xyz   |
 2011-07-12 10:25:15 | abc   | 11.98
 2011-07-12 10:25:16 | abc   |
 2011-07-12 10:31:12 | xyz   | 65.25
(9 rows)
```

Query

```
 SELECT ts, stock, bid, last_value(price1 IGNORE NULLS)
   OVER(ORDER BY ts) - last_value(price2 IGNORE NULLS)
   OVER(ORDER BY ts)   as price_diff
 FROM
  (SELECT ts, stock, bid,
    CASE WHEN stock = 'abc' THEN bid ELSE NULL END AS price1,
    CASE WHEN stock = 'xyz' then bid ELSE NULL END AS price2
    FROM ticks
    WHERE stock IN ('abc','xyz')
  ) v1
 ORDER BY ts;
         ts          | stock |  bid  | price_diff
---------------------+-------+-------+------------
 2011-07-12 10:23:54 | abc   | 10.12 |
 2011-07-12 10:23:58 | abc   | 10.34 |
 2011-07-12 10:23:59 | abc   | 10.75 |
 2011-07-12 10:25:15 | abc   | 11.98 |
 2011-07-12 10:25:16 | abc   |       |
 2011-07-12 10:25:22 | xyz   | 45.16 |     -33.18
 2011-07-12 10:25:27 | xyz   | 49.33 |     -37.35
 2011-07-12 10:31:12 | xyz   | 65.25 |     -53.27
 2011-07-12 10:31:15 | xyz   |       |     -53.27
(9 rows)
```

Calculating the Moving Average

Calculate a 40-second moving average of bids for one stock.

Note: This examples uses the ticks table schema defined in Getting Price Differential for Two Stocks.

Query

```
SELECT ts, bid, AVG(bid)
```

```
   OVER(ORDER BY ts
       RANGE BETWEEN INTERVAL '40 seconds'
       PRECEDING AND CURRENT ROW)
FROM ticks
WHERE stock = 'abc'
GROUP BY bid, ts
ORDER BY ts;
         ts          |  bid  |     ?column?
---------------------+-------+------------------
 2011-07-12 10:23:54 | 10.12 |            10.12
 2011-07-12 10:23:58 | 10.34 |            10.23
 2011-07-12 10:23:59 | 10.75 | 10.4033333333333
 2011-07-12 10:25:15 | 11.98 |            11.98
 2011-07-12 10:25:16 |       |            11.98
(5 rows)

  DROP TABLE Ticks CASCADE;


  CREATE TABLE Ticks (ts TIMESTAMP, Stock varchar(10), Bid float);
  INSERT INTO Ticks VALUES('2011-07-12 10:23:54', 'abc', 10.12);
  INSERT INTO Ticks VALUES('2011-07-12 10:23:58', 'abc', 10.34);
  INSERT INTO Ticks VALUES('2011-07-12 10:23:59', 'abc', 10.75);
  INSERT INTO Ticks VALUES('2011-07-12 10:25:15', 'abc', 11.98);
  INSERT INTO Ticks VALUES('2011-07-12 10:25:16', 'abc');
  INSERT INTO Ticks VALUES('2011-07-12 10:25:22', 'xyz', 45.16);
  INSERT INTO Ticks VALUES('2011-07-12 10:25:27', 'xyz', 49.33);
  INSERT INTO Ticks VALUES('2011-07-12 10:31:12', 'xyz', 65.25);
  INSERT INTO Ticks VALUES('2011-07-12 10:31:15', 'xyz');

  COMMIT;
```

Avoiding GROUPBY HASH with Projection Design

If your query contains a `GROUP BY` clause, HP Vertica computes the result with either the `GROUPBY PIPELINED` or `GROUPBY HASH` algorithm.

Both algorithms compute the same results and have similar performance when the query produces a small number of distinct groups (typically a thousand per node in the cluster). For queries that contain a large number of groups, `GROUPBY PIPELINED` uses less memory and can be faster but is only used when the input data is pre-sorted on the `GROUP BY` columns.

To improve the performance of a query that has a large number of distinct groups that is currently using the `GROUP BY HASH` algorithm, you can enable the use of the `GROUPBY PIPELINED` algorithm, as this section describes.

To determine which algorithm your query is using, run the EXPLAIN statement on the query.

The three conditions described in this section refer to the following schema.

```
CREATE TABLE sortopt (
    a INT NOT NULL,
    b INT NOT NULL,
     c INT,
     d INT
);
CREATE PROJECTION sortopt_p (
   a_proj,
   b_proj,
   c_proj,
   d_proj )
AS SELECT * FROM sortopt
ORDER BY a,b,c
UNSEGMENTED ALL NODES;
INSERT INTO sortopt VALUES(5,2,13,84);
INSERT INTO sortopt VALUES(14,22,8,115);
INSERT INTO sortopt VALUES(79,9,401,33);
```

Condition #1

All columns in the query's `GROUP BY` clause must be included in the projection's sort columns. If even one column in the GROUP BY clause is excluded from the projection's `ORDER BY` clause, HP Vertica uses `GROUPBY HASH` instead of `GROUPBY PIPELINED`:

Given a projection sort order `ORDER BY a, b, c`:

`GROUP BY a`
`GROUP BY a,b`
`GROUP BY b,a`
`GROUP BY a,b,c`
`GROUP BY c,a,b`
	The query optimizer uses `GROUPBY PIPELINED` because columns `a`, `b`, and `c` are included in the projection sort columns.

`GROUP BY a,b,c,d`	The query optimizer uses `GROUPBY HASH` because column `d` is not part of the projection sort columns.

Condition #2

If the number of columns in the query's `GROUP BY` clause is less than the number of columns in the projection's `ORDER BY` clause, columns in the query's `GROUP BY` clause must occur *first* in the projection's `ORDER BY` clause.

Given a projection sort order `ORDER BY a, b ,c`:

`GROUP BY a`
`GROUP BY a,b`
`GROUP BY b,a`
`GROUP BY a,b,c`
`GROUP BY c,a,b`
	The query optimizer uses `GROUPBY PIPELINED` because columns `a`, `b`, `c` are included in the projection sort columns.

`GROUP BY a,c`	The query optimizer uses `GROUPBY HASH` because columns `a` and `c` do not occur first in the projection sort columns.

Condition #3

If the columns in a query's `GROUP BY` clause do not appear first in the projection's `ORDER BY` clause, then any early-appearing projection sort columns that are missing in the query's `GROUP BY` clause must be present as single-column constant equality predicates in the query's `WHERE` clause.

Given a projection sort order `ORDER BY a, b, c:`

`SELECT a FROM tab WHERE a = 10 GROUP BY b`	The query optimizer uses `GROUPBY PIPELINED` because all columns preceding `b` in the projection sort order appear as constant equality predicates.
`SELECT a FROM tab WHERE a = 10 GROUP BY a, b`	The query optimizer uses `GROUPBY PIPELINED` even if redundant grouping column `a` is present.
`SELECT a FROM tab WHERE a = 10 GROUP BY b, c`	The query optimizer uses `GROUPBY PIPELINED` because all columns preceding `b` and `c` in the projection sort order appear as constant equality predicates.
`SELECT a FROM tab WHERE a = 10 GROUP BY c, b`	The query optimizer uses `GROUPBY PIPELINED` because all columns preceding `b` and `c` in the projection sort order appear as constant equality predicates.
`SELECT a FROM tab WHERE a = 10 GROUP BY c`	The query optimizer uses `GROUPBY HASH` because all columns preceding `c` in the projection sort order do *not* appear as constant equality predicates.

Getting Latest Bid and Ask Results

The following query fills in missing (null) values to create a full book order showing latest bid and ask price and size, by vendor id. Original rows have values for (typically) one price and one size, so use last_value with "ignore nulls" to find the most recent non-null value for the other pair each time there is an entry for the ID. Sequenceno provides a unique total ordering.

Schema:

```
CREATE TABLE bookorders(
    vendorid VARCHAR(100),
    date TIMESTAMP,
    sequenceno INT,
    askprice FLOAT,
    asksize INT,
    bidprice FLOAT,
    bidsize INT);
```

```
INSERT INTO bookorders VALUES('3325XPK','2011-07-12 10:23:54', 1, 10.12, 55, 10.23, 59);
INSERT INTO bookorders VALUES('3345XPZ','2011-07-12 10:23:55', 2, 10.55, 58, 10.75, 57);
INSERT INTO bookorders VALUES('445XPKF','2011-07-12 10:23:56', 3, 10.22, 43, 54);
INSERT INTO bookorders VALUES('445XPKF','2011-07-12 10:23:57', 3, 10.22, 59, 10.25, 61);
INSERT INTO bookorders VALUES('3425XPY','2011-07-12 10:23:58', 4, 11.87, 66, 11.90, 66);
INSERT INTO bookorders VALUES('3727XVK','2011-07-12 10:23:59', 5, 11.66, 51, 11.67, 62);
INSERT INTO bookorders VALUES('5325XYZ','2011-07-12 10:24:01', 6, 15.05, 44, 15.10, 59);
INSERT INTO bookorders VALUES('3675XVS','2011-07-12 10:24:05', 7, 15.43, 47, 58);
INSERT INTO bookorders VALUES('8972VUG','2011-07-12 10:25:15', 8, 14.95, 52, 15.11, 57);
COMMIT;
```

Query:

```
SELECT
    sequenceno Seq,
    date "Time",
    vendorid ID,
    LAST_VALUE (bidprice IGNORE NULLS)
     OVER (PARTITION BY vendorid ORDER BY sequenceno
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS "Bid Price",
    LAST_VALUE (bidsize IGNORE NULLS)
     OVER (PARTITION BY vendorid ORDER BY sequenceno
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS "Bid Size",
    LAST_VALUE (askprice IGNORE NULLS)
     OVER (PARTITION BY vendorid ORDER BY sequenceno
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS "Ask Price",
    LAST_VALUE (asksize IGNORE NULLS)
     OVER (PARTITION BY vendorid order by sequenceno
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
    AS  "Ask Size"
 FROM bookorders
ORDER BY sequenceno;

Seq |        Time         |   ID    | Bid Price | Bid Size | Ask Price | Ask Size
-----+---------------------+---------+-----------+----------+-----------+----------
   1 | 2011-07-12 10:23:54 | 3325XPK |     10.23 |       59 |     10.12 |       55
   1 | 2011-07-12 10:23:54 | 3325XPK |     10.23 |       59 |     10.12 |       55
   2 | 2011-07-12 10:23:55 | 3345XPZ |     10.75 |       57 |     10.55 |       58
   2 | 2011-07-12 10:23:55 | 3345XPZ |     10.75 |       57 |     10.55 |       58
   3 | 2011-07-12 10:23:56 | 445XPKF |        54 |       61 |     10.22 |       43
   3 | 2011-07-12 10:23:57 | 445XPKF |     10.25 |       61 |     10.22 |       59
   3 | 2011-07-12 10:23:56 | 445XPKF |        54 |       61 |     10.22 |       43
   3 | 2011-07-12 10:23:57 | 445XPKF |     10.25 |       61 |     10.22 |       59
   4 | 2011-07-12 10:23:58 | 3425XPY |      11.9 |       66 |     11.87 |       66
   4 | 2011-07-12 10:23:58 | 3425XPY |      11.9 |       66 |     11.87 |       66
   5 | 2011-07-12 10:23:59 | 3727XVK |     11.67 |      662 |     11.66 |       51
   5 | 2011-07-12 10:23:59 | 3727XVK |     11.67 |       62 |     11.66 |       51
   6 | 2011-07-12 10:24:01 | 5325XYZ |      15.1 |       59 |     15.05 |       44
   6 | 2011-07-12 10:24:01 | 5325XYZ |      15.1 |       59 |     15.05 |       44
   7 | 2011-07-12 10:24:05 | 3675XVS |        58 |          |     15.43 |       47
   7 | 2011-07-12 10:24:05 | 3675XVS |        58 |          |     15.43 |       47
   8 | 2011-07-12 10:25:15 | 8972VUG |     15.11 |       57 |     14.95 |       52
   8 | 2011-07-12 10:25:15 | 8972VUG |     15.11 |       57 |     14.95 |       52
(18 rows)
```

Event-Based Windows

Event-based windows let you break time series data into windows that border on significant events within the data. This is especially relevant in financial data where analysis often focuses on specific events as triggers to other activity.

There are two event-based window functions in HP Vertica. These functions are an HP Vertica extension and are not part of the SQL-99 standard:

- CONDITIONAL_CHANGE_EVENT() assigns an event window number to each row, starting from 0, and increments by 1 when the result of evaluating the argument expression on the current row differs from that on the previous row. This function is similar to the analytic function ROW_NUMBER, which assigns a unique number, sequentially, starting from 1, to each row within a partition.
- CONDITIONAL_TRUE_EVENT() assigns an event window number to each row, starting from 0, and increments the number by 1 when the result of the boolean argument expression evaluates true.

These functions are described in greater detail below.

Note: The CONDITIONAL_CHANGE_EVENT and CONDITIONAL_TRUE_EVENT functions do not allow Window Framing.

Example Schema

The examples in this topic use the following schema:

```
CREATE TABLE TickStore3 (
    ts TIMESTAMP,
    symbol VARCHAR(8),
    bid FLOAT
);
CREATE PROJECTION TickStore3_p (ts, symbol, bid) AS
SELECT * FROM TickStore3
ORDER BY ts, symbol, bid UNSEGMENTED ALL NODES;
INSERT INTO TickStore3 VALUES ('2009-01-01 03:00:00', 'XYZ', 10.0);
INSERT INTO TickStore3 VALUES ('2009-01-01 03:00:03', 'XYZ', 11.0);
INSERT INTO TickStore3 VALUES ('2009-01-01 03:00:06', 'XYZ', 10.5);
INSERT INTO TickStore3 VALUES ('2009-01-01 03:00:09', 'XYZ', 11.0);
COMMIT;
```

Using the CONDITIONAL_CHANGE_EVENT Function

The analytical function CONDITIONAL_CHANGE_EVENT returns a sequence of integers indicating event window numbers, starting from 0. The function increments the event window number when the result of evaluating the function expression on the current row differs from the previous value.

In the following example, the first query returns all records from the TickStore3 table. The second query uses the CONDITIONAL_CHANGE_EVENT function on the bid column. Since each bid row value is different from the previous value, the function increments the window ID from 0 to 3:

```
SELECT ts, symbol, bidFROM Tickstore3
ORDER BY ts;
```

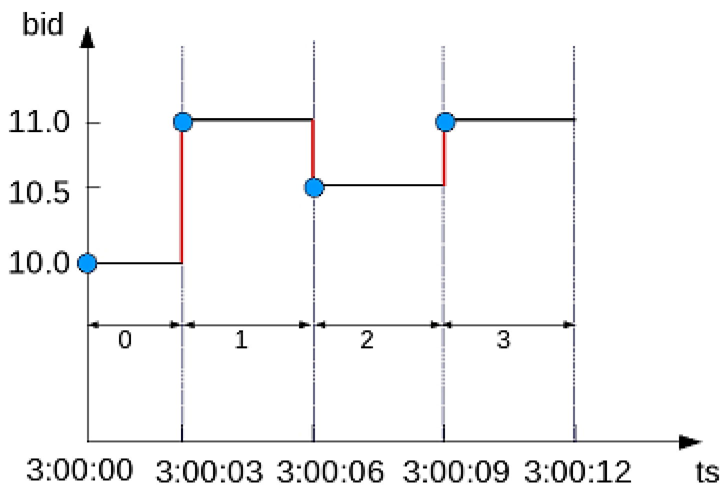
```
         ts          | symbol | bid
---------------------+--------+------
 2009-01-01 03:00:00 | XYZ    |   10
 2009-01-01 03:00:03 | XYZ    |   11
 2009-01-01 03:00:06 | XYZ    | 10.5
 2009-01-01 03:00:09 | XYZ    |   11
(4 rows)
```

==>

```
SELECT CONDITIONAL_CHANGE_EVENT(bid)
OVER(ORDER BY ts)
FROM Tickstore3;
```

```
         ts          | symbol | bid  | cce
---------------------+--------+------+-----
 2009-01-01 03:00:00 | XYZ    |   10 | 0
 2009-01-01 03:00:03 | XYZ    |   11 | 1
 2009-01-01 03:00:06 | XYZ    | 10.5 | 2
 2009-01-01 03:00:09 | XYZ    |   11 | 3
(4 rows)
```

The following figure is a graphical illustration of the change in the bid price. Each value is different from its previous one, so the window ID increments for each time slice:



So the window ID starts at 0 and increments at every change in from the previous value.

In this example, the bid price changes from $10 to $11 in the second row, but then stays the same. The CONDITIONAL_CHANGE_EVENT function increments the event window ID in row 2, but not subsequently:

```
SELECT ts, symbol, bidFROM Ticksto
re3
ORDER BY ts;
```

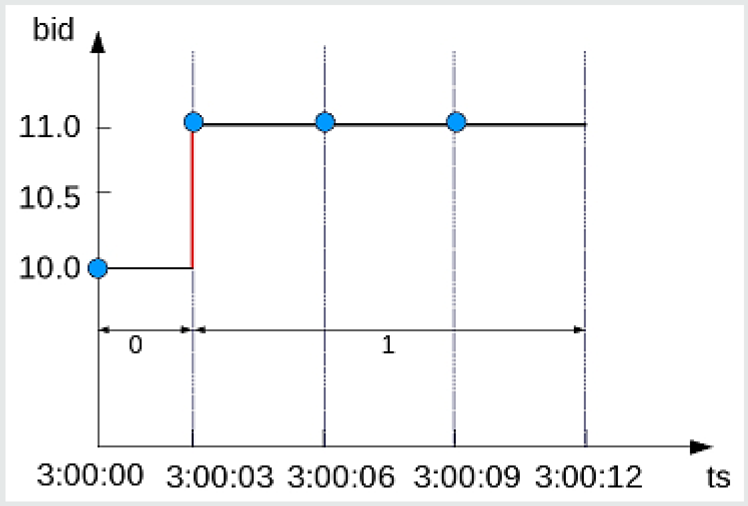
```
         ts          | symbol | bi
d
---------------------+--------+---
---
 2009-01-01 03:00:00 | XYZ    |
10
 2009-01-01 03:00:03 | XYZ    |
11
 2009-01-01 03:00:06 | XYZ    |
11
 2009-01-01 03:00:09 | XYZ    |
11
```

```
=
=>
```

```
SELECT CONDITIONAL_CHANGE_EVENT(bid)  OVER(ORDER B
Y ts)
FROM Tickstore3;
```

```
         ts          | symbol | bid  | cce
---------------------+--------+------+-----
 2009-01-01 03:00:00 | XYZ    |   10 | 0
 2009-01-01 03:00:03 | XYZ    |   11 | 1
 2009-01-01 03:00:06 | XYZ    |   11 | 1
 2009-01-01 03:00:09 | XYZ    |   11 | 1
```

The following figure is a graphical illustration of the change in the bid price at 3:00:03 only. The price stays the same at 3:00:06 and 3:00:09, so the window ID remains at 1 for each time slice after the change:



Using the CONDITIONAL_TRUE_EVENT Function

Like CONDITIONAL_CHANGE_EVENT, the analytic function CONDITIONAL_TRUE_EVENT also returns a sequence of integers indicating event window numbers, starting from 0. The difference between the two functions is that the CONDITIONAL_TRUE_EVENT function increments the window ID each time its expression evaluates to true, while CONDITIONAL_ CHANGE_EVENT increments on a comparison expression with the previous value.

In the following example, the first query returns all records from the TickStore3 table. The second query uses the CONDITIONAL_TRUE_EVENT function to test whether the current bid is greater than a given value (10.6). Each time the expression tests true, the function increments the window ID. The first time the function increments the window ID is on row 2, when the value is 11. The expression tests false for the next row (value is not greater than 10.6), so the function does not increment the event window ID. In the final row, the expression is true for the given condition, and the function increments the window:

```
SELECT ts, symbol, bidFRO
M Tickstore3
ORDER BY ts;
```

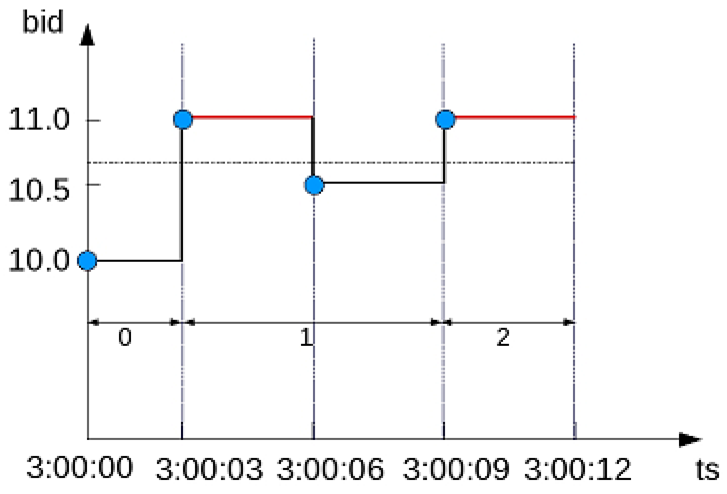
```
SELECT CONDITIONAL_TRUE_EVENT(bid > 10.6)
OVER(ORDER BY ts)
FROM Tickstore3;
```

```
         ts          | sy
mbol | bid
---------------------+---
-----+------
 2009-01-01 03:00:00 | XY
Z    |   10
 2009-01-01 03:00:03 | XY
Z    |   11
 2009-01-01 03:00:06 | XY
Z    | 10.5
 2009-01-01 03:00:09 | XY
Z    |   11
```

```
=
=
>
```

```
         ts          | symbol | bid  | cte------------------
---+--------+------+-----
 2009-01-01 03:00:00 | XYZ    |   10 | 0
 2009-01-01 03:00:03 | XYZ    |   11 | 1
 2009-01-01 03:00:06 | XYZ    | 10.5 | 1
 2009-01-01 03:00:09 | XYZ    |   11 | 2
```

The following figure is a graphical illustration that shows the bid values and window ID changes. Because the bid value is greater than $10.6 on only the second and fourth time slices (3:00:03 and 3:00:09), the window ID returns <0,**1**,1,**2**>:



In the following example, the first query returns all records from the TickStore3 table, ordered by the tickstore values (ts). The second query uses the CONDITIONAL_TRUE_EVENT function to increment the window ID each time the bid value is greater than 10.6. The first time the function increments the event window ID is on row 2, where the value is 11. The window ID then increments each time after that, because the expression (bid > 10.6) tests true for each time slice:

```
SELECT ts, symbol, bidFRO
M Tickstore3
ORDER BY ts;
```

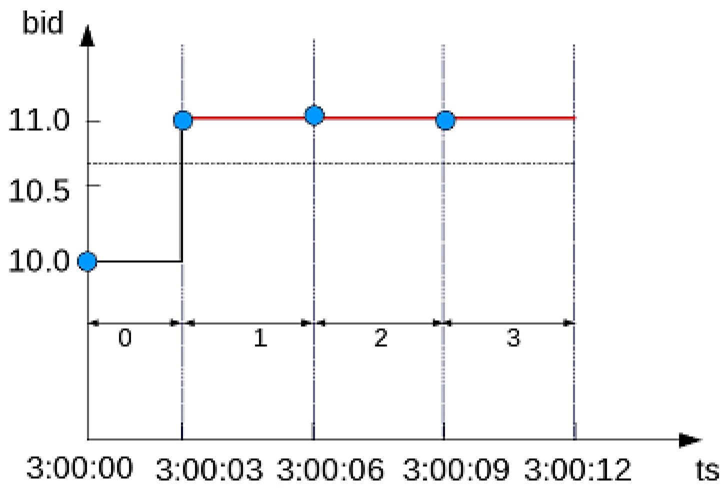
```
SELECT CONDITIONAL_TRUE_EVENT(bid > 10.6)
OVER(ORDER BY ts)
FROM Tickstore3;
```

```
         ts          | sy
mbol | bid
---------------------+---
-----+------
 2009-01-01 03:00:00 | XY
Z    |   10
 2009-01-01 03:00:03 | XY
Z    |   11
 2009-01-01 03:00:06 | XY
Z    |   11
 2009-01-01 03:00:09 | XY
Z    |   11
```

```
=
=
>
```

```
         ts          | symbol | bid  | cte------------------
---+--------+------+-----
 2009-01-01 03:00:00 | XYZ    |   10 | 0
 2009-01-01 03:00:03 | XYZ    |   11 | 1
 2009-01-01 03:00:06 | XYZ    |   11 | 2
 2009-01-01 03:00:09 | XYZ    |   11 | 3
```

The following figure is a graphical illustration that shows the bid values and window ID changes. The bid value is greater than 10.6 on the second time slice (3:00:03) and remains for the remaining two time slices. The function increments the event window ID each time because the expression tests true:



Advanced Use of Event-Based Windows

In event-based window functions, the condition expression accesses values from the current row only. To access a previous value, you can use a more powerful event-based window that allows the window event condition to include previous data points. For example, the LAG(x, n) analytic function retrieves the value of column X in the *n*th to last input record. In this case, LAG() shares the OVER() clause specifications of the CONDITIONAL_CHANGE_EVENT or CONDITIONAL_ TRUE_EVENT function expression.

In the following example, the first query returns all records from the TickStore3 table. The second query uses the CONDITIONAL_TRUE_EVENT function with the LAG() function in its boolean expression. In this case, the CONDITIONAL_TRUE_EVENT function increments the event window ID each time the bid value on the current row is less than the previous value. The first time CONDITIONAL_TRUE_EVENT increments the window ID starts on the third time slice, when the expression tests true. The current value (10.5) is less than the previous value. The window ID is not incremented in the last row because the final value is greater than the previous row:

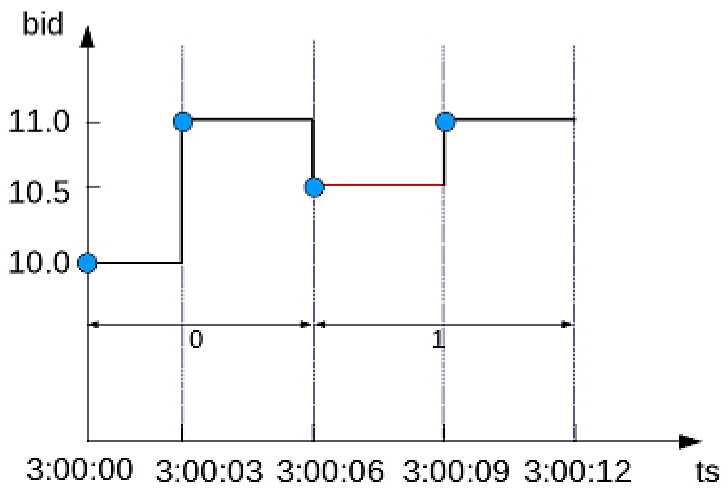
```
SELECT ts, symbol, bidFROM Tickstore3
ORDER BY ts;
```

```
         ts          | symbol | bid
---------------------+--------+------
 2009-01-01 03:00:00 | XYZ    |   10
 2009-01-01 03:00:03 | XYZ    |   11
 2009-01-01 03:00:06 | XYZ    | 10.5
 2009-01-01 03:00:09 | XYZ    |   11
```

```
SELECT CONDITIONAL_TRUE_EVENT(bid <  LAG(bid))
OVER(ORDER BY ts)
FROM Tickstore;
```

```
         ts          | symbol | bid  | cte
---------------------+--------+------+-----
 2009-01-01 03:00:00 | XYZ    |   10 | 0
 2009-01-01 03:00:03 | XYZ    |   11 | 0
 2009-01-01 03:00:06 | XYZ    | 10.5 | 1
 2009-01-01 03:00:09 | XYZ    |   11 | 1
```

The following figure illustrates the second query above. When the bid price is less than the previous value, the window ID gets incremented, which occurs only in the third time slice (3:00:06):



See Also

- Sessionization with Event-Based Windows
- Using Time Series Analytics
- CONDITIONAL_CHANGE_EVENT [Analytic]
- CONDITIONAL_TRUE_EVENT [Analytic]
- LAG [Analytic]

Sessionization with Event-Based Windows

Sessionization, a special case of event-based windows, is a feature often used to analyze click streams, such as identifying web browsing sessions from recorded web clicks.

In HP Vertica, given an input clickstream table, where each row records a Web page click made by a particular user (or IP address), the sessionization computation attempts to identify Web browsing sessions from the recorded clicks by grouping the clicks from each user based on the time-intervals between the clicks. If two clicks from the same user are made too far apart in time, as defined by a time-out threshold, the clicks are treated as though they are from two different browsing sessions.

Example Schema

The examples in this topic use the following WebClicks schema to represent a simple clickstream table:

```
CREATE TABLE WebClicks(userId INT, timestamp TIMESTAMP);
INSERT INTO WebClicks VALUES (1, '2009-12-08 3:00:00 pm');
INSERT INTO WebClicks VALUES (1, '2009-12-08 3:00:25 pm');
INSERT INTO WebClicks VALUES (1, '2009-12-08 3:00:45 pm');
INSERT INTO WebClicks VALUES (1, '2009-12-08 3:01:45 pm');
INSERT INTO WebClicks VALUES (2, '2009-12-08 3:02:45 pm');
INSERT INTO WebClicks VALUES (2, '2009-12-08 3:02:55 pm');
INSERT INTO WebClicks VALUES (2, '2009-12-08 3:03:55 pm');
COMMIT;
```

The input table `WebClicks` contains the following rows:

```
=> SELECT * FROM WebClicks;
 userId |      timestamp
--------+---------------------
      1 | 2009-12-08 15:00:00
      1 | 2009-12-08 15:00:25
      1 | 2009-12-08 15:00:45
      1 | 2009-12-08 15:01:45
      2 | 2009-12-08 15:02:45
      2 | 2009-12-08 15:02:55
      2 | 2009-12-08 15:03:55
(7 rows)
```

In the following query, sessionization performs computation on the SELECT list columns, showing the difference between the current and previous timestamp value using `LAG()`. It evaluates to true and increments the window ID when the difference is greater than 30 seconds.

```
=> SELECT userId, timestamp,
     CONDITIONAL_TRUE_EVENT(timestamp - LAG(timestamp) > '30 seconds')
     OVER(PARTITION BY userId ORDER BY timestamp) AS session FROM WebClicks;
 userId |      timestamp      | session
--------+---------------------+---------
      1 | 2009-12-08 15:00:00 |       0
      1 | 2009-12-08 15:00:25 |       0
      1 | 2009-12-08 15:00:45 |       0
      1 | 2009-12-08 15:01:45 |       1
      2 | 2009-12-08 15:02:45 |       0
      2 | 2009-12-08 15:02:55 |       0
      2 | 2009-12-08 15:03:55 |       1
(7 rows)
```

In the output, the session column contains the window ID from the CONDITIONAL_TRUE_EVENT function. The window ID evaluates to true on row 4 (timestamp 15:01:45), and the ID that follows row 4 is zero because it is the start of a new partition (for user ID 2), and that row does not evaluate to true until the last line in the output.

You might want to give users different time-out thresholds. For example, one user might have a slower network connection or be multi-tasking, while another user might have a faster connection and be focused on a single Web site, doing a single task.

To compute an adaptive time-out threshold based on the last 2 clicks, use CONDITIONAL_TRUE_EVENT with LAG to return the average time between the last 2 clicks with a grace period of 3 seconds:

```
SELECT userId, timestamp, CONDITIONAL_TRUE_EVENT(timestamp - LAG(timestamp) >
(LAG(timestamp, 1) - LAG(timestamp, 3)) / 2 + '3 seconds')
OVER(PARTITION BY userId ORDER BY timestamp) AS session
FROM WebClicks;
 userId |      timestamp      | session
--------+---------------------+---------
      2 | 2009-12-08 15:02:45 |       0
      2 | 2009-12-08 15:02:55 |       0
      2 | 2009-12-08 15:03:55 |       0
      1 | 2009-12-08 15:00:00 |       0
      1 | 2009-12-08 15:00:25 |       0
      1 | 2009-12-08 15:00:45 |       0
      1 | 2009-12-08 15:01:45 |       1
(7 rows)
```

Note: You cannot define a moving window in time series data. For example, if the query is evaluating the first row and there’s no data, it will be the current row. If you have a lag of 2, no results are returned until the third row.

See Also

- Event-Based Windows
- CONDITIONAL_TRUE_EVENT [Analytic]

Using Time Series Analytics

Time series analytics evaluate the values of a given set of variables over time and group those values into a window (based on a time interval) for analysis and aggregation.

Common scenarios are changes over time, such as stock market trades and performance, as well as charting trend lines over data.

Because both time and the state of data within a time series are continuous, it can be challenging to evaluate SQL queries over time. Input records usually occur at non-uniform intervals, which means they might have gaps. HP Vertica provides gap-filling functionality—which fills in missing data points, as—and an interpolation scheme, which is a method of constructing new data points within the range of a discrete set of known data points. HP Vertica interpolates the non-time series columns in the data (such as analytic function results computed over time slices) and adds the missing data points to the output. Gap filling and interpolation are described in detail in this section.

You can also use Event-Based Windows to break time series data into windows that border on significant events within the data. This is especially relevant in financial data where analysis might focus on specific events as triggers to other activity. Sessionization, a special case of event-based windows, is a feature often used to analyze click streams, such as identifying web browsing sessions from recorded web clicks.

HP Vertica provides additional support for time series analytics with the following SQL extensions, which you can read about in the SQL Reference Manual.

- The SELECT..TIMESERIES clause supports gap-filling and interpolation (GFI) computation.
- TS_FIRST_VALUE and TS_LAST_VALUE are time series aggregate functions that return the value at the start or end of a time slice, respectively, which is determined by the interpolation scheme.
- TIME_SLICE is a (SQL extension) date/time function that aggregates data by different fixed-time intervals and returns a rounded-up input TIMESTAMP value to a value that corresponds with the start or end of the time slice interval.

See Also

- Using SQL Analytics
- Event-Based Windows
- Sessionization with Event-Based Windows

Gap Filling and Interpolation (GFI)

The examples and graphics that explain the concepts in this topic use the following simple schema:

```
CREATE TABLE TickStore (ts TIMESTAMP, symbol VARCHAR(8), bid FLOAT);
INSERT INTO TickStore VALUES ('2009-01-01 03:00:00', 'XYZ', 10.0);
INSERT INTO TickStore VALUES ('2009-01-01 03:00:05', 'XYZ', 10.5);
COMMIT;
```

In HP Vertica, time series data is represented by a sequence of rows that conforms to a particular table schema, where one of the columns stores the time information.

Both time and the state of data within a time series are continuous. This means that evaluating SQL queries over time can be challenging because input records usually occur at non-uniform intervals and could contain gaps. Consider, for example, the following table, which contains two input rows five seconds apart, at 3:00:00 and 3:00:05.

```
=> SELECT * FROM TickStore;
         ts          | symbol | bid
---------------------+--------+------
 2009-01-01 03:00:00 | XYZ    |   10
 2009-01-01 03:00:05 | XYZ    | 10.5
(2 rows)
```

Given those two inputs, how would you determine a bid price that fell between the two points, such as at 3:00:03 PM?

The TIME_SLICE function, which normalizes timestamps into corresponding time slices, might seem like a logical candidate; however, TIME_SLICE does not solve the problem of missing inputs (time slices) in the data. Instead, HP Vertica provides gap-filling and interpolation (GFI) functionality, which fills in missing data points and adds new (missing) data points within a range of known data points to the output using time series aggregate functions and the SQL TIMESERIES clause.

But first, we'll illustrate the components that make up gap filling and interpolation in HP Vertica, starting with Constant Interpolation.

The images in the following topics use the following legend:

- The x-axis represents the timestamp (`ts`) column
- The y-axis represents the bid column.
- The vertical blue lines delimit the time slices.
- The red dots represent the input records in the table, $10.0 and $10.5.
- The blue stars represent the output values, including interpolated values.

Constant Interpolation

Given known input timestamps at 03:00:00 and 03:00:05 in the sample TickStore schema, how might you determine the bid price at 03:00:03?

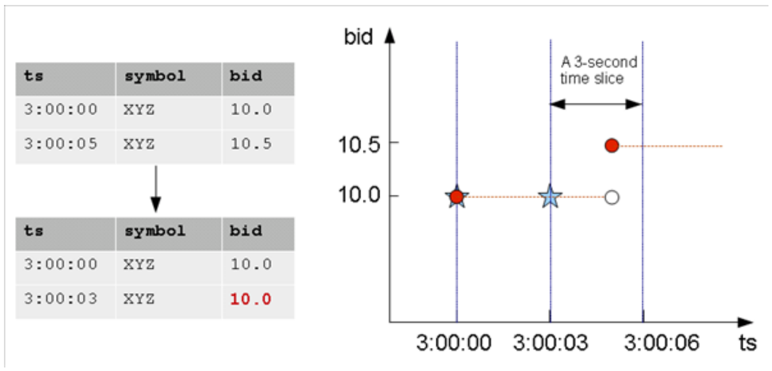
A common interpolation scheme used on financial data is to set the bid price to *the last seen value so far*. This scheme is referred to as **constant interpolation**, in which HP Vertica computes a new value based on the previous input records.

Note: Constant is HP Vertica's default interpolation scheme. Another interpolation scheme, linear, is discussed in an upcoming topic.

Returning to the problem query, here is the table output, which shows a 5-second lag between bids at 03:00:00 and 03:00:05:

```
=> SELECT * FROM TickStore;
         ts          | symbol | bid
---------------------+--------+------
 2009-01-01 03:00:00 | XYZ    |   10
 2009-01-01 03:00:05 | XYZ    | 10.5
(2 rows)
```

Using constant interpolation, the interpolated bid price of XYZ remains at $10.0 at 3:00:03, which falls between the two known data inputs (3:00:00 PM and 3:00:05). At 3:00:05, the value changes to $10.5. The known data points are represented by a red dot, and the interpolated value at 3:00:03 is represented by the blue star.



ts	symbol	bid
3:00:00	XYZ	10.0
3:00:05	XYZ	10.5

ts	symbol	bid
3:00:00	XYZ	10.0
3:00:03	XYZ	10.0

In order to write a query that makes the input rows more uniform, you first need to understand the TIMESERIES clause and time series aggregate functions.

The TIMESERIES Clause and Aggregates

The SELECT..TIMESERIES clause and time series aggregates help solve the problem of gaps in input records by normalizing the data into 3-second time slices and interpolating the bid price when it finds gaps.

The TIMESERIES Clause

The TIMESERIES clause is an important component of time series analytics computation. It performs gap filling and interpolation (GFI) to generate time slices missing from the input records. The clause applies to the timestamp columns/expressions in the data, and takes the following form:

```
TIMESERIES slice_time AS 'length_and_time_unit_expression'
OVER ( ... [ window_partition_clause  [ , ... ] ]
... ORDER BY time_expression )
... [ ORDER BY table_column [ , ... ] ]
```

Note: The TIMESERIES clause requires an ORDER BY operation on the timestamp column.

Time Series Aggregate (TSA) Functions

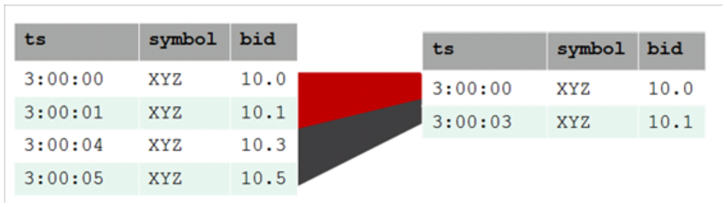
Timeseries Aggregate (TSA) functions evaluate the values of a given set of variables over time and group those values into a window for analysis and aggregation.

TSA functions process the data that belongs to each time slice. One output row is produced per time slice or per partition per time slice if a partition expression is present.

The following table shows 3-second time slices where:

- The first two rows fall within the first time slice, which runs from 3:00:00 to 3:00:02. These are the input rows for the TSA function's output for the time slice starting at 3:00:00.
- The second two rows fall within the second time slice, which runs from 3:00:03 to 3:00:05. These are the input rows for the TSA function's output for the time slice starting at 3:00:03.

  The result is the start of each time slice.



ts	symbol	bid
3:00:00	XYZ	10.0
3:00:01	XYZ	10.1
3:00:04	XYZ	10.3
3:00:05	XYZ	10.5

ts	symbol	bid
3:00:00	XYZ	10.0
3:00:03	XYZ	10.1

Example

The following statement uses both the TIMESERIES clause and the TS_FIRST_VALUE TSA function to process the data that belongs to each 3-second time slice. The query returns the values of the `bid` column, as determined by the specified constant interpolation scheme:

```
=> SELECT slice_time, TS_FIRST_VALUE(bid, 'CONST') bid FROM TickStore
   TIMESERIES slice_time AS '3 seconds' OVER(PARTITION by symbol ORDER BY ts);
```

Now the original data inputs (at left) look like the output on the right because HP Vertica interpolated the last known value and filled in the missing datapoint, returning 10 at 3:00:03:

Original query		Interpolated value
`slice_time | bid`	`==>`	`slice_time | bid`
`---------------------+-----`		`---------------------+-----`
`2009-01-01 03:00:00 |  10`		`2009-01-01 03:00:00 |  10`
`2009-01-01 03:00:03 |10.5`		`2009-01-01 03:00:03 |  10`
`(2 rows)`		`(2 rows)`

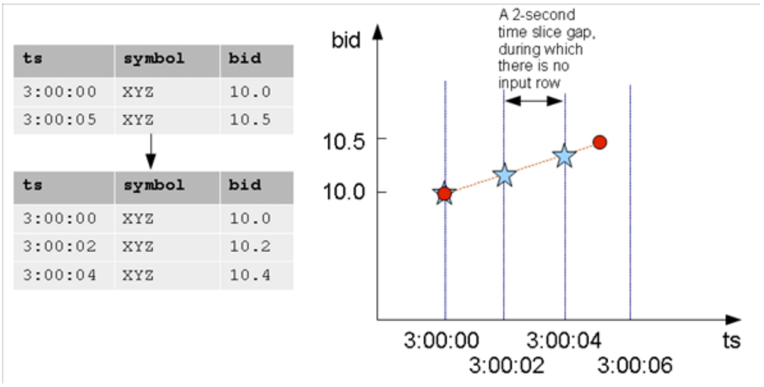
Linear Interpolation

Instead of interpolating data points based on the last seen value (Constant Interpolation), linear interpolation is where HP Vertica interpolates values in a linear slope based on the specified time slice.

The query that follows uses linear interpolation to place the input records in 2-second time slices and return the first bid value for each symbol/time slice combination (the value at the start of the time slice):

```
=> SELECT slice_time, TS_FIRST_VALUE(bid, 'LINEAR') bid FROM Tickstore
   TIMESERIES slice_time AS '2 seconds' OVER(PARTITION BY symbol ORDER BY ts);
     slice_time      | bid
---------------------+------
 2009-01-01 03:00:00 |   10
 2009-01-01 03:00:02 | 10.2
 2009-01-01 03:00:04 | 10.4
(3 rows)
```

The following figure illustrates the previous query results, showing the 2-second time gaps (3:00:02 and 3:00:04) in which no input record occurs. Note that the interpolated bid price of `XYZ` changes to 10.2 at 3:00:02 and 10.3 at 3:00:03 and 10.4 at 3:00:04, all of which fall between the two known data inputs (3:00:00 and 3:00:05). At 3:00:05, the value would change to 10.5.



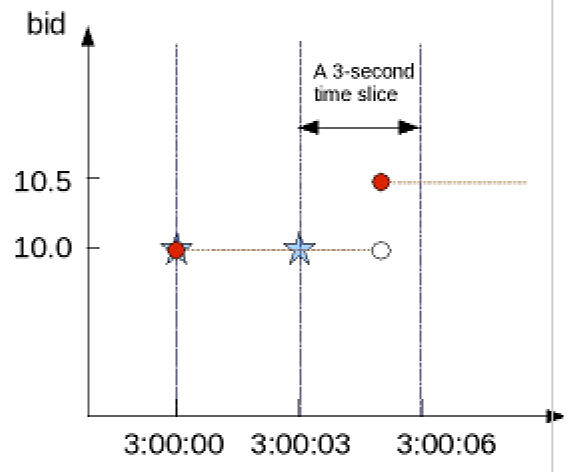
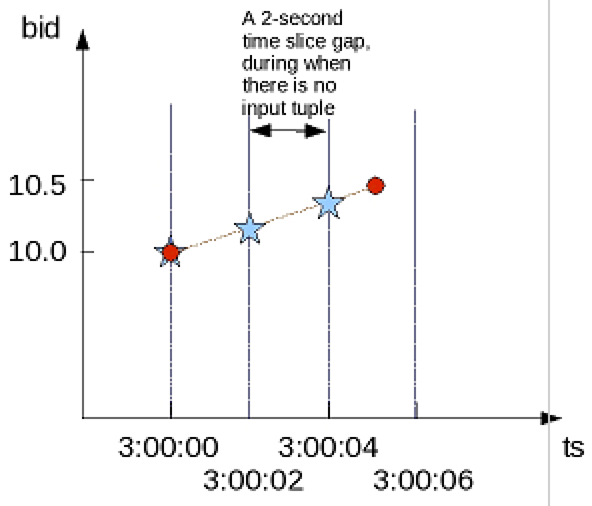
ts	symbol	bid
3:00:00	XYZ	10.0
3:00:05	XYZ	10.5

ts	symbol	bid
3:00:00	XYZ	10.0
3:00:02	XYZ	10.2
3:00:04	XYZ	10.4

bid
A 2-second time slice gap, during which there is no input row
10.5
10.0
3:00:00
3:00:04
ts
3:00:02
3:00:06

Note: The known data points above are represented by a red dot, and the interpolated values are represented by blue stars.

The following is a side-by-side comparison of constant and linear interpolation schemes.

CONST interpolation	**LINEAR interpolation**
	

GFI Examples

This topic illustrates some of the queries you can write using the constant and linear interpolation schemes.

Constant Interpolation

The first query uses TS_FIRST_VALUE() and the TIMESERIES clause to place the input records in 3-second time slices and return the first bid value for each symbol/time slice combination (the value at the start of the time slice).

Note: The TIMESERIES clause requires an ORDER BY operation on the TIMESTAMP column.

```
=> SELECT slice_time, symbol, TS_FIRST_VALUE(bid) AS first_bid FROM TickStore
   TIMESERIES slice_time AS '3 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

Because the bid price of stock `XYZ` is 10.0 at 3:00:03, the `first_bid` value of the second time slice, which starts at 3:00:03 is till 10.0 (instead of 10.5) because the input value of 10.5 does not occur until 3:00:05. In this case, the interpolated value is inferred from the last value seen on stock `XYZ` for time 3:00:03:

```
     slice_time      | symbol | first_bid
---------------------+--------+-----------
 2009-01-01 03:00:00 | XYZ    |        10
 2009-01-01 03:00:03 | XYZ    |        10
(2 rows)
```

The next example places the input records in 2-second time slices to return the first bid value for each symbol/time slice combination:

```
=> SELECT slice_time, symbol, TS_FIRST_VALUE(bid) AS first_bid FROM TickStore
   TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

The result now contains three records in 2-second increments, all of which occur between the first input row at 03:00:00 and the second input row at 3:00:05. Note that the second and third output record correspond to a time slice where there is no input record:

```
     slice_time      | symbol | first_bid
---------------------+--------+-----------
 2009-01-01 03:00:00 | XYZ    |        10
2009-01-01 03:00:02 | XYZ    |        10
2009-01-01 03:00:04 | XYZ    |        10
(3 rows)
```

Using the same table schema, the next query uses TS_LAST_VALUE(), with the TIMESERIES clause to return the last values of each time slice (the values at the end of the time slices).

Note: Time series aggregate functions process the data that belongs to each time slice. One output row is produced per time slice or per partition per time slice if a partition expression is present.

```
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid) AS last_bid FROM TickStore
   TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

Notice that the last value output row is 10.5 because the value 10.5 at time 3:00:05 was the last point inside the 2-second time slice that started at 3:00:04:

```
     slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |       10
 2009-01-01 03:00:02 | XYZ    |       10
 2009-01-01 03:00:04 | XYZ    |     10.5
(3 rows)
```

Remember that because constant interpolation is the default, the same results are returned if you write the query using the CONST parameter as follows:

```
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid, 'CONST') AS last_bid FROM TickStore
   TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

Linear Interpolation

Based on the same input records described in the constant interpolation examples, which specify 2-second time slices, the result of TS_LAST_VALUE with linear interpolation is as follows:

```
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid, 'linear') AS last_bid   FROM TickStore
   TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

In the results, no last_bid value is returned for the last row because the query specified TS_LAST_VALUE, and there is no data point after the 3:00:04 time slice to interpolate.

```
     slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |     10.2
 2009-01-01 03:00:02 | XYZ    |     10.4
 2009-01-01 03:00:04 | XYZ    |
(3 rows)
```

Using Multiple Time Series Aggregate Functions

Multiple time series aggregate functions can exists in the same query. They share the same *gap-filling* policy as defined in the TIMESERIES clause; however, each time series aggregate function can specify its own interpolation policy. In the following example, there are two constant and one linear interpolation schemes, but all three functions use a three-second time slice:

```
=> SELECT slice_time, symbol,
       TS_FIRST_VALUE(bid, 'const') fv_c,
       TS_FIRST_VALUE(bid, 'linear') fv_l,
       TS_LAST_VALUE(bid, 'const') lv_c
   FROM TickStore
   TIMESERIES slice_time AS '3 seconds' OVER(PARTITION BY symbol ORDER BY ts);
```

In the following output, the original output is compared to output returned by multiple time series aggregate functions.

```
    ts    | symbol | bid      ==>       slice_time      | symbol | fv_c | fv_l | lv_c
----------+--------+------          ---------------------+--------+------+------+------
 03:00:00 | XYZ    |   10            2009-01-01 03:00:00 | XYZ    |   10 |   10 |   10
 03:00:05 | XYZ    | 10.5            2009-01-01 03:00:03 | XYZ    |   10 | 10.3 | 10.5
(2 rows)                            (2 rows)
```

Using the Analytic LAST_VALUE() Function

Here's an example using LAST_VALUE(), so you can see the difference between it and the GFI syntax.

```
=> SELECT *, LAST_VALUE(bid) OVER(PARTITION by symbol ORDER BY ts)
   AS "last bid" FROM TickStore;
```

There is no gap filling and interpolation to the output values.

```
         ts          | symbol | bid  | last bid
---------------------+--------+------+----------
 2009-01-01 03:00:00 | XYZ    |   10 |       10
 2009-01-01 03:00:05 | XYZ    | 10.5 |     10.5
(2 rows)
```

Using slice_time

In a TIMESERIES query, you cannot use the column `slice_time` in the WHERE clause because the WHERE clause is evaluated before the TIMESERIES clause, and the `slice_time` column is not generated until the TIMESERIES clause is evaluated. For example, HP Vertica does not support the following query:

```
=> SELECT symbol, slice_time, TS_FIRST_VALUE(bid IGNORE NULLS) AS fv
   FROM TickStore
   WHERE slice_time = '2009-01-01 03:00:00'
   TIMESERIES slice_time as '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
ERROR:  Time Series timestamp alias/Time Series Aggregate Functions not allowed in WHERE
clause
```

Instead, you could write a subquery and put the predicate on `slice_time` in the outer query:

```
=> SELECT * FROM (
      SELECT symbol, slice_time,
        TS_FIRST_VALUE(bid IGNORE NULLS) AS fv
      FROM TickStore
      TIMESERIES slice_time AS '2 seconds'
      OVER (PARTITION BY symbol ORDER BY ts) ) sq
   WHERE slice_time = '2009-01-01 03:00:00';
 symbol |     slice_time      | fv
--------+---------------------+----
 XYZ    | 2009-01-01 03:00:00 | 10
(1 row)
```

Creating a Dense Time Series

The TIMESERIES clause provides a convenient way to create a dense time series for use in an outer join with fact data. The results represent every time point, rather than just the time points for which data exists.

The examples that follow use the same TickStore schema described in Gap Filling and Interpolation (GFI), along with the addition of a new inner table for the purpose of creating a join:

```
=> CREATE TABLE inner_table (
       ts TIMESTAMP,
       bid FLOAT
   );
=> CREATE PROJECTION inner_p (ts, bid) as SELECT * FROM inner_table
   ORDER BY ts, bid UNSEGMENTED ALL NODES;
=> INSERT INTO inner_table VALUES ('2009-01-01 03:00:02', 1);
=> INSERT INTO inner_table VALUES ('2009-01-01 03:00:04', 2);
=> COMMIT;
```

You can create a simple union between the start and end range of the timeframe of interest in order to return every time point. This example uses a 1-second time slice:

```
=> SELECT ts FROM (
     SELECT '2009-01-01 03:00:00'::TIMESTAMP AS time FROM TickStore
     UNION
     SELECT '2009-01-01 03:00:05'::TIMESTAMP FROM TickStore) t
   TIMESERIES ts AS '1 seconds' OVER(ORDER BY time);
         ts
---------------------
 2009-01-01 03:00:00
 2009-01-01 03:00:01
 2009-01-01 03:00:02
 2009-01-01 03:00:03
 2009-01-01 03:00:04
 2009-01-01 03:00:05
(6 rows)
```

The next query creates a union between the start and end range of the timeframe using 500-millisecond time slices:

```
=> SELECT ts FROM (
     SELECT '2009-01-01 03:00:00'::TIMESTAMP AS time
     FROM TickStore
     UNION
     SELECT '2009-01-01 03:00:05'::TIMESTAMP FROM TickStore) t
   TIMESERIES ts AS '500 milliseconds' OVER(ORDER BY time);
          ts
-----------------------
 2009-01-01 03:00:00
 2009-01-01 03:00:00.5
 2009-01-01 03:00:01
 2009-01-01 03:00:01.5
 2009-01-01 03:00:02
 2009-01-01 03:00:02.5
 2009-01-01 03:00:03
 2009-01-01 03:00:03.5
 2009-01-01 03:00:04
 2009-01-01 03:00:04.5
 2009-01-01 03:00:05
(11 rows)
```

The following query creates a union between the start- and end-range of the timeframe of interest using 1-second time slices:

```
=> SELECT * FROM (
     SELECT ts FROM (
       SELECT '2009-01-01 03:00:00'::timestamp AS time FROM TickStore
       UNION
       SELECT '2009-01-01 03:00:05'::timestamp FROM TickStore) t
       TIMESERIES ts AS '1 seconds' OVER(ORDER BY time) ) AS outer_table
   LEFT OUTER JOIN inner_table ON outer_table.ts = inner_table.ts;
```

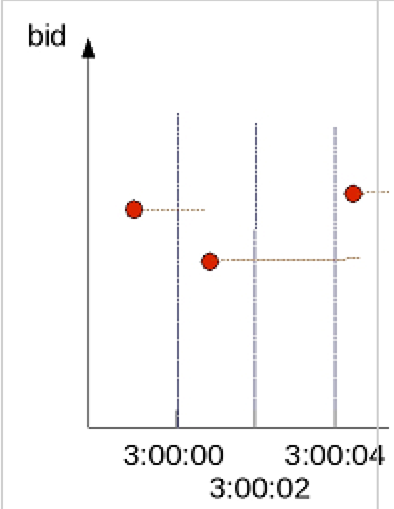
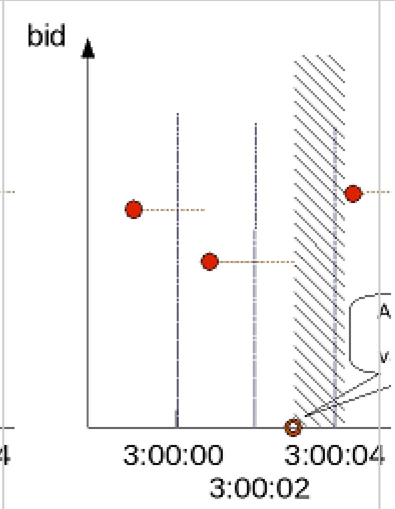
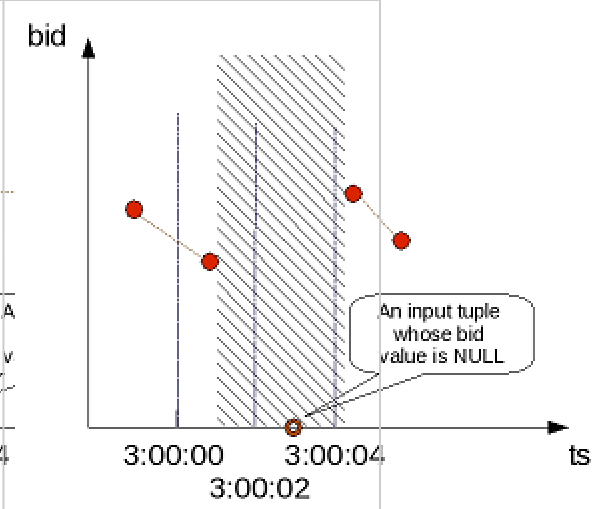
The union returns a complete set of records from the left-joined table with the matched records in the right-joined table. Where the query found no match, it extends the right side column with null values:

```
         ts          |         ts          | bid
---------------------+---------------------+-----
 2009-01-01 03:00:00 |                     |
 2009-01-01 03:00:01 |                     |
 2009-01-01 03:00:02 | 2009-01-01 03:00:02 |   1
 2009-01-01 03:00:03 |                     |
 2009-01-01 03:00:04 | 2009-01-01 03:00:04 |   2
 2009-01-01 03:00:05 |                     |
(6 rows)
```

When Time Series Data Contains Null Values

Null values are not common inputs for gap-filling and interpolation (GFI) computation, but if null values do exist, you can use time series aggregate functions (TS_FIRST_VALUE/TS_LAST_ VALUE) with the IGNORE NULLS arguments to affect output of the interpolated values. The TSA functions are treated similarly to their analytic counterparts (FIRST_VALUE/LAST_VALUE) in that if the timestamp itself is null HP Vertica filter out those rows before gap filling and interpolation occurs.

The three images below will illustrate the points that follow on how HP Vertica handles time series data that contains null values.

Figure 1. Interpolated bid values when the input has no NULLs	**Figure 2.** CONST-interpolated bid values when the input has NULL values	**Figure 3.** LINEAR-interpolated bid values when the input has NULL values
		

Constant Interpolation with Null Values

Figure 1 illustrates a default (constant) interpolation result on four input rows where none of the inputs contains a NULL value. Figure 2 shows the same input rows with the addition of another input record whose bid value is NULL, and whose timestamp (ts) value is 3:00:03.

For constant interpolation, the bid value starting at 3:00:03 is null until the next non-null bid value appears in time. In Figure 2, the presence of the null row makes the interpolated bid value null in the time interval denoted by the shaded region. As a result, if `TS_FIRST_VALUE(bid)` is evaluated with constant interpolation on the time slice that begins at 3:00:02, its output is non-null. However, `TS_FIRST_VALUE(bid)` on the next time slice produces null. If the last value of the 3:00:02 time slice is null, the first value for the next time slice (3:00:04) is null. However, if you were to use a TSA function with IGNORE NULLS, then the value at 3:00:04 would be the same value as it was at 3:00:02.

To illustrate, insert a new row into the TickStore table at 03:00:03 with a null bid value, HP Vertica will output a row for the 03:00:02 record with a null value but no row for the 03:00:03 input:

```
=> INSERT INTO tickstore VALUES('2009-01-01 03:00:03', 'XYZ', NULL);
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid) AS last_bid FROM TickStore
-> TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
     slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |       10
 2009-01-01 03:00:02 | XYZ    |
 2009-01-01 03:00:04 | XYZ    |     10.5
(3 rows)
```

If you specify IGNORE NULLS, HP Vertica fills in the missing data point using a constant interpolation scheme. Here, the bid price at 03:00:02 is interpolated to the last known input record for bid, which was $10 at 03:00:00:

```
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid IGNORE NULLS) AS last_bid FROM TickStore
-> TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
     slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |       10
 2009-01-01 03:00:02 | XYZ    |       10
 2009-01-01 03:00:04 | XYZ    |     10.5
(3 rows)
```

Now if you were to insert a row where the timestamp column contained a null value, HP Vertica would filter out that row before gap filling and interpolation occurred.

```
=> INSERT INTO tickstore VALUES(NULL, 'XYZ', 11.2);
=> SELECT slice_time, symbol, TS_LAST_VALUE(bid) AS last_bid FROM TickStore
-> TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
```

Notice there is no output for the 11.2 bid row:

```
     slice_time      | symbol | last_bid
---------------------+--------+----------
 2009-01-01 03:00:00 | XYZ    |       10
 2009-01-01 03:00:02 | XYZ    |
 2009-01-01 03:00:04 | XYZ    |     10.5
(3 rows)
```

Linear Interpolation with Null Values

For linear interpolation, the interpolated bid value becomes null in the time interval, which is represented by the shaded region in Figure 3. In the presence of an input null value at 3:00:03, HP Vertica cannot linearly interpolate the bid value around that time point.

HP Vertica takes the closest non null value on either side of the time slice and uses that value. For example, if you use a linear interpolation scheme and you do not specify IGNORE NULLS, and your data has one real value and one null, the result is null. If the value on either side is null, the result is null. Therefore, to evaluate `TS_FIRST_VALUE(bid)` with linear interpolation on the time slice that begins at 3:00:02, its output is null. `TS_FIRST_VALUE(bid)` on the next time slice remains null.

```
=> SELECT slice_time, symbol, TS_FIRST_VALUE(bid, 'linear') AS fv_l FROM TickStore
-> TIMESERIES slice_time AS '2 seconds' OVER (PARTITION BY symbol ORDER BY ts);
     slice_time      | symbol | fv_l
---------------------+--------+------
 2009-01-01 03:00:00 | XYZ    |   10
 2009-01-01 03:00:02 | XYZ    |
 2009-01-01 03:00:04 | XYZ    |
(3 rows)
```

Event Series Joins

An **event series** join is an HP Vertica SQL extension that enables the analysis of two series when their measurement intervals don't align precisely, such as with mismatched timestamps. You can compare values from the two series directly, rather than having to normalize the series to the same measurement interval.

Event series joins are an extension of Outer Joins, but instead of padding the non-preserved side with NULL values when there is no match, the event series join pads the non-preserved side values that it **interpolates** from the previous value.

The difference in how you write a regular join versus an event series join is the INTERPOLATE predicate, which is used in the ON clause. For example, the following two statements show the differences, which are shown in greater detail in Writing Event Series Joins.

Regular full outer join	Event series join
`SELECT * FROM hTicks h FULL OUTER JOIN aTicks a ON (h.time = a.time);`	`SELECT * FROM hTicks h FULL OUTER JOIN aTicks a ON (h.time` **`INTERPOLATE PREVIOUS VALUE`** `a.time);`

Similar to regular joins, an event series join has inner and outer join modes, which are described in the topics that follow.

For full syntax, including notes and restrictions, see INTERPOLATE in the SQL Reference Manual

Sample Schema for Event Series Joins Examples

If you don't plan to run the queries and just want to look at the examples, you can skip this topic and move straight to Writing Event Series Joins.

Schema of hTicks and aTicks Tables

The examples that follow use the following hTicks and aTicks tables schemas:

```
CREATE TABLE hTicks (
   stock VARCHAR(20),
   time TIME,
   price NUMERIC(8,2)
);
CREATE TABLE aTicks (
   stock VARCHAR(20),
   time TIME,
   price NUMERIC(8,2)
);
```

Although TIMESTAMP is more commonly used for the event series column, the examples in this topic use TIME to keep the output simple.

```
INSERT INTO hTicks VALUES ('HPQ', '12:00', 50.00);
INSERT INTO hTicks VALUES ('HPQ', '12:01', 51.00);
INSERT INTO hTicks VALUES ('HPQ', '12:05', 51.00);
INSERT INTO hTicks VALUES ('HPQ', '12:06', 52.00);
INSERT INTO aTicks VALUES ('ACME', '12:00', 340.00);
INSERT INTO aTicks VALUES ('ACME', '12:03', 340.10);
INSERT INTO aTicks VALUES ('ACME', '12:05', 340.20);
INSERT INTO aTicks VALUES ('ACME', '12:05', 333.80);
COMMIT;
```

Output of the two tables:

hTicks	aTicks

`=> SELECT * FROM hTicks;`
Notice there are no entry records between 12:02-12:04:
`stock |   time   | price`
`-------+----------+-------`
`HPQ   | 12:00:00 | 50.00`
`HPQ   | 12:01:00 | 51.00`
`HPQ   | 12:05:00 | 51.00`
`HPQ   | 12:06:00 | 52.00`
`(4 rows)`
	`=> SELECT * FROM aTicks;`
	Notice there are no entry records at 12:01, 12:02 and at 12:04:
	`stock |   time   | price`
	`-------+----------+--------`
	`ACME  | 12:00:00 | 340.00`
	`ACME  | 12:03:00 | 340.10`
	`ACME  | 12:05:00 | 340.20`
	`ACME  | 12:05:00 | 333.80`
	`(4 rows)`

Example Query Showing Gaps

A full outer join shows the gaps in the timestamps:

```
=> SELECT * FROM hTicks h FULL OUTER JOIN aTicks a ON h.time = a.time;
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 |       |          |
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 |       |          |
       |          |       | ACME  | 12:03:00 | 340.10
(6 rows)
```

Schema of Bid and Asks Tables

The examples that follow use the following hTicks and aTicks tables schemas:

```
CREATE TABLE bid(stock VARCHAR(20), time TIME, price NUMERIC(8,2));
CREATE TABLE ask(stock VARCHAR(20), time TIME, price NUMERIC(8,2));
INSERT INTO bid VALUES ('HPQ', '12:00', 100.10);
INSERT INTO bid VALUES ('HPQ', '12:01', 100.00);
INSERT INTO bid VALUES ('ACME', '12:00', 80.00);
INSERT INTO bid VALUES ('ACME', '12:03', 79.80);
INSERT INTO bid VALUES ('ACME', '12:05', 79.90);
INSERT INTO ask VALUES ('HPQ', '12:01', 101.00);
INSERT INTO ask VALUES ('ACME', '12:00', 80.00);
INSERT INTO ask VALUES ('ACME', '12:02', 75.00);
COMMIT;
```

Output of the two tables:

bid

```
=> SELECT * FROM bid;
```

Notice there are no entry records for stock `ORCL` at 12:02 and at 12:04:

```
 stock |   time   | price
-------+----------+--------
 HPQ   | 12:00:00 | 100.10
 HPQ   | 12:01:00 | 100.00
 ACME  | 12:00:00 |  80.00
 ACME  | 12:03:00 |  79.80
 ACME  | 12:05:00 |  79.90
(5 rows)
```

ask

```
=> SELECT * FROM ask;
```

Notice there are no entry records for stock IBM at 12:00 and none for `ORCL` at 12:01:

```
 stock |   time   | price
-------+----------+--------
 HPQ   | 12:01:00 | 101.00
 ACME  | 12:00:00 |  80.00
 ACME  | 12:02:00 |  75.00
(3 rows)
```

Example Query Showing Gaps

A full outer join shows the gaps in the timestamps:

```
=> SELECT * FROM bid b FULL OUTER JOIN ask a ON b.time = a.time;
 stock |   time   | price  | stock |   time   | price
-------+----------+--------+-------+----------+--------
 HPQ   | 12:00:00 | 100.10 | ACME  | 12:00:00 |  80.00
 HPQ   | 12:01:00 | 100.00 | HPQ   | 12:01:00 | 101.00
 ACME  | 12:00:00 |  80.00 | ACME  | 12:00:00 |  80.00
 ACME  | 12:03:00 |  79.80 |       |          |
 ACME  | 12:05:00 |  79.90 |       |          |
       |          |        | ACME  | 12:02:00 |  75.00
(6 rows)
```

Writing Event Series Joins

The examples in this topic contains mismatches between timestamps—just as you'd find in real life situations; for example, there could be a period of inactivity on stocks where no trade occurs, which can present challenges when you want to compare two stocks whose timestamps don't match.

The hTicks and aTicks Tables

As described in the example ticks schema, tables, `hTicks` is missing input rows for 12:02, 12:03, and 12:04, and `aTicks` is missing inputs at 12:01, 12:02, and 12:04.

hTicks	aTicks
`=> SELECT * FROM hTicks;`	`=> SELECT * FROM aTicks;`
` stock |   time   | price`	` stock |   time   | price`
`-------+----------+-------`	`-------+----------+--------`
` HPQ   | 12:00:00 | 50.00`	` ACME  | 12:00:00 | 340.00`
` HPQ   | 12:01:00 | 51.00`	` ACME  | 12:03:00 | 340.10`
` HPQ   | 12:05:00 | 51.00`	` ACME  | 12:05:00 | 340.20`
` HPQ   | 12:06:00 | 52.00`	` ACME  | 12:05:00 | 333.80`
`(4 rows)`	`(4 rows)`

Querying Event Series Data with Full Outer Joins

Using a traditional full outer join, this query find a match between tables hTicks and aTicks at 12:00 and 12:05 and pads the missing data points with NULL values.

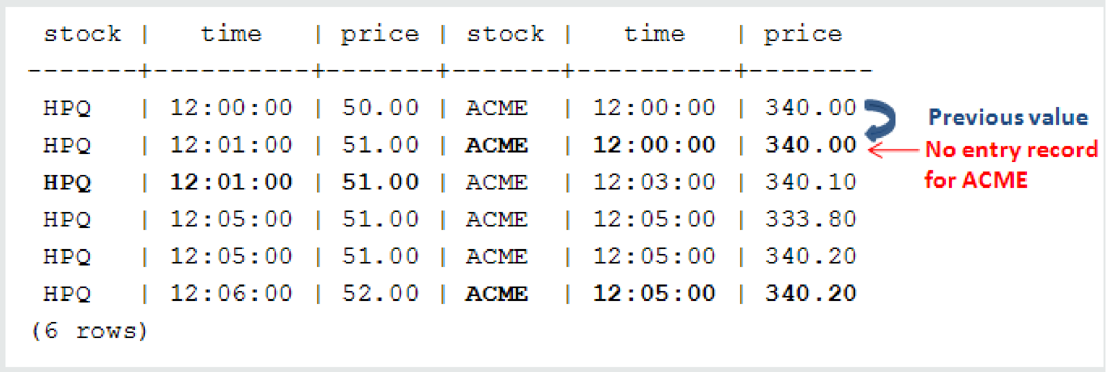
```
=> SELECT * FROM hTicks h FULL OUTER JOIN aTicks a ON (h.time = a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 |       |          |
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 |       |          |
       |          |       | ACME  | 12:03:00 | 340.10
(6 rows)
```

To replace the gaps with **interpolated** values for those missing data points, use the INTERPOLATE predicate to create an **event series** join. The join condition is restricted to the ON clause, which evaluates the equality predicate on the timestamp columns from the two input tables. In other words, for each row in outer table hTicks, the ON clause predicates are evaluated for each combination of each row in the inner table aTicks.

Simply rewrite the full outer join query to use the INTERPOLATE predicate with the required PREVIOUS VALUE keywords. Note that a full outer join on event series data is the most common scenario for event series data, where you keep all rows from both tables

```
=> SELECT * FROM hTicks h FULL OUTER JOIN aTicks a
   ON (h.time INTERPOLATE PREVIOUS VALUE a.time);
```

HP Vertica **interpolates** the missing values (which appear as NULL in the full outer join) using that table's previous value:



```
 stock |   time    | price | stock |   time    | price
-------+-----------+-------+-------+-----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 | ACME  | 12:03:00 | 340.10
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 | ACME  | 12:05:00 | 340.20
(6 rows)
```

Note: The output ordering above is different from the regular full outer join because in the event series join, interpolation occurs independently for each stock (hTicks and aTicks), where the data is partitioned and sorted based on the equality predicate. This means that interpolation occurs within, not across, partitions.

If you review the regular full outer join output, you can see that both tables have a match in the time column at 12:00 and 12:05, but at 12:01, there is no entry record for ACME. So the operation interpolates a value for ACME (`ACME,12:00,340`) based on the previous value in the aTicks table.

Querying Event Series Data with Left Outer Joins

You can also use left and right outer joins. You might, for example, decide you want to preserve only hTicks values. So you'd write a left outer join:

```
=> SELECT * FROM hTicks h LEFT OUTER JOIN aTicks a
   ON (h.time INTERPOLATE PREVIOUS VALUE a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 | ACME  | 12:05:00 | 340.20
(5 rows)
```

Here's what the same data looks like using a traditional left outer join:

```
=> SELECT * FROM hTicks h LEFT OUTER JOIN aTicks a ON h.time = a.time;
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:01:00 | 51.00 |       |          |
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
 HPQ   | 12:06:00 | 52.00 |       |          |
(5 rows)
```

Note that a right outer join has the same behavior with the preserved table reversed.

Querying Event Series Data with Inner Joins

Note that INNER event series joins behave the same way as normal ANSI SQL-99 joins, where all gaps are omitted. Thus, there is nothing to interpolate, and the following two queries are equivalent and return the same result set:

A regular inner join:

```
=> SELECT * FROM HTicks h JOIN aTicks a
    ON (h.time INTERPOLATE PREVIOUS VALUE a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
(3 rows)
```

An event series inner join:

```
=> SELECT * FROM HTicks h INNER JOIN aTicks a ON (h.time = a.time);
 stock |   time   | price | stock |   time   | price
-------+----------+-------+-------+----------+--------
 HPQ   | 12:00:00 | 50.00 | ACME  | 12:00:00 | 340.00
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 333.80
 HPQ   | 12:05:00 | 51.00 | ACME  | 12:05:00 | 340.20
(3 rows)
```

The Bid and Ask Tables

Using the example schema for the `bid` and `ask` tables, write a full outer join to interpolate the missing data points:

```
=> SELECT * FROM bid b FULL OUTER JOIN ask a
    ON (b.stock = a.stock AND b.time INTERPOLATE PREVIOUS VALUE a.time);
```

In the below output, the first row for stock HPQ shows nulls because there is no entry record for HPQ before 12:01.

```
 stock |   time   | price  | stock |   time   | price
-------+----------+--------+-------+----------+--------
 ACME  | 12:00:00 |  80.00 | ACME  | 12:00:00 |  80.00
 ACME  | 12:00:00 |  80.00 | ACME  | 12:02:00 |  75.00
 ACME  | 12:03:00 |  79.80 | ACME  | 12:02:00 |  75.00
 ACME  | 12:05:00 |  79.90 | ACME  | 12:02:00 |  75.00
 HPQ   | 12:00:00 | 100.10 |       |          |
 HPQ   | 12:01:00 | 100.00 | HPQ   | 12:01:00 | 101.00
(6 rows)
```

Note also that the same row (`ACME,12:02,75`) from the `ask` table appears three times. The first appearance is because no matching rows are present in the `bid` table for the row in `ask`, so Vertica interpolates the missing value using the ACME value at 12:02 (75.00). The second appearance occurs because the row in `bid` (`ACME,12:05,79.9`) has no matches in `ask`. The row from `ask` that contains (`ACME,12:02,75`) is the closest row; thus, it is used to interpolate the values.

If you write a regular full outer join, you can see where the mismatched timestamps occur:

```
=> SELECT * FROM bid b FULL OUTER JOIN ask a ON (b.time = a.time);
 stock |   time   | price  | stock |   time   | price
-------+----------+--------+-------+----------+--------
 ACME  | 12:00:00 |  80.00 | ACME  | 12:00:00 |  80.00
 ACME  | 12:03:00 |  79.80 |       |          |
 ACME  | 12:05:00 |  79.90 |       |          |
 HPQ   | 12:00:00 | 100.10 | ACME  | 12:00:00 |  80.00
 HPQ   | 12:01:00 | 100.00 | HPQ   | 12:01:00 | 101.00
       |          |        | ACME  | 12:02:00 |  75.00
(6 rows)
```

Event Series Pattern Matching

The SQL MATCH clause syntax (described in the SQL Reference Manual) lets you screen large amounts of historical data in search of event patterns. You specify a pattern as a regular expression and can then search for the pattern within a sequence of input events. MATCH provides subclauses for analytic data partitioning and ordering, and the pattern matching occurs on a contiguous set of rows.

Pattern matching is particularly useful for clickstream analysis where you might want to identify users' actions based on their Web browsing behavior (page clicks). A typical online clickstream funnel is:

Company home page -> product home page -> search -> results -> purchase online

Using the above clickstream funnel, you can search for a match on the user's sequence of web clicks and identify that the user:

- landed on the company home page
- navigated to the product page
- ran a search
- clicked a link from the search results
- made a purchase

Clickstream Funnel Schema

The examples in this topic use this clickstream funnel and the following `clickstream_log` table schema:

```
CREATE TABLE clickstream_log (
  uid INT,             --user ID
  sid INT,             --browsing session ID, produced by previous sessionization computati
on
  ts TIME,             --timestamp that occurred during the user's page visit
  refURL VARCHAR(20),  --URL of the page referencing PageURL
  pageURL VARCHAR(20), --URL of the page being visited
  action CHAR(1)       --action the user took after visiting the page ('P' = Purchase, 'V'
= View)
);
INSERT INTO clickstream_log VALUES (1,100,'12:00','website1.com','website2.com/home', 'V');
INSERT INTO clickstream_log VALUES (1,100,'12:01','website2.com/home','website2.com/floby',
'V');
INSERT INTO clickstream_log VALUES (1,100,'12:02','website2.com/floby','website2.com/shamwo
w', 'V');
insert into clickstream_log values (1,100,'12:03','website2.com/shamwow','website2.com/bu
y', 'P');
insert into clickstream_log values (2,100,'12:10','website1.com','website2.com/home', 'V');
insert into clickstream_log values (2,100,'12:11','website2.com/home','website2.com/forks',
'V');
insert into clickstream_log values (2,100,'12:13','website2.com/forks','website2.com/buy',
'P');
COMMIT;
```

Here's the clickstream_log table's output:

```
=> SELECT * FROM clickstream_log;
 uid | sid |    ts    |        refURL         |       pageURL         | action
-----+-----+----------+-----------------------+-----------------------+--------
   1 | 100 | 12:00:00 | website1.com          | website2.com/home     | V
   1 | 100 | 12:01:00 | website2.com/home     | website2.com/floby    | V
   1 | 100 | 12:02:00 | website2.com/floby    | website2.com/shamwow  | V
   1 | 100 | 12:03:00 | website2.com/shamwow  | website2.com/buy      | P
   2 | 100 | 12:10:00 | website1.com          | website2.com/home     | V
   2 | 100 | 12:11:00 | website2.com/home     | website2.com/forks    | V
   2 | 100 | 12:13:00 | website2.com/forks    | website2.com/buy      | P
(7 rows)
```

Example

This example includes the HP Vertica pattern matching functions to analyze users' browsing history over website2.com. It identifies patterns where the user performed the following tasks:

- Landed on website2.com from another web site (Entry)
- Browsed to any number of other pages (Onsite)
- Made a purchase (Purchase)

In the following statement, pattern P (`Entry Onsite* Purchase`) consist of three event types: Entry, Onsite, and Purchase. When HP Vertica finds a match in the input table, the associated pattern instance must be an event of type Entry followed by 0 or more events of type Onsite, and an event of type Purchase

```
SELECT uid,
       sid,
       ts,
       refurl,
       pageurl,
       action,
       event_name(),
       pattern_id(),
       match_id()
FROM clickstream_log
MATCH
  (PARTITION BY uid, sid ORDER BY ts
   DEFINE
     Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',
     Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',
     Purchase AS PageURL ILIKE     '%website2.com%' AND Action = 'P'
   PATTERN
     P AS (Entry Onsite* Purchase)
   RESULTS ALL ROWS);
```

In the output below, the first four rows represent the pattern for user 1's browsing activity, while the following three rows show user 2's browsing habits.

```
 uid | sid |    ts    |        refurl         |       pageurl         | action | event_name
| pattern_id | match_id
-----+-----+----------+-----------------------+-----------------------+--------+-----------
-+------------+----------
   1 | 100 | 12:00:00 | website1.com          | website2.com/home     | V      | Entry
|          1 |        1
   1 | 100 | 12:01:00 | website2.com/home     | website2.com/floby    | V      | Onsite
|          1 |        2
   1 | 100 | 12:02:00 | website2.com/floby    | website2.com/shamwow  | V      | Onsite
|          1 |        3
   1 | 100 | 12:03:00 | website2.com/shamwow  | website2.com/buy      | P      | Purchase
|          1 |        4
   2 | 100 | 12:10:00 | website1.com          | website2.com/home     | V      | Entry
|          1 |        1
   2 | 100 | 12:11:00 | website2.com/home     | website2.com/forks    | V      | Onsite
|          1 |        2
   2 | 100 | 12:13:00 | website2.com/forks    | website2.com/buy      | P      | Purchase
|          1 |        3
(7 rows)
```

See Also

- MATCH Clause
- Pattern Matching Functions
- Perl Regular Expressions Documentation

Optimizing Query Performance

When you submit a query to HP Vertica for processing, the HP Vertica query optimizer automatically chooses a set of operations to compute the requested result. These operations together are called a *query plan*. The choice of operations can drastically affect the run-time performance and resource consumption needed to compute the query results. Depending on the properties of the projections defined in your database, the query optimizer can choose faster and more efficient operations to compute the query results.

This section describes the different operations that the optimizer uses and how you can get the optimizer to use the most efficient operations to compute the results of your query.

Note: Database response time depends on factors such as type and size of the application query, database design, data size and data types stored, available computational power, and network bandwidth. Adding nodes to a database cluster does not necessarily improve the system response time for every query, especially if the response time is already short, e.g., less then 10 seconds, or the response time is not hardware bound.

First Steps for Improving Query Performance

To improve the performance of your queries, take the steps described in the following sections to ensure that the database is optimized for query performance:

Run Database Designer

Your first step should always be to run Database Designer. Database Designer creates a physical schema for your database that provides optimal query performance. The first time you run Database Designer, create a comprehensive design and make sure Database Designer has relevant sample queries and data on which to base the design. If you develop performance issues later, consider loading additional queries that you run frequently and rerun Database Designer to create an incremental design.

For more information about running Database Designer, see Using the Database Designer.

When you run Database Designer, choose the **Update Statistics** option. The HP Vertica query optimizer uses statistics about the data to create a query plan. Statistics help the optimizer determine:

- Multiple eligible projections to answer the query
- The best order in which to perform joins
- Data distribution algorithms, such as broadcast and re-segmentation

If your statistics become out of date, run the ANALYZE_STATISTICS or ANALYZE_HISTOGRAM function to update statistics for a given schema, table, or column. For more information, see Collecting Database Statistics.

Check Query Events Proactively

The QUERY_EVENTS system table identifies whether there are issues with the planning phase of a query. In particular, the following values in the EVENT_TYPE column might indicate a problem that needs to be addressed:

- PREDICATE OUTSIDE HISTOGRAM: The optimizer encountered a predicate that was false for the entire histogram created by ANALYZE_STATISTICS or ANALYZE HISTOGRAM.
- NO HISTOGRAM: The optimizer encountered a predicate on a column for which it does not have a histogram.
- MEMORY LIMIT HIT: The optimizer used all its allocated memory creating the query plan. If you see this value, simplify your query instead of increasing the memory allocation.

The QUERY_EVENTS table also gives a detailed description of each issue and suggests solutions. For more information about this system table, see QUERY_EVENTS.

Review the Query Plan

A query plan is a sequence of step-like **paths** that the Vertica query optimizer selects to access or alter information in your Vertica database. There are two ways to get information about the query plan:

- Run the EXPLAIN command. Each step (path) represents a single operation that the optimizer uses for its execution strategy.

- Query the QUERY_PLAN_PROFILES system table. This table provides detailed execution status for currently running queries. Output from the QUERY_PLAN_PROFILES table shows the real-time flow of data and the time and resources consumed for each path in each query plan.

For more information, see How to get query plan information.

Optimizing Encoding to Improve Query Performance

You can potentially make queries faster by changing the encoding of the columns. Encoding reduces the on-disk size of your data so that the amount of I/O required for queries is reduced, resulting in faster execution times. Make sure that all columns and projections included in the query are using the correct encoding for the data. To do this, take the following steps:

1. Run Database Designer to create an incremental design. Database Designer implements the optimum encoding and projection design.

2. After creating the incremental design, update statistics using the ANALYZE_STATISTICS function.

3. Run EXPLAIN with one or more of the queries you submitted to the design to make sure it is using the new projections.

Alternatively, run DESIGNER_DESIGN_PROJECTION_ENCODINGS to re-evaluate the current encoding and update it if necessary.

Improving the Compression of FLOAT Columns

If you are seeing slow performance or a large storage footprint with your FLOAT data, evaluate the data and your business needs to to see if it can be contained in a NUMERIC column with a precision of 18 digits or less. Converting a FLOAT column to a NUMERIC column can improve data compression, reduce the on-disk size of your database, and improve the performance of queries on that column.

When you define a NUMERIC data type, you specify the precision and the scale; NUMERIC data are exact representations of data. FLOAT data types represent variable precision and approximate values; they take up more space in the database.

Converting FLOAT columns to NUMERIC columns is most effective when:

- The precision of the NUMERIC column is 18 digits or less. Vertica has finetuned the performance of NUMERIC data for the common case of 18 digits of precision. Vertica does not recommend converting FLOAT columns to NUMERIC columns that require a precision of more than 18 digits.

- The precision of the FLOAT values is bounded, and the values will all fall within a specified precision for a NUMERIC column. One example is monetary values like product prices or financial transaction amounts. For example, a column defined as NUMERIC(11,2) can accommodate prices from 0 to a few million dollars and can store cents, and compresses more efficiently than a FLOAT column.

If you try to load a value into a NUMERIC column that exceeds the specified precision, Vertica gives an error and does not load the data. If you assign a value with more decimal digits than the specified scale, the value is rounded to match the specified scale and stored in that column.

For more information, see Numeric Data Types.

Using Run Length Encoding (RLE) to Improve Query Performance

If you run Database Designer and choose to optimize for loads, which minimizes database footprint, Database Designer applies the most appropriate encodings, including RLE, to columns in order to maximize query performance.

In an HP Vertica database, run length encoding (RLE) replaces sequences (runs) of identical data values in a column with a set of pairs, where each pair represents the value and number of occurrences. For example, a gender column might have 47 instances of F and 56 instances of M. Using RLE, HP Vertica can save disk space by storing the pairs (47, F) and (56, M).

The advantage of RLE is that it reduces disk I/O and results in a smaller storage footprint for the database. Use RLE for low-cardinality columns, where the average repetition count is less than 10. For example, a gender column with 47 F values and 56 M values can benefit from RLE. A gender column with 6 F values and 10 M values, where the average repetition count is 8, does not benefit from RLE.

Optimizing Projections for Queries with Predicates

If your query contains one or more predicates, you can modify the projections to improve the query's performance, as described in the following two examples.

Example 1: Queries That Use Date Ranges

This first example shows how to encode data using RLE and change the projection sort order to improve the performance of a query that retrieves all data within a given date range.

Suppose you have a query that looks like this:

```
=> SELECT * FROM trades
   WHERE trade_date BETWEEN '2007-11-01' AND '2007-12-01';
```

To optimize this query, determine whether all of the projections can perform the SELECT operation in a timely manner. Run SELECT COUNT(*) statement for each projection, specifying the date range, and note the response time. For example:

```
=> SELECT COUNT(*) FROM [ projection_name ]
   WHERE trade_date BETWEEN '2007-11-01' AND '2007-12-01;
```

If one or more of the queries is slow, check the uniqueness of the `trade_date` column and determine if it needs to be in the projection's ORDER BY clause and/or can be encoded using **RLE**. RLE replaces sequences of the same data values within a column by a pair that represents the value and a count. For best results, order the columns in the projection from lowest cardinality to highest cardinality, and use RLE to encode the data in low-cardinality columns.

Note: For an example of using sorting and RLE, see Choosing Sort Order: Best Practices.

If the number of unique columns is unsorted, or if the average number of repeated rows is less than 10, `trade_date` is too close to being unique and cannot be encoded using RLE. In this case, add a new column to minimize the search scope.

The following example adds a new column `trade_year`:

1. Determine if the new column `trade_year` returns a manageable result set. The following query returns the data grouped by `trade_year`:

   ```
   => SELECT DATE_TRUNC('trade_year', trade_date), COUNT(*)
      FROM trades
      GROUP BY DATE_TRUNC('trade_year',trade_date);
   ```

2. Assuming that `trade_year = 2007` is near 8k, add a column for `trade_year` to the `trades` table. The `SELECT` statement then becomes:

   ```
   => SELECT * FROM trades
      WHERE trade_year = 2007
      AND trade_date BETWEEN '2007-11-01' AND '2007-12-01';
   ```

   As a result, you have a projection that is sorted on `trade_year`, which can be encoded using RLE.

Example 2: Queries for Tables with a High-Cardinality Primary Key

This example demonstrates how you can modify the projection to improve the performance of queries that select data from a table with a high-cardinality primary key.

Suppose you have the following query:

```
=> SELECT FROM [table]
   WHERE pk IN (12345, 12346, 12347,...);
```

Because the primary key is a high-cardinality column, HP Vertica has to search a large amount of data.

To optimize the schema for this query, create a new column named `buckets` and assign it the value of the primary key divided by 10000. In this example, `buckets=(int) pk/10000`. Use the `buckets` column to limit the search scope as follows:

```
=> SELECT FROM [table]
   WHERE buckets IN (1,...)
   AND pk IN (12345, 12346, 12347,...);
```

Creating a lower cardinality column and adding it to the query limits the search scope and improves the query performance. In addition, if you create a projection where `buckets` is first in the sort order, the query may run even faster.

Optimizing Projections for MERGE Operations

The HP Vertica query optimizer automatically picks the best projections to use for queries, but you can help improve the performance of `MERGE` operations by ensuring projections are designed for optimal use.

Good projection design lets HP Vertica choose the faster merge join between the **target** and **source** tables without having to perform additional sort and data transfer operations.

HP recommends that you first use **Database Designer** to generate a comprehensive design and then customize projections, as needed. Be sure to first review the topics in Planning Your Design. Failure to follow those considerations could result in non-functioning projections.

In the following `MERGE` statement, HP Vertica inserts and/or updates records from the source table's column `b` into the target table's column `a`:

```
=> MERGE INTO target t USING source s ON t.a = s.b WHEN ....
```

HP Vertica can use a local merge join if tables `target` and `source` use one of the following projection designs, where their inputs are pre-sorted through the `CREATE PROJECTION ORDER BY` clause:

- **Replicated** projections that are sorted on:
  - Column `a` for `target`
  - Column `b` for `source`
- **Segmented** projections that are identically segmented on:
  - Column `a` for `target`
  - Column `b` for `source`
  - Corresponding segmented columns

Tip: For best merge performance, the source table should be smaller than the target table.

See Also

- Optimized Versus Non-Optimized MERGE
- Best Practices for Optimizing MERGE Statements

Optimizing GROUP BY Queries

This section explains several ways you can design your projections to optimize the performance of your `GROUP BY` queries.

Partially Sorted GROUPBY

Partially sorted GROUPBY is an optimization for queries for which a projection is sorted on a subset of the aggregated columns.

When processing aggregate queries on large data sets, the HP Vertica optimizer automatically uses the partially sorted GROUPBY optimization to prevent or reduce the chance that hash tables will spill to disk during query processing. This optimization has a significant impact when a query uses two or more DISTINCT aggregate functions, such as COUNT or SUM.

The partially sorted GROUPBY optimization does not optimize queries where the data sets are small enough to fit entirely in main memory.

For two examples of queries that use the partially sorted GROUPBY optimization:

- Partially Sorted GROUPBY with Multiple DISTINCT Aggregate Function Calls
- Partially Sorted GROUPBY When GROUP BY Column Crosses Join

Partially Sorted GROUPBY with Multiple DISTINCT Aggregate Function Calls

The following example shows the EXPLAIN query plan that uses partially sorted GROUPBY. To try this example, take these steps to create and populate tables in the VMart database and then generate the query plan:

```
$ cd /opt/Vertica/examples/VMart_Schema
$ ls vmart_gen
$ vsql
-- create and populate the example tables from the .tbl files
=>\i vmart_create_schema.sql
=>\i vmart_load_data.sql
-- verify table population
=> SELECT COUNT(*) FROM store.store_sales_fact;
 COUNT
---------
5000000
(1 row)
=> CREATE PROJECTION store.store_sales_fact_by_store_and_date AS
SELECT * FROM store.store_sales_fact f
ORDER BY f.store_key, f.date_key;
=> SELECT START_REFRESH(); -- wait a few seconds before running queries
```

The query plan shows that optimizer uses the partially sorted GROUPBY optimization because the GROUP BY column, `store_key`, is one of the sort columns in the projection, `store.store_sales_` `fact_by_store_and_date`, and contains more than one call to a COUNT or SUM function with the DISTINCT keyword. The partially sorted GROUPBY optimization can have a significant impact on the performance of such queries.

```
VMart=> EXPLAIN SELECT COUNT(distinct customer_key) AS cntd_cust,
        store_key,
        COUNT(DISTINCT product_key) AS cntd_prod,
        COUNT(DISTINCT promotion_key) AS cntd_promo,
        SUM(sales_dollar_amount) AS sum_sales_dollar,
        SUM(cost_dollar_amount) AS sum_cost_dollar
 FROM store.store_sales_fact
 GROUP BY store_key
 ORDER BY cntd_cust DESC
 LIMIT 25;

Access Path:

+-SELECT  LIMIT 25 [Cost: 43K, Rows: 25 (NO STATISTICS)] (PATH ID: 0)
|  Output Only: 25 tuples
| +---> SORT [TOPK] [Cost: 43K, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
| |      Order: "Sqry$_1".cntd_cust DESC
| |      Output Only: 25 tuples
| | +---> JOIN MERGEJOIN(inputs presorted) [Cost: 43K, Rows: 10K (NO STATISTICS)] (PATH I
D: 2)
| | |      Join Cond: ("Sqry$_2".store_key <=> "Sqry$_3".store_key)
| | | +-- Outer -> JOIN MERGEJOIN(inputs presorted) [Cost: 32K, Rows: 10K (NO STATISTIC
S)] (PATH ID: 3)
| | | |      Join Cond: ("Sqry$_1".store_key <=> "Sqry$_2".store_key)
| | | | +-- Outer -> SELECT [Cost: 22K, Rows: 10K (NO STATISTICS)] (PATH ID: 4)
| | | | | +---> GROUPBY HASH (SORT OUTPUT) (LOCAL RESEGMENT GROUPS) [Cost: 22K, Rows: 10K
(NO STATISTICS)] (PATH ID: 5)
| | | | |      Aggregates: count(DISTINCT store_sales_fact.customer_key), sum(<SVAR>), su
m(<SVAR>)
| | | | | |      Group By: store_sales_fact.store_key
| | | | | |      Partially sorted keys: 1
| | | | | | +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 22K, Rows: 10K (NO STATIST
ICS)] (PATH ID: 6)
| | | | | | |      Aggregates: sum(store_sales_fact.sales_dollar_amount), sum(store_sale
s_fact.cost_dollar_amount)
| | | | | | |      Group By: store_sales_fact.store_key, store_sales_fact.customer_key
| | | | | | |      Partially sorted keys: 1
| | | | | | | +---> STORAGE ACCESS for store_sales_fact [Cost: 18K, Rows: 5M (NO STATISTI
CS)] (PATH ID: 7)
...
```

Partially Sorted GROUPBY When GROUP BY Column Crosses Join

The following example illustrates a query that uses partially sorted GROUPBY. To try this example, take these steps to create and populate tables in the VMart database and then generate the query plan:

```
$ cd /opt/Vertica/examples/VMart_Schema
$ ls vmart_gen
$ vsql
```

```
-- create and populate the example tables from the .tbl files
=>\i vmart_create_schema.sql
=>\i vmart_load_data.sql
-- verify table population
=> SELECT COUNT(*) FROM store.store_sales_fact;
 COUNT
---------
5000000
(1 row)
=> CREATE PROJECTION store.store_sales_fact_by_store_and_date AS
SELECT * FROM store.store_sales_fact f
ORDER BY f.store_key, f.date_key;
=> SELECT START_REFRESH(); -- wait a few seconds before running queries
```

If the GROUP BY columns are a subset of the sort order and descriptive attributes joined from another table (typically, a dimension table), the optimizer may choose a partially sorted GROUPBY optimization if it estimates that doing so would reduce the cost of query execution.

In the following example, the fact table has an available projection sorted on `store_key`, but the query results are to be grouped on the more descriptive attribute `store_name`. The GROUP BY query can be restructured as a nested query in which the subquery is an optimized query against the fact table, while the outer query joins with the dimension table to provide the `store_name` column:

```
Access Path:
+-SELECT  LIMIT 25 [Cost: 44K, Rows: 25 (NO STATISTICS)] (PATH ID: 0)
|  Output Only: 25 tuples
| +---> SORT [TOPK] [Cost: 44K, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
| |      Order: subq.cntd_cust DESC
| |      Output Only: 25 tuples
| | +---> JOIN MERGEJOIN(inputs presorted) [Cost: 43K, Rows: 10K (NO STATISTICS)] (PATH I
D: 2)
| | |      Join Cond: (subq.store_key = sdim.store_key)
| | | +-- Outer -> SELECT [Cost: 43K, Rows: 10K (NO STATISTICS)] (PATH ID: 3)
| | | | +---> JOIN MERGEJOIN(inputs presorted) [Cost: 43K, Rows: 10K (NO STATISTICS)] (PA
TH ID: 4)
| | | | |      Join Cond: ("Sqry$_2".store_key <=> "Sqry$_3".store_key)
| | | | | +-- Outer -> JOIN MERGEJOIN(inputs presorted) [Cost: 32K, Rows: 10K (NO STATIST
ICS)] (PATH ID: 5)
| | | | | |      Join Cond: ("Sqry$_1".store_key <=> "Sqry$_2".store_key)
| | | | | | +-- Outer -> SELECT [Cost: 22K, Rows: 10K (NO STATISTICS)] (PATH ID: 6)
| | | | | | | +---> GROUPBY HASH (SORT OUTPUT) (LOCAL RESEGMENT GROUPS) [Cost:22K, Rows:
10K (NO STATISTICS)] (PATH ID: 7)
| | | | | | | |      Aggregates: count(DISTINCT store_sales_fact.customer_key), sum(<SVA
R>), sum(<SVAR>)
| | | | | | | |      Group By: store_sales_fact.store_key
| | | | | | | |      Partially sorted keys: 1
| | | | | | | | +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 22K, Rows: 1 0K (NO ST
ATISTICS)] (PATH ID: 8)
| | | | | | | | |      Aggregates: sum(store_sales_fact.sales_dollar_amount), sum(store_s
ales_fact.cost_dollar_amount)
| | | | | | | | |      Group By: store_sales_fact.store_key, store_sales_fact.customer_ke
y
| | | | | | | | |      Partially sorted keys: 1
| | | | | | | | | +---> STORAGE ACCESS for store_sales_fact [Cost: 18K, Rows: 5M (NO
STATISTICS)] (PATH ID: 9)
| | | | | | | | | |      Materialize: store_sales_fact.store_key, store_sales_fact.custom
er_key, store_sales_fact.sales_dollar_amount, store_sales_fact.cost_dollar_amount
| | | | | | | | | |      Runtime Filters: (SIP2(MergeJoin): "Sqry$_1".store_key), (SIP2(M
ergeJoin): "Sqry$_1".store_key), (SIP3(MergeJoin): subq.store_key)
| | | | | | | +-- Inner -> SELECT [Cost: 11K, Rows: 10K (NO STATISTICS)] (PATH ID: 10)
| | | | | | | | +---> GROUPBY HASH (SORT OUTPUT) (LOCAL RESEGMENT GROUPS) [Cost:11K, Rows:
10K (NO STATISTICS)] (PATH ID: 11)
| | | | | | | |      Aggregates: count(DISTINCT store_sales_fact.product_key)
| | | | | | | |      Group By: store_sales_fact.store_key
| | | | | | | |      Partially sorted keys: 1
| | | | | | | | | +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 11K, Rows: 1 0K (NO ST
ATISTICS)] (PATH ID: 12)
| | | | | | | | |      Group By: store_sales_fact.store_key, store_sales_fact.product_key
| | | | | | | | |      Partially sorted keys: 1
| | | | | | | | | | +---> STORAGE ACCESS for store_sales_fact [Cost: 9K, Rows: 5M (NO STATI
STICS)] (PATH ID: 13)
...
```

Avoiding GROUPBY HASH with Projection Design

If your query contains a `GROUP BY` clause, HP Vertica computes the result with either the `GROUPBY PIPELINED` or `GROUPBY HASH` algorithm.

Both algorithms compute the same results and have similar performance when the query produces a small number of distinct groups (typically a thousand per node in the cluster). For queries that contain a large number of groups, `GROUPBY PIPELINED` uses less memory and can be faster but is only used when the input data is pre-sorted on the `GROUP BY` columns.

To improve the performance of a query that has a large number of distinct groups that is currently using the `GROUP BY HASH` algorithm, you can enable the use of the `GROUPBY PIPELINED` algorithm, as this section describes.

To determine which algorithm your query is using, run the EXPLAIN statement on the query.

The three conditions described in this section refer to the following schema.

```
CREATE TABLE sortopt (
    a INT NOT NULL,
    b INT NOT NULL,
    c INT,
    d INT
);
CREATE PROJECTION sortopt_p (
   a_proj,
   b_proj,
   c_proj,
   d_proj )
AS SELECT * FROM sortopt
ORDER BY a,b,c
UNSEGMENTED ALL NODES;
INSERT INTO sortopt VALUES(5,2,13,84);
INSERT INTO sortopt VALUES(14,22,8,115);
INSERT INTO sortopt VALUES(79,9,401,33);
```

Condition #1

All columns in the query's `GROUP BY` clause must be included in the projection's sort columns. If even one column in the GROUP BY clause is excluded from the projection's `ORDER BY` clause, HP Vertica uses `GROUPBY HASH` instead of `GROUPBY PIPELINED`:

Given a projection sort order `ORDER BY a, b, c:`

`GROUP BY a`
`GROUP BY a,b`
`GROUP BY b,a`
`GROUP BY a,b,c`
`GROUP BY c,a,b`
	The query optimizer uses `GROUPBY PIPELINED` because columns `a`, `b`, and `c` are included in the projection sort columns.

`GROUP BY a,b,c,d`	The query optimizer uses `GROUPBY HASH` because column `d` is not part of the projection sort columns.

Condition #2

If the number of columns in the query's `GROUP BY` clause is less than the number of columns in the projection's `ORDER BY` clause, columns in the query's `GROUP BY` clause must occur *first* in the projection's `ORDER BY` clause.

Given a projection sort order `ORDER BY a, b ,c:`

`GROUP BY a`
`GROUP BY a,b`
`GROUP BY b,a`
`GROUP BY a,b,c`
`GROUP BY c,a,b`
	The query optimizer uses `GROUPBY PIPELINED` because columns `a`, `b`, `c` are included in the projection sort columns.

`GROUP BY a,c`	The query optimizer uses `GROUPBY HASH` because columns `a` and `c` do not occur first in the projection sort columns.

Condition #3

If the columns in a query's `GROUP BY` clause do not appear first in the projection's `ORDER BY` clause, then any early-appearing projection sort columns that are missing in the query's `GROUP BY` clause must be present as single-column constant equality predicates in the query's `WHERE` clause.

Given a projection sort order `ORDER BY a, b, c:`

`SELECT a FROM tab WHERE a = 10 GROUP BY b`	The query optimizer uses `GROUPBY PIPELINED` because all columns preceding `b` in the projection sort order appear as constant equality predicates.

`SELECT a FROM tab WHERE a = 10 GROUP BY a, b`	The query optimizer uses `GROUPBY PIPELINED` even if redundant grouping column `a` is present.
`SELECT a FROM tab WHERE a = 10 GROUP BY b, c`	The query optimizer uses `GROUPBY PIPELINED` because all columns preceding `b` and `c` in the projection sort order appear as constant equality predicates.
`SELECT a FROM tab WHERE a = 10 GROUP BY c, b`	The query optimizer uses `GROUPBY PIPELINED` because all columns preceding `b` and `c` in the projection sort order appear as constant equality predicates.
`SELECT a FROM tab WHERE a = 10 GROUP BY c`	The query optimizer uses `GROUPBY HASH` because all columns preceding `c` in the projection sort order do *not* appear as constant equality predicates.

Avoiding Resegmentation During GROUP BY Optimization with Projection Design

To compute the correct result of a query that contains a `GROUP BY` clause, HP Vertica must ensure that all rows with the same value in the `GROUP BY` expressions end up at the same node for final computation. If the projection design already guarantees the data is segmented by the `GROUP BY` columns, no resegmentation is required ar run time.

To avoid resegmentation, the `GROUP BY` clause must contain all the segmentation columns of the projection, but it can also contain other columns.

When your query includes a `GROUP BY` clause and joins, the joins are performed first. The result of the join operation is the input to the `GROUP BY` clause. The segmentation of those intermediate results may not be consistent with the `GROUP BY` clause in your query, resulted in resegmentation at run time.

If your query does not include joins, the `GROUP BY` clauses are processed using the existing database projections.

Examples

Assume the following projection:

```
CREATE PROJECTION … SEGMENTED BY HASH(a,b) ALL NODES
```

The following table explains whether or not resegmentation occurs at run time and why.

`GROUP BY a`	Requires resegmentation at run time. The query does not contain all the projection segmentation columns.

`GROUP BY a, b`	Does not require resegmentation at run time. The `GROUP BY` clause contains all the projection segmentation columns.
`GROUP BY a, b, c`	Does not require resegmentation at run time. The `GROUP BY` clause contains all the projection segmentation columns.
`GROUP BY a+1, b`	Requires resegmentation at run time because of the expression on column a.

To determine if resegmentation will occurs during your `GROUP BY` query, look at the `EXPLAIN` plan.

For example, the following plan uses `GROUPBY PIPELINED` sort optimization and requires resegmentation to perform the `GROUP BY` calculation:

```
+-GROUPBY PIPELINED (RESEGMENT GROUPS) [Cost: 194, Rows: 10K (NO STATISTICS)]
(PATH ID: 1)
```

The following plan uses `GROUPBY PIPELINED` sort optimization, but does not require resegmentation:

```
+-GROUPBY PIPELINED [Cost: 459, Rows: 10K (NO STATISTICS)] (PATH ID: 1)
```

Optimizing DISTINCT in a SELECT Query List

This section describes how to optimize queries that have the `DISTINCT` keyword in their `SELECT` list. The techniques for optimizing DISTINCT queries are similar to the techniques for optimizing GROUP BY queries because when processing queries that use `DISTINCT`, the HP Vertica optimizer rewrites the query as a `GROUP BY` query.

The examples in this section use the following table:

```
CREATE TABLE table1 (
    a INT,
    b INT,
    c INT
);
```

The following section give examples for specific situations:

- If the Query Has No Aggregates in the SELECT List
- Optimizing COUNT (DISTINCT) and Other DISTINCT Aggregates
- If the Query Has a Single DISTINCT Aggregate
- If the Query Has Multiple DISTINCT Aggregates

If the Query Has No Aggregates in the SELECT List

If your query has no aggregates in the `SELECT` list, internally, Vertica treats the query as if it uses `GROUP BY` instead.

For example, you can rewrite the following query:

```
SELECT DISTINCTa, b, c FROM table1;
```

as:

```
SELECT a, b, c FROM table1 GROUP BY a, b, c;
```

For fastest execution, apply the optimization techniques for `GROUP BY` queries described in Optimizing GROUP BY Queries.

Optimizing COUNT (DISTINCT) and Other DISTINCT Aggregates

Computing a `DISTINCT` aggregate generally requires much more work than other aggregates, so if your query can be expressed without `DISTINCT` aggregates, the query executes faster. Similarly, a query that uses a single `DISTINCT` aggregate requires less time and resources to compute than a query with multiple `DISTINCT` aggregates. Internally, Vertica handles queries with a single `DISTINCT` aggregate differently from queries with multiple `DISTINCT` aggregates.

The examples in this section use the following table:

```
CREATE TABLE table1 (
    a INT,
    b INT,
    c INT
);
```

Optimizing COUNT (DISTINCT) by Calculating Approximate Counts

HP Vertica provides the COUNT(DISTINCT) function to compute the exact number of distinct values in a data set. If projections are available that allow COUNT(DISTINCT) to execute using the GROUPBY PIPELINED algorithm, COUNT(DISTINCT) performs well. In some situations, however, using APPROXIMATE_COUNT_DISTINCT performs better than COUNT(DISTINCT).

A COUNT [Aggregate] operation performs well when:

- One of the sorted projections delivers an order that enables sorted aggregation to be performed.
- The number of distinct values is fairly small.

- Hashed aggregation is required to execute the query.

When an approximate value will suffice or the values need to be rolled up, consider using the APPROXIMATE_COUNT_DISTINCT* functions.

Note: The APPROXIMATE_COUNT_DISTINCT* functions cannot appear in the same query block as DISTINCT aggregates.

When to Use the Approximate Count Distinct Functions

Use APPROXIMATE_COUNT_DISTINCT as a direct replacement for COUNT (DISTINCT) when:

- You have a large data set and you do not require an exact count of distinct values.
- The performance of COUNT(DISTINCT) on a given data set is insufficient.
- You calculate several distinct counts in the same query.
- The plan for COUNT(DISTINCT) uses hashed aggregation.

Most of the time, APPROXIMATE_COUNT_DISTINCT executes faster than a comparable COUNT(DISTINCT) operation when hashed.

The expected value that APPROXIMATE_COUNT_DISTINCT returns is equal to COUNT (DISTINCT), with an error that is lognormally distributed with standard deviation *s*. You can control the standard deviation directly by setting the *error_tolerance*.

Use APPROXIMATE_COUNT_DISTINCT_SYNOPSIS and APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS together when:

- You have a large data set and you don't require an exact count of distinct values.
- The performance of COUNT(DISTINCT) on a given data set is insufficient.

  and

- You want to pre-compute the distinct counts and later combine them in different ways.

Pass APPROXIMATE_COUNT_DISTINCT_SYNOPSIS the data set and a normally distributed confidence interval. The function returns a materialized view of the data, called a *synopsis*;

Pass the synopsis to the APPROXIMATE_COUNT_DISTINCT_OF_SYNOPSIS function, which then performs an approximate count distinct on the synopsis.

If the Query Has a Single DISTINCT Aggregate

Vertica computes a `DISTINCT` aggregate by first removing all duplicate values of the aggregate's argument to find the distinct values. Then it computes the aggregate.

For example, you can rewrite the following query:

```
SELECT a, b, COUNT(DISTINCT c) AS dcnt FROM table1 GROUP BY a, b;
```

as:

```
SELECT a, b, COUNT(dcnt) FROM
  (SELECT a, b, c AS dcnt FROM table1 GROUP BY a, b, c)
GROUP BY a, b;
```

For fastest execution, apply the optimization techniques for GROUP BY queries.

If the Query Has Multiple DISTINCT Aggregates

If your query has multiple `DISTINCT` aggregates, there is no straightforward SQL rewrite that can compute them. The following query cannot easily be rewritten for improved performance:

```
SELECT a, COUNT(DISTINCT b), COUNT(DISTINCT c) AS dcnt FROM table1 GROUP BY a;
```

For a query with multiple `DISTINCT` aggregates, there is no projection design that can avoid using `GROUPBY HASH` and resegmenting the data. To improve performance of this query, make sure that it has large amounts of memory available. For more information about memory allocation for queries, see The Resource Manager.

Optimizing JOIN Queries

When you run a query that references more than one table, HP Vertica may need to do one or both of the following operations to join the tables together:

- Sort the data
- Resegment the data

The following sections provide recommendations for designing your projections and your queries to reduce query run time and improve the performance of queries that perform joins.

Hash Joins vs. Merge Joins

When processing a join, the HP Vertica optimizer has two algorithms to choose from:

- **Merge join**—If both inputs are pre-sorted on the join column, the optimizer chooses the faster merge join, which also uses less memory. Vertica can only perform merge joins on queries that have `INSERT` and `SELECT` operations.
- **Hash join**—Using the hash join algorithm, HP Vertica uses the smaller (inner) joined table to build an in-memory hash table on the join column. HP Vertica then scans the outer (larger) table and probes the hash table for matches. A hash join has no sort requirement, but it consumes more memory because a hash table is built with the values in the inner table.The cost of performing a hash join is low if the entire hash table can fit in memory, but the cost rises when the hash table is written to disk. The optimizer chooses a hash join when projections are not sorted on the join columns.

The optimizer automatically chooses the most appropriate algorithm given the query and projections in a system. You can facilitate a merge join by adding a projection that is sorted on the join key.

Optimizing for Merge Joins

To have HP Vertica perform a merge join, which is usually faster than a hash join, design projections where the join key is the first sorted column. HP Vertica also performs a merge join if the join key is second in the sort order, following the column used in the equality predicate. Otherwise, use a subquery to sort the table on the join key before performing the join.

The following projections are ordered by `join_key`, so the `SELECT` statement executes as a merge join:

```
CREATE TABLE first (
  data INT,
  join_key INT
  );
CREATE TABLE second (
  data INT,
  join_key INT
  );
CREATE PROJECTION first_p (data, join_key) AS
  SELECT data, join_key FROM first
  ORDER BY join_key;
CREATE PROJECTION second_p (data, join_key) AS
  SELECT data, join_key FROM second
  ORDER BY join_key;
SELECT first.data, second.data FROM first, second
  WHERE first.join_key = second.join_key;
```

You also get a merge join if your query has an equality predicate in your query and the join key immediately follows the column used in the equality predicate, as in the following example. After applying the predicate, the data is sorted by the `join_key` column:

```
CREATE TABLE first (
  data INT,
  join_key INT
  );
CREATE TABLE second (
  data INT,
  join_key INT
  );
CREATE PROJECTION first_p (data, join_key)
  AS SELECT data, join_key FROM first
  ORDER BY data, join_key;
CREATE PROJECTION second_p (data, join_key)
  AS SELECT data, join_key FROM second
  ORDER BY join_key;
SELECT first.data, second.data FROM first, second
  WHERE first.join_key = second.join_key AND first.data = 5;
```

Using Equality Predicates to Optimize Joins

Joins run faster if all the columns on the left side of the equality predicate come from one table and all the columns on the right side of the equality predicate come from another table. For example:

```
=> SELECT * FROM T JOIN X WHERE T.a + T.b = X.x1 - X.x2;
```

The following query requires much more work to compute:

```
=> SELECT * FROM T JOIN X WHERE T.a = X.x1 + T.b
```

Specifying INNER and OUTER Tables to Optimize Joins

To improve the performance of queries that perform joins, make sure that HP Vertica chooses the larger table as the outer (left hand) input by ensuring that any applicable constraints are defined.

Avoiding Resegmentation During Joins

To improve query performance when you join multiple tables, create projections that are identically segmented on the join keys. These are called *identically-segmented projections (ISPs)*. Identically-segmented projections allow the joins to occur locally on each node without any data movement across the network during query processing.

To determine if the projections are identically-segmented on the query join keys, create a query plan with `EXPLAIN`. If the query plan contains `RESEGMENT` or `BROADCAST`, the projections are not identically segmented.

The HP Vertica optimizer chooses a projection to supply rows for each table in a query. If two chosen projections to be joined are segmented, the optimizer uses their segmentation expressions and the join expressions in the query to determine if the rows are correctly placed to perform the join without any data movement.

Note: Executing queries that join identically-segmented projections is relevant for multi-node databases.

Join Conditions for Identically Segmented Projections (ISPs)

A projection `p` is segmented on join columns if all column references in `p`'s segmentation expression are a subset of the columns in the join expression.

The following conditions must be true for two segmented projections `p1` of table `t1` and `p2` of table `t2` to participate in a join of `t1` to `t2`:

- The join condition must have the following form:

  ```
  t1.j1 = t2.j1 AND t1.j2 = t2.j2 AND ... t1.jN = t2.jN
  ```

- The join columns must share the same base data type. For example:
  - If `t1.j1` is an INTEGER, `t2.j1` can be an INTEGER but it cannot be a FLOAT.
  - If `t1.j1` is a CHAR(10), `t2.j1` can be any CHAR or VARCHAR (for example, CHAR(10), VARCHAR(10), VARCHAR(20)), but `t2.j1` cannot be an INTEGER.
- If `p1` is segmented by an expression on columns {`t1.s1, t1.s2, ... t1.sN`}, each segmentation column `t1.sX` must be in the join column set {`t1.jX`}.
- If `p2` is segmented by an expression on columns {`t2.s1, t2.s2, ... t2.sN`}, each segmentation column `t2.sX` must be in the join column set {`t2.jX`}.
- The segmentation expressions of `p1` and `p2` must be structurally equivalent. For example:
  - If `p1` is `SEGMENTED BY hash(t1.x)` and `p2` is `SEGMENTED BY hash(t2.x)`, `p1` and `p2` are identically segmented.
  - If `p1` is `SEGMENTED BY hash(t1.x)` and `p2` is `SEGMENTED BY hash(t2.x + 1)`, `p1` and `p2` are not identically segmented.
- `p1` and `p2` must have the same segment count.
- The assignment of segments to nodes must match. For example, if `p1` and `p2` use an `OFFSET` clause, their offsets must match.
- If HP Vertica finds projections for `t1` and `t2` that are not identically segmented, the data is redistributed across the network during query run time, as necessary.

  Tip: If you are creating custom designs, try to use segmented projections for joins whenever possible. See the following section "Designing Identically Segmented Projections for K-Safety".

The following statements create two tables and specify ISP conditions:

```
CREATE TABLE t1 (id INT, x1 INT, y1 INT) SEGMENTED BY HASH(id) ALL NODES;
CREATE TABLE t2 (id INT, x2 INT, y2 INT) SEGMENTED BY HASH(id) ALL NODES;
```

Corresponding to this design, the following syntax shows ISP-supported join conditions:

```
SELECT * FROM t1 JOIN t2 ON t1.id = t2.id;                   -- ISP
SELECT * FROM t1 JOIN t2 ON t1.id = t2.id AND t1.x1 = t2.x2; -- ISP
SELECT * FROM t1 JOIN t2 ON t1.x1 = t2.x2;                   -- NOT ISP
SELECT * FROM t1 JOIN t2 ON t1.id = t2.x2;                   -- NOT ISP
```

Designing Identically Segmented Projections for K-Safety

For **K-safety**, if A and B are two identically segmented projections, their **buddy projections**, A_{buddy} and B_{buddy}, should also be identically segmented to each another.

The following syntax illustrates suboptimal buddy projection design because the projections are not identically segmented to each other because their OFFSET values differ:

CREATE PROJECTION t1_b1 (id, x1, y1)
AS SELECT * FROM t1
SEGMENTED BY HASH(id)
ALL NODES **OFFSET 1;**
	CREATE PROJECTION t2_b1 (id, x2, y2)
	AS SELECT * FROM t2
	SEGMENTED BY HASH(id)
	ALL NODES **OFFSET 2;**

The following syntax is another example of suboptimal buddy projection design. The projections are not identically segmented to each other because their segmentation expressions differ, so the projections do not qualify as buddies:

CREATE PROJECTION t1_b2 (id, x1, y1)
AS SELECT * FROM t1
SEGMENTED BY **HASH(id, x1)**
ALL NODES OFFSET 1;
	CREATE PROJECTION t2_b2 (id, x2, y2)
	AS SELECT * FROM t2
	SEGMENTED BY **HASH(id)**
	ALL NODES OFFSET 2;

Buddy projections can use different sort orders. For details, see Hash Segmentation in the SQL Reference Manual.

Notes

- HP Vertica recommends that you use **Database Designer** to create projections, which uses HASH and ALL NODES syntax.
- HP Vertica recommends that all tables use hash segmentation or be replicated.

See Also

- Partitioning and Segmenting Data
- CREATE PROJECTION

Optimizing ORDER BY Queries

You can improve the performance of queries that contain only `ORDER BY` clauses if the columns in a projection's `ORDER BY` clause are the same as the columns in the query.

Pre-Sorting Projections to Optimize ORDER BY Clauses

If you define the projection sort order in the `CREATE PROJECTION` statement, the HP Vertica query optimizer does not have to sort projection data before performing certain `ORDER BY` queries.

The following table, `sortopt`, contains the columns `a`, `b`, `c`, and `d`. Projection `sortopt_p` specifies to order on columns `a`, `b`, and `c`.

```
CREATE TABLE sortopt (
    a INT NOT NULL,
    b INT NOT NULL,
    c INT,
    d INT
);
CREATE PROJECTION sortopt_p (
   a_proj,
   b_proj,
   c_proj,
   d_proj )
AS SELECT * FROM sortopt
ORDER BY a,b,c
UNSEGMENTED ALL NODES;
INSERT INTO sortopt VALUES(5,2,13,84);
INSERT INTO sortopt VALUES(14,22,8,115);
INSERT INTO sortopt VALUES(79,9,401,33);
```

Based on this sort order, if a `SELECT * FROM sortopt` query contains one of the following `ORDER BY` clauses, the query does not have to resort the projection:

- `ORDER BY a`
- `ORDER BY a, b`
- `ORDER BY a, b, c`

For example, HP Vertica does not have to resort the projection in the following query because the sort order includes columns specified in the `CREATE PROJECTION..ORDER BY a, b, c` clause, which mirrors the query's `ORDER BY a, b, c` clause:

```
=> SELECT * FROM sortopt ORDER BY a, b, c;
 a  | b  |  c  |  d
----+----+-----+-----
  5 |  2 |  13 |  84
 14 | 22 |   8 | 115
 79 |  9 | 401 |  33
(3 rows)
```

If you include column `d` in the query, HP Vertica must re-sort the projection data because column `d` was not defined in the `CREATE PROJECTION..ORDER BY` clause. Therefore, the `ORDER BY d` query won't benefit from any sort optimization.

You cannot specify an ASC or DESC clause in the CREATE PROJECTION statement's ORDER BY clause. HP Vertica always uses an ascending sort order in physical storage, so if your query specifies descending order for any of its columns, the query still causes HP Vertica to re-sort the projection data. For example, the following query requires HP Vertica to sort the results:

```
=> SELECT * FROM sortopt ORDER BY a DESC, b, c;
 a  | b  |  c  |  d
----+----+-----+-----
 79 |  9 | 401 |  33
 14 | 22 |   8 | 115
  5 |  2 |  13 |  84
(3 rows)
```

See Also

- CREATE PROJECTION

Optimizing SQL-99 Analytic Functions

The following sections describe how to optimizing the SQL-99 analytic functions that HP Vertica supports.

Avoiding Single-Node Execution By Avoiding Empty OVER() Clauses

The `OVER()` clause does not require a windowing clause. If your query uses an analytic function like `SUM(x)` and you specify an empty `OVER()` clause, the analytic function is used as a reporting function, where the entire input is treated as a single partition; the aggregate returns the same aggregated value for each row of the result set. The query executes on a single node, potentially resulting in poor performance.

If you add a `PARTITION BY` clause to the `OVER()` clause, the query executes on multiple nodes, improving its performance.

NULL Placement By Analytic Functions

By default, projection column values are stored in ascending order, but the placement of NULLs depends on a column's data type.

The analytic `OVER(window_order_clause)` and the SQL `ORDER BY` clause have slightly different semantics:

OVER(ORDER BY ...)	(SQL) ORDER BY
The analytic window_order_clause sorts data, based on the results of the analytic function, as either ascending (`ASC`) or descending (`DESC`) and specifies where NULL values appear in the sorted result as either `NULLS FIRST` or `NULLS LAST`. The following is the analytics default sort order and NULL placement: • If you order `ASC`, the null placement is `NULLS LAST`. NULL values appear at the end of the sorted result. • If you order `DESC`, the null placement is `NULLS FIRST`. NULL values appear at the beginning of the sorted result.	The SQL `ORDER BY` clause specifies only ascending or descending order. In HP Vertica, however, default NULL placement depends on that column's data type: • NUMERIC, INTEGER, DATE, TIME, TIMESTAMP, and INTERVAL columns: `NULLS FIRST` (NULL values appear at the beginning of a sorted projection.) • FLOAT, STRING, and BOOLEAN columns: `NULLS LAST` (NULL values appear at the end of a sorted projection.)

If you do not care about NULL placement in queries that involve analytic computations, or if you know that columns contain no NULL values, specify `NULLS AUTO`—irrespective of data type. HP

Vertica chooses the placement that gives the fastest performance. Otherwise, specify NULLS FIRST or NULLS LAST.

You can carefully formulate queries so HP Vertica can avoid sorting the data and process the query quicker, as illustrated by the following example. HP Vertica sorts inputs from table t on column x, as specified in the OVER(ORDER BY) clause, and then evaluates RANK():

```
=> CREATE TABLE t (
    x FLOAT,
    y FLOAT );
=> CREATE PROJECTION t_p (x, y) AS SELECT * FROM t
   ORDER BY x, y UNSEGMENTED ALL NODES;
=> SELECT x, RANK() OVER (ORDER BY x) FROM t;
```

In the preceding SELECT statement, HP Vertica eliminates the ORDER BY clause and runs the query quickly because column x is a FLOAT data type. As a result, the projection sort order matches the analytic default ordering (ASC + NULLS LAST). HP Vertica can also avoid having to sort the data when the underlying projection is already sorted.

However, if column x is defined as INTEGER, HP Vertica must sort the data because the projection sort order for INTEGER data types (ASC + NULLS FIRST) does not match the default analytic ordering (ASC + NULLS LAST). To help HP Vertica eliminate the sort, specify the placement of NULLs to match the default ordering:

```
=> SELECT x, RANK() OVER (ORDER BY x NULLS FIRST) FROM t;
```

If column x is a STRING, the following query eliminates the sort:

```
=> SELECT x, RANK() OVER (ORDER BY x NULLS LAST) FROM t;
```

If you omit NULLS LAST in the preceding query, it eliminates the sort because ASC + NULLS LAST is the default sort specification for both the analytic ORDER BY clause and for string-related columns in HP Vertica.

If you do not care about NULL placement in queries that involve analytic computations, or if you know that columns contain no NULL values, specify NULLS AUTO. In the following query, HP Vertica chooses the placement that gives the fastest performance:

```
=> SELECT x, RANK() OVER (ORDER BY x NULLS AUTO) FROM t;
```

See Also

- Designing Tables to Minimize Run-Time Sorting of NULL Values in Analytic Functions
- Using SQL Analytics

Designing Tables to Minimize Run-Time Sorting of NULL Values in Analytic Functions

By carefully writing queries or creating your design (or both), you can help the HP Vertica query optimizer skip sorting all columns in a table when performing an analytic function, which can improve query performance.

To minimize HP Vertica's need to sort projections during query execution, redefine the `employee` table and specify that NULL values are not allowed in the sort fields:

```
DROP TABLE employee CASCADE;
```

```
CREATE TABLE employee
   (empno INT,
    deptno INT NOT NULL,
    sal INT NOT NULL);
CREATE PROJECTION employee_p AS
   SELECT * FROM employee
   ORDER BY deptno, sal;
```

```
INSERT INTO employee VALUES(101,10,50000);
INSERT INTO employee VALUES(103,10,43000);
INSERT INTO employee VALUES(104,10,45000);
INSERT INTO employee VALUES(105,20,97000);
INSERT INTO employee VALUES(108,20,33000);
INSERT INTO employee VALUES(109,20,51000);
```

```
=> SELECT * FROM employee;
 empno | deptno |  sal
-------+--------+-------
   101 |     10 | 50000
   103 |     10 | 43000
   104 |     10 | 45000
   105 |     20 | 97000
   108 |     20 | 33000
   109 |     20 | 51000
(6 rows)
=> SELECT deptno, sal, empno, RANK() OVER
     (PARTITION BY deptno ORDER BY sal)
   FROM employee;
 deptno |  sal  | empno | ?column?
--------+-------+-------+----------
     10 | 43000 |   103 |        1
     10 | 45000 |   104 |        2
     10 | 50000 |   101 |        3
     20 | 33000 |   108 |        1
     20 | 51000 |   109 |        2
     20 | 97000 |   105 |        3
(6 rows)
```

Tip: If you do not care about NULL placement in queries that involve analytic computations, or if you know that columns contain no NULL values, specify `NULLS AUTO` in your queries. HP Vertica attempts to choose the placement that gives the fastest performance. Otherwise, specify NULLS FIRST or NULLS LAST.

Optimizing LIMIT Queries with ROW_NUMBER Predicates

Queries that use the LIMIT Clause with ORDER BY or the SQL-99 analytic function ROW_NUMBER() return a specific subset of rows in the query result. HP Vertica processes these queries efficiently using *Top-K Optimization*, which is a database query ranking process. Top-K optimization avoids sorting (and potentially writing to disk) an entire data set to find a small number of rows. This can significantly improve query performance.

For example, in the following query, HP Vertica extracts only the three smallest rows from column x:

```
=> SELECT * FROM t1 ORDER BY x LIMIT 3;
```

If table t1 contains millions of rows, it is time consuming to sort all the x values. Instead, HP Vertica keeps track of the smallest three values in x.

Note: If you omit the ORDER BY clause, when using the LIMIT clause, the results can be **nondeterministic**.

Sort operations that precede a SQL analytics computation benefit from Top-K optimization if the query contains an OVER(ORDER BY) clause and a predicate on the ROW_NUMBER function, as in the following example:

```
=> SELECT x FROM     (SELECT *, ROW_NUMBER() OVER (ORDER BY x) AS row
     FROM t1) t2 WHERE row <= 3;
```

The preceding query has the same behavior as the following query, which uses a LIMIT clause:

```
=> SELECT ROW_NUMBER() OVER (ORDER BY x) AS RANK FROM t1 LIMIT 3;
```

You can use ROW_NUMBER() with the analytic window_partition_clause, something you cannot do if you use LIMIT:

```
=> SELECT x, y FROM
     (SELECT *, ROW_NUMBER() OVER (PARTITION BY x ORDER BY y)
     AS row FROM t1) t2 WHERE row <= 3;
```

Note: When the OVER() clause includes the window_partition_clause, Top-K optimization occurs only when the analytic sort matches the input's sort, for example, if the projection is sorted on columns x and y in table t1.

If you still want to improve the performance of your query, consider using the optimization techniques described in Optimizing ORDER BY Queries.

Optimizing INSERT-SELECT Operations

There are several ways to optimize an INSERT-SELECT query that has the following format:

```
INSERT /*+direct*/ INTO destination SELECT * FROM source;
```

Optimizing INSERT-SELECT Queries for Tables with Pre-Join Projections

If you have an `INSERT-SELECT` query where the `SELECT` clause includes a join, HP Vertica determines the order for the `SELECT` part using the rules defined in Hash Joins vs. Merge Joins. When inserting into a pre-join projection, a join must be performed during the `INSERT-SELECT` query. If the incoming data is not already sorted correctly for a merge join, add an `ORDER BY` clause that matches the sort order of the dimension table's projection to the `SELECT` clause to facilitate the merge join.

To determine whether your query is using a hash join or a merge join, run the EXPLAIN statement on the query.

The following example generates a hash join instead of a merge join for a FK-PK validation when inserting into a pre-join projection:

```
-- Would like to use a MERGE JOIN for FK-PK validation, but getting a HASH JOIN
DROP TABLE f1 CASCADE;
DROP TABLE d1 CASCADE;
DROP TABLE f1_staging CASCADE;
CREATE TABLE f1(a varchar(10) NOT NULL, b varchar(10) NOT NULL);
CREATE TABLE d1(a varchar(10) NOT NULL, b varchar(10) NOT NULL);
CREATE TABLE f1_staging(a varchar(10) NOT NULL, b varchar(10) NOT NULL);
ALTER TABLE d1 ADD CONSTRAINT d1_pk PRIMARY KEY (a, b);
ALTER TABLE f1 ADD CONSTRAINT f1_fk FOREIGN KEY (a, b) references d1 (a, b);
CREATE PROJECTION f1_super(a, b) AS SELECT * FROM f1 ORDER BY a, b;
CREATE PROJECTION d1_super(a, b) AS SELECT * FROM d1 ORDER BY a, b;
CREATE PROJECTION f1_staging_super(a, b) AS SELECT * FROM f1_staging ORDER BY a, b;
CREATE PROJECTION prejoin(f1_a, f1_b, d1_a, d1_b)
AS SELECT f1.a, f1.b, d1.a, d1.b
FROM f1 join d1 on f1.a=d1.a and f1.b=d1.b
ORDER BY d1.a, d1.b;
COPY d1 FROM stdin delimiter ' ' direct;
one one
two two
\.
COPY f1 FROM stdin delimiter ' ' direct;
one one
two two
\.
INSERT INTO f1_staging values('one', 'one');
-- Performing HASH JOIN instead of MERGE JOIN
INSERT INTO f1
SELECT f1s.a, f1s.b
FROM f1_staging f1s join d1
```

```
on f1s.a=d1.a and f1s.b=d1.b;
-- Adding an ORDER BY clause to force a MERGE JOIN
INSERT INTO f1
SELECT f1s.a, f1s.b
FROM f1_staging f1s join d1
on f1s.a=d1.a and f1s.b=d1.b ORDER BY f1s.a, f1s.b;
```

Optimizing INSERT-SELECT Queries By Matching Sort Orders

When performing `INSERT`-`SELECT` operations, to avoid the sort phase of the `INSERT`, make sure that the sort order for the `SELECT` query matches the projection sort order of the target table.

For example, on a single-node database:

```
=> CREATE TABLE source (col1 INT, col2 INT, col3 INT);
=> CREATE PROJECTION source_p (col1, col2, col3)
     AS SELECT col1, col2, col3 FROM source
     ORDER BY col1, col2, col3
     SEGMENTED BY HASH(col3)
     ALL NODES;
=> CREATE TABLE destination (col1 INT, col2 INT, col3 INT);
=> CREATE PROJECTION destination_p (col1, col2, col3)
     AS SELECT col1, col2, col3 FROM destination
     ORDER BY col1, col2, col3
     SEGMENTED BY HASH(col3)
     ALL NODES;
```

The following `INSERT` does not require a sort because the query result has the column order of the projection:

```
=> INSERT /*+direct*/ INTO destination SELECT * FROM source;
```

The following `INSERT` requires a sort because the order of the columns in the `SELECT` statement does not match the projection order:

```
=> INSERT /*+direct*/ INTO destination SELECT col1, col3, col2 FROM source;
```

The following `INSERT` does not require a sort. The order of the columns doesn't match, but the explicit ORDER BY causes the output to be sorted by `c1`, `c3`, `c2` in HP Vertica:

```
=> INSERT /*+direct*/ INTO destination SELECT col1, col3, col2 FROM source
      GROUP BY col1, col3, col2
      ORDER BY col1, col2, col3 ;
```

Avoiding Resegmentation of INSERT-SELECT Queries

When performing an `INSERT-SELECT` operation from a segmented source table to a segmented destination table, segment both projections on the same column to avoid resegmenting the data, as in the following example:

```
CREATE TABLE source (col1 INT, col2 INT, col3 INT);
CREATE PROJECTION source_p (col1, col2, col3) AS
   SELECT col1, col2, col3 FROM source
   SEGMENTED BY HASH(col3) ALL NODES;
CREATE TABLE destination (col1 INT, col2 INT, col3 INT);
CREATE PROJECTION destination_p (col1, col2, col3) AS
    SELECT col1, col2, col3 FROM destination
    SEGMENTED BY HASH(col3) ALL NODES;
INSERT /*+direct*/ INTO destination SELECT * FROM source;
```

Optimizing DELETE and UPDATE Queries

HP Vertica is optimized for query-intensive workloads, so `DELETE` and `UPDATE` queries might not achieve the same level of performance as other queries. A `DELETE` and `UPDATE` operation has to update all projections, so the operation is as slow as the slowest projection. For additional information, see Using INSERT, UPDATE, and DELETE.

The topics that follow discuss best practices for optimizing `DELETE` and `UPDATE` queries in HP Vertica.

Performance Considerations for DELETE and UPDATE Queries

To improve the performance of your `DELETE` and `UPDATE` queries, consider the following issues:

- **Query performance after large deletes**—A large number of (unpurged) deleted rows can negatively affect query performance.

  To eliminate rows that have been deleted from the result, a query must do extra processing. If 10% or more of the total rows in a table have been deleted, the performance of a query on the table degrades. However, your experience may vary depending on the size of the table, the table definition, and the query. If a table has a large number of deleted rows, consider purging those rows to improve performance. For more information on purging, see Purging Deleted Data.

- **Recovery performance**—Recovery is the action required for a cluster to restore K-safety after a crash. Large numbers of deleted records can degrade the performance of a recovery. To improve recovery performance, purge the deleted rows. For more information on purging, see Purging Deleted Data.

- **Concurrency**—`DELETE` and `UPDATE` take exclusive locks on the table. Only one `DELETE` or `UPDATE` transaction on a table can be in progress at a time and only when no loads (or `INSERT`s) are in progress. `DELETE`s and `UPDATE`s on different tables can be run concurrently.

- **Pre-join projections**—Avoid pre-joining dimension tables that are frequently updated. `DELETE` and `UPDATE` operations on pre-join projections cascade to the fact table, causing large `DELETE` or `UPDATE` operations.

For detailed tips about improving `DELETE` and `UPDATE` performance, see Optimizing DELETEs and UPDATEs for Performance.

Caution: HP Vertica does not remove deleted data immediately but keeps it as history for the purposes of **historical query.** A large amount of history can result in slower query performance. For information about how to configure the appropriate amount of history to retain, see Purging Deleted Data.

Optimizing DELETEs and UPDATEs for Performance

The process of optimizing DELETE and UPDATE queries is the same for both operations. Some simple steps can increase the query performance by tens to hundreds of times. The following sections describe several ways to improve projection design and improve DELETE and UPDATE queries to significantly increase DELETE and UPDATE performance.

Note: For large bulk deletion, HP Vertica recommends using Partitioned Tables where possible because they provide the best DELETE performance and improve query performance.

Projection Column Requirements for Optimized Deletes

When all columns required by the DELETE or UPDATE predicate are present in a projection, the projection is optimized for DELETEs and UPDATEs. DELETE and UPDATE operations on such projections are significantly faster than on non-optimized projections. Both simple and pre-join projections can be optimized.

For example, consider the following table and projections:

```
CREATE TABLE t (a INTEGER, b INTEGER, c INTEGER);
CREATE PROJECTION p1 (a, b, c) AS SELECT * FROM t ORDER BY a;
CREATE PROJECTION p2 (a, c) AS SELECT a, c FROM t ORDER BY c, a;
```

In the following query, both p1 and p2 are eligible for DELETE and UPDATE optimization because column a is available:

```
DELETE from t WHERE a = 1;
```

In the following example, only projection p1 is eligible for DELETE and UPDATE optimization because the b column is not available in p2:

```
DELETE from t WHERE b = 1;
```

Optimized Deletes in Subqueries

To be eligible for DELETE optimization, all target table columns referenced in a DELETE or UPDATE statement's WHERE clause must be in the projection definition.

For example, the following simple schema has two tables and three projections:

```
CREATE TABLE tb1 (a INT, b INT, c INT, d INT);
CREATE TABLE tb2 (g INT, h INT, i INT, j INT);
```

The first projection references all columns in tb1 and sorts on column a:

```
CREATE PROJECTION tb1_p AS SELECT a, b, c, d FROM tb1 ORDER BY a;
```

The buddy projection references and sorts on column `a` in `tb1`:

```
CREATE PROJECTION tb1_p_2 AS SELECT a FROM tb1 ORDER BY a;
```

This projection references all columns in `tb2` and sorts on column `i`:

```
CREATE PROJECTION tb2_p AS SELECT g, h, i, j FROM tb2 ORDER BY i;
```

Consider the following DML statement, which references `tb1.a` in its `WHERE` clause. Since both projections on `tb1` contain column `a`, both are eligible for the optimized `DELETE`:

```
DELETE FROM tb1 WHERE tb1.a IN (SELECT tb2.i FROM tb2);
```

Restrictions

Optimized `DELETE`s are not supported under the following conditions:

- With pre-join projections on nodes that are down
- With replicated and pre-join projections if subqueries reference the target table. For example, the following syntax is not supported:

  ```
  DELETE FROM tb1 WHERE tb1.a IN (SELECT e FROM tb2, tb2 WHERE tb2.e = tb1.e);
  ```

- With subqueries that do not return multiple rows. For example, the following syntax is not supported:

  ```
  DELETE FROM tb1 WHERE tb1.a = (SELECT k from tb2);
  ```

Projection Sort Order for Optimizing Deletes

Design your projections so that frequently-used `DELETE` or `UPDATE` predicate columns appear in the sort order of all projections for large `DELETE`s and `UPDATE`s.

For example, suppose most of the DELETE queries you perform on a projection look like the following:

```
DELETE from t where time_key < '1-1-2007'
```

To optimize the DELETEs, make time_key appear in the ORDER BY clause of all your projections. This schema design results in better performance of the DELETE operation.

In addition, add additional sort columns to the sort order such that each combination of the sort key values uniquely identifies a row or a small set of rows. For more information, see Choosing Sort Order: Best Practices. To analyze projections for sort order issues, use the EVALUATE_DELETE_PERFORMANCE function.

Using External Procedures

An external procedure is a procedure external to HP Vertica that you create, maintain, and store on the server. External procedures are simply executable files such as shell scripts, compiled code, code interpreters, and so on.

Implementing External Procedures

To implement an external procedure:

1. Create an external procedure executable file.

   See Requirements for External Procedures.

2. Enable the UID attribute for the file and allow read and execute permission for the group (if the owner is not the database administrator). For example:

   ```
   chmod 4777 helloplanet.sh
   ```

3. Install the external procedure executable file.

4. Create the external procedure in HP Vertica.

Once a procedure is created in HP Vertica, you can execute or drop it, but you cannot alter it.

Requirements for External Procedures

External procedures have requirements regarding their attributes, where you store them, and how you handle their output. You should also be cognizant of their resource usage.

Procedure File Attributes

A procedure file must be owned by the database administrator (OS account) or by a user in the same group as the administrator. The procedure file owner cannot be root and must have the set UID attribute enabled and allow read and execute permission for the group if the owner is not the database administrator.

Note: The file should end with *exit 0*, and exit 0 must reside on its own line. This naming convention instructs HP Vertica to return *0* when the script succeeds.

Handling Procedure Output

HP Vertica does not provide a facility for handling procedure output. Therefore, you must make your own arrangements for handling procedure output, which should include writing error, logging, and program information directly to files that you manage.

Handling Resource Usage

The HP Vertica resource manager is unaware of resources used by external procedures. Additionally, HP Vertica is intended to be the only major process running on your system. If your external procedure is resource intensive, it could affect the performance and stability of HP Vertica.

Consider the types of external procedures you create and when you run them. For example, you might run a resource-intensive procedure during off hours.

Sample Procedure File

```
#!/bin/bash
echo "hello planet argument: $1" >> /tmp/myprocedure.log
exit 0
```

Installing External Procedure Executable Files

To install an external procedure, use the Administration Tools from either the graphical user interface or the command line.

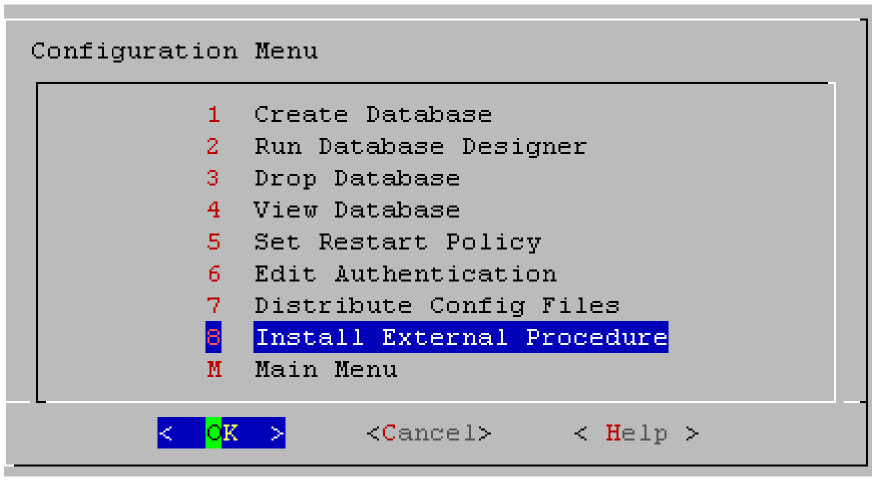
Graphical User Interface

a. Run the **Administration Tools**.

```
$ /opt/vertica/bin/adminTools
```

b. On the AdminTools **Main Menu**, click **Configuration Menu**, and then click **OK**.

c. On the **Configuration Menu**, click **Install External Procedure** and then click **OK**.



```
Configuration Menu

    1  Create Database
    2  Run Database Designer
    3  Drop Database
    4  View Database
    5  Set Restart Policy
    6  Edit Authentication
    7  Distribute Config Files
    8  Install External Procedure
    M  Main Menu

 <  OK  >     <Cancel>     < Help >
```

d. Select the database on which you want to install the external procedure.

e. Either select the file to install or manually type the complete file path, and then click **OK**.

f. If you are not the superuser, you are prompted to enter your password and click **OK**.

The Administration Tools automatically create the `<database_catalog_path>/procedures` directory on each node in the database and installs the external procedure in these directories for you.

g. Click **OK** in the dialog that indicates that the installation was successful.

Command Line

If you use the command line, be sure to specify the full path to the procedure file and the password of the Linux user who owns the procedure file;

For example:

```
$ admintools -t install_procedure -d vmartdb -f /scratch/helloworld.sh -p ownerpassword
Installing external procedure...
External procedure installed
```

Once you have installed an external procedure, you need to make HP Vertica aware of it. To do so, use the `CREATE PROCEDURE` statement, but review Creating External Procedures first.

Creating External Procedures

Once you have installed an external procedure, you need to make HP Vertica aware of it. To do so, use the CREATE PROCEDURE statement.

By default, only a superuser can create and execute a procedure. However, a superuser can grant the right to execute a stored procedure to a user on the operating system. (See GRANT (Procedure).)

Once created, a procedure is listed in the `V_CATALOG.USER_PROCEDURES` system table. Users can see only those procedures that they have been granted the privilege to execute.

Example

This example creates a procedure named `helloplanet` for the `helloplanet.sh` external procedure file. This file accepts one `VARCHAR` argument. The sample code is provided in Requirements for External Procedures.

```
=> CREATE PROCEDURE helloplanet(arg1 VARCHAR) AS 'helloplanet.sh' LANGUAGE 'external'
   USER 'release';
```

This example creates a procedure named `proctest` for the `copy_vertica_database.sh` script. This script copies a database from one cluster to another, and it is included in the server RPM located in the `/opt/vertica/scripts` directory.

```
=> CREATE PROCEDURE proctest(shosts VARCHAR, thosts VARCHAR, dbdir VARCHAR)
   AS 'copy_vertica_database.sh' LANGUAGE 'external' USER 'release';
```

See Also

- CREATE PROCEDURE
- GRANT (Procedure)

Executing External Procedures

Once you define a procedure through the CREATE PROCEDURE statement, you can use it as a meta command through a simple `SELECT` statement. HP Vertica does not support using procedures in more complex statements or in expressions.

The following example runs a procedure named `helloplanet`:

```
=> SELECT helloplanet('earthlings');
 helloplanet
-------------
           0
(1 row)
```

The following example runs a procedure named `proctest`. This procedure references the `copy_vertica_database.sh` script that copies a database from one cluster to another. It is installed by the server RPM in the `/opt/vertica/scripts` directory.

```
=> SELECT proctest(
    '-s qa01',
    '-t rbench1',
    '-D /scratch_b/qa/PROC_TEST' );
```

Note: External procedures have no direct access to database data. If available, use ODBC or JDBC for this purpose.

Procedures are executed on the initiating node. HP Vertica runs the procedure by forking and executing the program. Each procedure argument is passed to the executable file as a string. The parent fork process waits until the child process ends.

To stop execution, cancel the process by sending a cancel command (for example, CTRL+C) through the client. If the procedure program exits with an error, an error message with the exit status is returned.

Permissions

To execute an external procedure, the user needs:

- EXECUTE privilege on procedure
- USAGE privilege on schema that contains the procedure

See Also

- CREATE PROCEDURE
- External Procedure Privileges

Dropping External Procedures

Only a superuser can drop an external procedure. To drop the definition for an external procedure from HP Vertica, use the DROP PROCEDURE statement. Only the reference to the procedure is removed. The external file remains in the `<database_catalog_path>/procedures` directory on each node in the database.

Note: The definition HP Vertica uses for a procedure cannot be altered; it can only be dropped.

Example

```
=> DROP PROCEDURE helloplanet(arg1 varchar);
```

See Also

- DROP PROCEDURE

Using User-Defined SQL Functions

User-Defined SQL Functions let you define and store commonly-used SQL expressions as a function. User-Defined SQL Functions are useful for executing complex queries and combining HP Vertica built-in functions. You simply call the function name you assigned in your query.

A User-Defined SQL Function can be used anywhere in a query where an ordinary SQL expression can be used, except in the table partition clause or the projection segmentation clause.

For syntax and parameters for the commands and system table discussed in this section, see the following topics in the SQL Reference Manual:

- CREATE FUNCTION
- ALTER FUNCTION
- DROP FUNCTION
- GRANT (Function)
- REVOKE (Function)
- V_CATALOG.USER_FUNCTIONS

Creating User-Defined SQL Functions

A user-defined SQL function can be used anywhere in a query where an ordinary SQL expression can be used, except in the table partition clause or the projection segmentation clause.

To create a SQL function, the user must have CREATE privileges on the schema. To use a SQL function, the user must have USAGE privileges on the schema and EXECUTE privileges on the defined function.

This following statement creates a SQL function called `myzeroifnull` that accepts an `INTEGER` argument and returns an `INTEGER` result.

```
=> CREATE FUNCTION myzeroifnull(x INT) RETURN INT
   AS BEGIN
     RETURN (CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END);
   END;
```

You can use the new SQL function (`myzeroifnull`) anywhere you use an ordinary SQL expression. For example, create a simple table:

```
=> CREATE TABLE tabwnulls(col1 INT);
=> INSERT INTO tabwnulls VALUES(1);
=> INSERT INTO tabwnulls VALUES(NULL);
=> INSERT INTO tabwnulls VALUES(0);
=> SELECT * FROM tabwnulls;
```

```
 a
---
 1
 0
(3 rows)
```

Use the `myzeroifnull` function in a `SELECT` statement, where the function calls `col1` from table tabwnulls:

```
=> SELECT myzeroifnull(col1) FROM tabwnulls;
 myzeroifnull
--------------
          1
          0
          0
(3 rows)
```

Use the `myzeroifnull` function in the `GROUP BY` clause:

```
=> SELECT COUNT(*) FROM tabwnulls GROUP BY myzeroifnull(col1);
 count
-------
     2
     1
(2 rows)
```

If you want to change a user-defined SQL function's body, use the `CREATE OR REPLACE` syntax. The following command modifies the CASE expression:

```
=> CREATE OR REPLACE FUNCTION myzeroifnull(x INT) RETURN INT
   AS BEGIN
     RETURN (CASE WHEN (x IS NULL) THEN 0 ELSE x END);
   END;
```

To see how this information is stored in the HP Vertica catalog, see Viewing Information About SQL Functions in this guide.

See Also

- CREATE FUNCTION (SQL Functions)
- USER_FUNCTIONS

Altering and Dropping User-Defined SQL Functions

HP Vertica allows multiple functions to share the same name with different argument types. Therefore, if you try to alter or drop a SQL function without specifying the argument data type, the system returns an error message to prevent you from dropping the wrong function:

```
=> DROP FUNCTION myzeroifnull();
ROLLBACK:  Function with specified name and parameters does not exist: myzeroifnull
```

Note: Only a superuser or owner can alter or drop a SQL Function.

Altering a User-Defined SQL Function

The ALTER FUNCTION command lets you assign a new name to a user-defined function, as well as move it to a different schema.

In the previous topic, you created a SQL function called `myzeroifnull`. The following command renames the `myzeroifnull` function to `zerowhennull`:

```
=> ALTER FUNCTION myzeroifnull(x INT) RENAME TO zerowhennull;
ALTER FUNCTION
```

This next command moves the renamed function into a new schema called `macros`:

```
=> ALTER FUNCTION zerowhennull(x INT) SET SCHEMA macros;
ALTER FUNCTION
```

Dropping a SQL Function

The DROP FUNCTION command drops a SQL function from the HP Vertica catalog.

Like with ALTER FUNCTION, you must specify the argument data type or the system returns the following error message:

```
=> DROP FUNCTION zerowhennull();
ROLLBACK:  Function with specified name and parameters does not exist: zerowhennull
```

Specify the argument type:

```
=> DROP FUNCTION macros.zerowhennull(x INT);
DROP FUNCTION
```

HP Vertica does not check for dependencies, so if you drop a SQL function where other objects reference it (such as views or other SQL Functions), HP Vertica returns an error when those objects are used, not when the function is dropped.

Tip: To view a list of all user-defined SQL functions on which you have EXECUTE privileges, (which also returns their argument types), query the V_CATALOG.USER_FUNCTIONS system table.

See Also

- ALTER FUNCTION
- DROP FUNCTION

Managing Access to SQL Functions

Before a user can execute a user-defined SQL function, he or she must have USAGE privileges on the schema and EXECUTE privileges on the defined function. Only the superuser or owner can grant/revoke EXECUTE usage on a function.

To grant EXECUTE privileges to user Fred on the `myzeroifnull` function:

```
=> GRANT EXECUTE ON FUNCTION myzeroifnull (x INT) TO Fred;
```

To revoke EXECUTE privileges from user Fred on the `myzeroifnull` function:

```
=> REVOKE EXECUTE ON FUNCTION myzeroifnull (x INT) FROM Fred;
```

See Also

- GRANT (Function)
- REVOKE (Function)

Viewing Information About User-Defined SQL Functions

You can access information about any User-Defined SQL Functions on which you have EXECUTE privileges. This information is available in the system table V_CATALOG.USER_FUNCTIONS and from the vsql meta-command `\df`.

To view all of the User-Defined SQL Functions on which you have EXECUTE privileges, query the USER_FUNCTIONS table:

```
=> SELECT * FROM USER_FUNCTIONS;
-[ RECORD 1 ]----------+---------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
function_return_type   | Integer
function_argument_type | x Integer
function_definition    | RETURN CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END
volatility             | immutable
is_strict              | f
```

If you want to change a User-Defined SQL Function's body, use the CREATE OR REPLACE syntax. The following command modifies the CASE expression:

```
=> CREATE OR REPLACE FUNCTION myzeroifnull(x INT) RETURN INT
   AS BEGIN
     RETURN (CASE WHEN (x IS NULL) THEN 0 ELSE x END);
   END;
```

Now when you query the USER_FUNCTIONS table, you can see the changes in the `function_definition` column:

```
=> SELECT * FROM USER_FUNCTIONS;
-[ RECORD 1 ]----------+-------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
function_return_type   | Integer
function_argument_type | x Integer
function_definition    | RETURN CASE WHEN (x IS NULL) THEN 0 ELSE x END
volatility             | immutable
is_strict              | f
```

If you use CREATE OR REPLACE syntax to change only the argument name or argument type (or both), the system maintains both versions of the function. For example, the following command tells the function to accept and return a numeric data type instead of an integer for the `myzeroifnull` function:

```
=> CREATE OR REPLACE FUNCTION myzeroifnull(z NUMERIC) RETURN NUMERIC
   AS BEGIN
     RETURN (CASE WHEN (z IS NULL) THEN 0 ELSE z END);
   END;
```

Now query the USER_FUNCTIONS table, and you can see the second instance of `myzeroifnull` in Record 2, as well as the changes in the `function_return_type`, `function_argument_type`, and `function_definition` columns.

Note: Record 1 still holds the original definition for the `myzeroifnull` function:

```
=> SELECT * FROM USER_FUNCTIONS;
-[ RECORD 1 ]----------+----------------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
function_return_type   | Integer
function_argument_type | x Integer
function_definition    | RETURN CASE WHEN (x IS NULL) THEN 0 ELSE x END
volatility             | immutable
is_strict              | f
-[ RECORD 2 ]----------+----------------------------------------------------------
schema_name            | public
function_name          | myzeroifnull
function_return_type   | Numeric
function_argument_type | z Numeric
```

```
function_definition    | RETURN (CASE WHEN (z IS NULL) THEN (0) ELSE z END)::numeric
volatility             | immutable
is_strict              | f
```

Because HP Vertica allows functions to share the same name with different argument types, you must specify the argument type when you alter or drop a function. If you do not, the system returns an error message:

```
=> DROP FUNCTION myzeroifnull();
ROLLBACK:  Function with specified name and parameters does not exist: myzeroifnull
```

See Also

- USER_FUNCTIONS

Migrating Built-In SQL Functions

If you have built-in SQL functions from another RDBMS that do not map to an HP Vertica-supported function, you can migrate them into your HP Vertica database by using a user-defined SQL function.

The example scripts below show how to create user-defined functions for the following DB2 built-in functions:

- `UCASE()`
- `LCASE()`
- `LOCATE()`
- `POSSTR()`

UCASE()

This script creates a user-defined SQL function for the `UCASE()` function:

```
=> CREATE OR REPLACE FUNCTION UCASE (x VARCHAR)
   RETURN VARCHAR
   AS BEGIN
   RETURN UPPER(x);
   END;
```

LCASE()

This script creates a user-defined SQL function for the `LCASE()` function:

```
=> CREATE OR REPLACE FUNCTION LCASE (x VARCHAR)
```

```
   RETURN VARCHAR
   AS BEGIN
   RETURN LOWER(x);
   END;
```

LOCATE()

This script creates a user-defined SQL function for the `LOCATE()` function:

```
=> CREATE OR REPLACE FUNCTION LOCATE(a VARCHAR, b VARCHAR)
   RETURN INT
   AS BEGIN
   RETURN POSITION(a IN b);
   END;
```

POSSTR()

This script creates a user-defined SQL function for the `POSSTR()` function:

```
=> CREATE OR REPLACE FUNCTION POSSTR(a VARCHAR, b VARCHAR)
   RETURN INT
   AS BEGIN
   RETURN POSITION(b IN a);
   END;
```

Developing and Using User Defined Extensions

Usr Defined Extensions (abbreviated as UDx) are extensions to HP Vertica developed using the APIs in the HP Vertica Software Development Kits (SDKs). UDxs are broken into several categories:

- User Defined Functions (UDFs) which are used in SQL statements similarly to HP Vertica's own functions. There are several different types of UDFs, each of which is designed for a different data processing task.

- User Defined Loads (UDLs) which define custom data load modules.

HP Vertica supports developing UDxs in three languages:

- C++

- Java

- R

Not all types of extensions and functions are supported by each language. All of the supported languages allow developing UDFs, but some languages do not support all UDF types. You can only develop UDLs in C++.

This chapter describes how to develop and use UDxs.

How UDxs Work

User Defined Extensions (UDxs) are contained in libraries. Multiple extensions can be defined in a library, and multiple libraries can be loaded by HP Vertica. You load a library by:

1. Copying the library file to a location on the **initiator node**.

2. Connecting to the initiator node using vsql.

3. Using the CREATE LIBRARY statement, passing it the path where you saved the library file.

The initiator node takes care of distributing the library file to the rest of the nodes in the cluster.

Once the library is loaded, you define individual User Defined Functions or User Defined Loads using SQL statements such as CREATE FUNCTION and CREATE SOURCE. These statement assigns SQL function names to the extension classes in the library. From then on, you are able to use your extension within your SQL statements. Whenever you call a UDx, HP Vertica creates an instance of the UDx class on each node in the cluster and uses .

The CREATE FUNCTION statement adds the UDF to the database catalog. They remain available after a database restart. The database superuser can grant access privileges to the UDFs for users. See GRANT (User Defined Extension) in the SQL Reference Manual for details.

Fenced Mode

User Defined Extensions (UDxs) written in the C++ programming language have the option of running in unfenced mode, which means running directly within the HP Vertica process. Since they run within HP Vertica, unfenced UDxs have little overhead, and can perform almost as fast as HP Vertica's own built-in functions. However, since they run within HP Vertica directly, any bugs in their code (memory leaks, for example) can destabilize the main HP Vertica process that can bring one or more database nodes down.

You can instead opt to run most C++ UDxs in fenced mode, which runs the UDxs code outside of the main HP Vertica process in a separate zygote process. UDx code that crashes while running in fenced mode does not impact the core HP Vertica process. There is a small performance impact when running UDx code in fenced mode. On average, using fenced mode adds about 10% more time to execution compared to unfenced mode.

Fenced mode is currently available for all C++ UDx's with the exception of User Defined Aggregates and User Defined Load. All UDxs developed in the R and Java programming languages must run in fenced mode, since the R and Java runtimes cannot be directly run within the HP Vertica process.

Using fenced mode does not affect the development of your UDx. Fenced mode is enabled by default for UDx's that support fenced mode. The CREATE FUNCTION command can optionally be issued with the NOT FENCED modifier to disable fenced mode for the function. Also, you can enable or disable fenced mode on any fenced mode-supported C++ UDx by using the ALTER FUNCTION command.

About the Zygote Process

The HP Vertica zygote process starts when HP Vertica starts. Each node has a single zygote process. Side processes are created "on demand". The zygote listens for requests and spawns a UDx side session that runs the UDx in fenced mode when a UDx is called by the user.

About Fenced Mode Logging:

UDx code that runs in fenced mode is logged in the `UDxZygote.log` and is stored in the `UDxLogs` directory in the catalog directory of HP Vertica. Log entries for the side process are denoted by the UDx language (for example, C++), node, and zygote process ID, and the UdxSideProcess ID.

For example, for the following processes...

```
dbadmin => select * from UDX_FENCED_PROCESSES;
    node_name     |   process_type   |            session_id            |  pid  | port  |
status
------------------+------------------+----------------------------------+-------+-------+
--------
 v_vmart_node0001 | UDxZygoteProcess |                                  | 27468 | 51900 |
UP
 v_vmart_node0001 | UDxSideProcess   | localhost.localdoma-27465:0x800b |  5677 | 44123 |
UP
```

... the corresponding log file displays:

```
2012-05-16 11:24:43.990 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 UDx s
ide process started
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 Finished settin
g up signal handlers.
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 My port: 44123
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 My address: 0.0
.0.0
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 Vertica port: 5
1900
 11:24:43.996 [C++-localhost.localdoma-27465:0x800b-5677]  0x2b3ff17e7fd0 Vertica addres
s: 127.0.0.1
 11:25:19.749 [C++-localhost.localdoma-27465:0x800b-5677]  0x41837940 Setting memory reso
urce limit to -1
 11:30:11.523 [C++-localhost.localdoma-27465:0x800b-5677]  0x41837940 Exiting UDx side pr
ocess
```

The last line indicates that the side process was killed. In this case it was killed when the user session (vsql) closed.

About Fenced Mode Configuration Parameters

Fenced mode supports two configuration parameters:

- FencedUDxMemoryLimitMB - The maximum memory size, in MB, to use for Fenced Mode processes. The default is -1 (no limit). The side process is killed if this limit is exceeded.
- ForceUDxFencedMode - When set to 1, force all UDx's that support fenced mode to run in fenced mode even if their definition specified NOT FENCED. The default is 0 (disabled).

See Also

- CREATE LIBRARY
- CREATE FUNCTION
- CREATE TRANSFORM FUNCTION
- CREATE ANALYTIC FUNCTION
- ALTER FUNCTION
- UDX_FENCED_PROCESSES

Developing User Defined Functions (UDFs)

User-Defined Functions (UDFs) are functions contained in external shared libraries that you develop in C++, R, or Java, and load into HP Vertica using the CREATE LIBRARY statement. They are best suited for analytic operations that are difficult to perform in SQL, and need to be performed frequently enough that their speed is a major concern.

UDFs primary strengths are:

- They can be used much more flexibly than external procedures within SQL statements. Generally, they can be used anywhere an internal function can be used.
- They take full advantage of HP Vertica's distributed computing features. Functions are executed in parallel on each node in the cluster.
- HP Vertica handles the distribution of the UDF library to the individual nodes. You only need to copy the library to the initiator node.
- All of the complicated aspects of developing a distributed piece of analytic code are handled for you by HP Vertica. Your main programming task is to read in data, process it, and then write it out using the HP Vertica SDK APIs.

There are a few things to keep in mind about developing UDFs:

- If you choose to run a C++ UDF in unfenced mode (directly within the HP Vertica process), any bugs in its code can cause database instability. You should thoroughly test any UDF you intend to run in unfenced mode before deploying them in a live environment. You should consider whether the performance boost of running a C++ UDF unfenced is worth the potential database instability that a buggy UDF can cause.
- UDFs can be developed in a three programming languages: C++, Java, and R. If you want to use another programming language to extend HP Vertica, you could develop an **external procedure**. However, external procedures are less efficient than UDFs.
- Since UDFs run on the HP Vertica cluster, they can take processor time and memory away from the database processes. UDFs that consume large amounts of computing resources can negatively impact database performance.

This section explains how to create and use user-defined functions (UDFs).

Types of UDFs

There are five different types of user defined functions:

- User Defined Scalar Functions (UDSFs) take in a single row of data and return a single value. These functions can be used anywhere a native HP Vertica function can be used, except CREATE TABLE BY PARTITION and SEGMENTED BY expressions.

- User Defined Transform Functions (UDTFs) operate on table segments and return zero or more rows of data. The data they return can be an entirely new table, unrelated to the schema of the input table, including having its own ordering and segmentation expressions. They can only be used in the SELECT list of a query. For details see Using User Defined Transforms.

- User Defined Aggregate Functions (UDAF) allow you to create custom aggregate functions specific to your needs. They read one column of data, and return one output column.

- User Defined Analytic Functions (UDAnF) are similar to UDSFs, in that they read a row of data and return a single row. However, the function can read input rows independently of outputting rows, so that the output values can be calculated over several input rows.

  The User Defined Load (UDL) feature allows you to create custom routines to load your data into HP Vertica. You create custom libraries using the HP Vertica SDK to handle various steps in the loading process.

There are many similarities in developing the different types of functions. They can even coexist in the same library. The main difference is the base class you use for your UDF (see Developing a UDF for details).

Developing a User Defined Function in C++

To create a User Defined Function (UDF) in C++, you need to create two classes:

- A function class that performs the actual processing you want the UDF to perform.
- A factory class that tells HP Vertica the name of the UDF and its parameters and return values.

The class you use depends on whether you are creating a scalar or transform UDF (see UDF Types for details).

The following sections explain how you develop and compile the code for your UDF.

HP Vertica C++ SDK Data Types

The HP Vertica SDK has typedefs and classes for representing HP Vertica data types within your UDF code. Using these typedefs ensures data type compatibility between the data your UDF processes and generates and the HP Vertica database. The following table describes some of the typedefs available. Consult the HP Vertica SDK API Documentation for a complete list, as well as lists of helper functions to convert and manipulate these data types.

Type Definition	Description
Interval	An HP Vertica interval
IntervalYM	An HP Vertica year-to-month interval.
Timestamp	An HP Vertica timestamp
vint	A standard HP Vertica 64-bit integer
vint_null	A null value for integer values
vbool	A Boolean value in HP Vertica
vbool_null	A null value for a Boolean data types
vfloat	An HP Vertica floating point value
VString	String data types (such as varchar and char) **Note:** Do not use a VString object to hold an intermediate result. Use a std::string or char[] instead.
VNumeric	Fixed-point data types from HP Vertica

Notes

- When making some HP Vertica SDK API calls (such as `VerticaType::getNumericLength`) on objects that have the correct data type. To minimize overhead and improve performance, most of the APIs do not check the data types of the objects on which they are called. Calling a function on an incorrect data type can result in an error.
- A NULL HP Vertica value string data type is converted into an empty C++ string.

Setting up a C++ UDF Development Environment

You should develop your UDF code on the same Linux platform that you use on your HP Vertica database cluster. This will ensure that your UDF library is compatible with the HP Vertica version deployed on your cluster.

At a minimum, you need to install the following on your development machine:

- g++ and its associated tool chain such as ld. (**Note:** some Linux distributions package g++ separately from gcc.)
- A copy of the HP Vertica SDK. See The HP Vertica SDK for details.

Note: The HP Vertica binaries are compiled using the default version of g++ installed on the supported Linux platforms. While other versions of g++ (or even entirely different compilers) may produce compatible libraries, only the platform’s default g++ version is supported for compiling UDFs.

While not required, the following additional software packages are highly recommended:

- make, or some other build-management tool.
- gdb or some other debugger.
- Valgrind, or similar tools that detect memory leaks.

You should also have access to a non-production HP Vertica database for testing and debugging. You may want to install a single-node HP Vertica database on your development machine for easier development.

If you want to use any third-party libraries (for example, statistical analysis libraries), you need to install them on your development machine. (If you do not statically link these libraries into your UDF library, you also have to install them on every node in the cluster. See Compiling Your C++ UDF for details.)

The C++ HP Vertica SDK

The HP Vertica C++ Software Development Kit (SDK) is distributed as part of the server installation. It contains the source and header files you need to create your UDF library, as well as several sample source files that you can use as a basis for your own UDFs.

The SDK files are located in the sdk subdirectory off of the root HP Vertica server directory (usually, `/opt/vertica/sdk`). This directory contains:

- `include` which contains the headers and source files needed to compile UDF libraries.
- `examples` which contains the source code and sample data for UDF examples.
- `doc` which contains the API documentation for the HP Vertica SDK.

Running the Examples

See the README file in the examples directory for instructions on compiling and running the examples. Running the examples not only helps you understand how a UDF works, it also helps you ensure your development environment is properly set up to compile UDF libraries.

Note: You can copy /opt/vertica/sdk/examples to your home directory and run "make" to build the example libraries. You must have a g++ development environment installed. To install a g++ development environment on Red Hat systems, run `yum install gcc gcc-c++ make`.

Include File Overview

There are two files in the include directory you need when compiling your UDF:

- `Vertica.h` is the main header file for the SDK. Your UDF code needs to include this file in order to find the SDK's definitions.
- `Vertica.cpp` contains support code that needs to be compiled into the UDF library.

Much of the HP Vertica SDK API is defined in the `VerticaUDx.h` header file (which is included by the `Vertica.h` file). If you're curious, you may want to review the contents of this file in addition to reading the API documentation.

The HP Vertica C++ SDK API Documentation

This documentation only provides a brief overview of the classes and class functions defined by the C++ User Defined Function API. To learn more, see the HP Vertica SDK API. You can find this documentation in two locations:

- In the same directory as the other HP Vertica SDK files: `/opt/vertica/sdk/doc`.
- Included with the full documentation set, available either online or for download. See Installing HP Vertica Documentation in the Installation Guide.

Developing a User Defined Scalar Function

A UDSF function returns a single value for each row of data it reads. It can be used anywhere a built-in HP Vertica function can be used. You usually develop a UDF to perform data manipulations that are too complex or too slow to perform using SQL statements and functions. UDFs also let you use analytic functions provided by third-party libraries within HP Vertica while still maintaining high performance.

The topics in this section guide you through developing a UDSF.

UDSF Requirements

There are several requirements for your UDSF:

- Your UDSF must return a value for every input row (unless it generates an error, see Handling Errors for details). Failing to return a value for a row will result in incorrect results (and potentially destabilizing the HP Vertica server if not run in Fenced Mode).

- A UDSF cannot have more than 32 arguments.

If you intend your UDSF to run in unfenced mode, it is vital you pay attention to these additional precautions (although fenced-mode UDSFs should following these guidelines as well).

- Your UDSF must not allow an exception to be passed back to HP Vertica. Doing so could result in a memory leak, as any memory allocated by the exception will not be reclaimed. It is a good practice to use a top-level try-catch block to catch any stray exceptions that may be thrown by your code or any functions or libraries your code calls.

- If your UDSF allocates its own memory, you must make **absolutely sure** it properly frees it. Failing to free even a single byte of allocated memory can have huge consequences if your UDF is called to operate on a multi-million row table. Instead of having your code allocate its own memory, you should use the `vt_alloc macro`, which uses HP Vertica's own memory manager to allocate and track memory. This memory is guaranteed to be properly disposed of when your UDSF completes execution. See Allocating Resources for UDFs for more information.

- In general, remember that unfenced UDSFs run within the HP Vertica process. Any problems it causes may result in database instability or even data loss.

UDSF Class Overview

You create your UDSF by subclassing two classes defined by the HP Vertica SDK: `Vertica::ScalarFunction` and `Vertica::ScalarFunctionFactory`.

The `ScalarFunctionFactory` performs two roles:

- It lists the parameters accepted by the UDSF and the data type of the UDSF's return value. HP Vertica uses this data when you call the CREATE FUNCTION SQL statement to add the function to the database catalog.

- It returns an instance of the UDSF function's `ScalarFunction` subclass that HP Vertica can call to process data.

The `ScalarFunction` class is where you define the `processBlock` function that performs the actual data processing. When a user calls your UDSF function in a SQL statement, HP Vertica bundles together the data from the function parameters and sends it to the `processBlock` statement.

The input and output of the `processBlock` function are supplied by objects of the `Vertica::BlockReader` and `Vertica::BlockWriter` class. They define functions that you use to read the input data and write the output data for your UDSF.

In addition to `processBlock`, the `ScalarFunction` class defines two optional class functions that you can implement to allocate and free resources: `setup` and `destroy`. You should use these class functions to allocate and deallocate resources that you do not allocate through the UDF API (see Allocating Resources for UDFs for details).

The ServerInterface Class

All of the class functions that you will define in your UDSF receive an instance of the `ServerInterface` class as a parameter. This object is used by the underlying HP Vertica SDK code to make calls back into the HP Vertica process. For example, the macro you use to instantiate a member of your `ScalarFunction` subclass (`vt_createFuncObj`) needs a pointer to this object to able able to ask HP Vertica to allocate the memory for the new object. You generally will not interact with this object directly, but instead pass it along to HP Vertica SDK function and macro calls.

Subclassing ScalarFunction

The `ScalarFunction` class is the heart of a UDSF. Your own subclass must contain a single class function named `processBlock` that carries out all of the processing that you want your UDSF to perform.

> **Note:** While the name you choose for your `ScalarFunction` subclass does not have to match the name of the SQL function you will later assign to it, HP considers making the names the same a best practice.

The following example shows a very basic subclass of `ScalarFunction` called `Add2ints`. As the name implies it adds two integers together, returning a single integer result. It also demonstrates including the main HP Vertica SDK header file (`HP Vertica.h`) and using the HP Vertica namespace. While not required, using the namespace saves you from having to prefix every HP Vertica SDK class reference with `HP Vertica::`.

```
// Include the top-level Vertica SDK file
#include "Vertica.h"
```

```
// Using the Vertica namespace means we don't have to prefix all
// class references with Vertica::
using namespace Vertica;
/*
 * ScalarFunction implementation for a UDSF that adds
 * two numbers together.
 */
class Add2Ints : public ScalarFunction
 {
    public:
   /*
    * This function does all of the actual processing for the UDF.
    * In this case, it simply reads two integer values and returns
    * their sum.
    *
    * The inputs are retrieved via arg_reader
    * The outputs are returned via arg_writer
    */
    virtual void processBlock(ServerInterface &srvInterface,
                            BlockReader &arg_reader,
                            BlockWriter &res_writer)
    {
    // While we have input to process
        do {
            // Read the two integer input parameters by calling the
            // BlockReader.getIntRef class function
            const vint a = arg_reader.getIntRef(0);
            const vint b = arg_reader.getIntRef(1);
            // Call BlockWriter.setInt to store the output value, which is the
            //  two input values added together
            res_writer.setInt(a+b);
            // Finish writing the row, and advance to the next output row
            res_writer.next();
            // Continue looping until there are no more input rows
        } while (arg_reader.next());
  }
};
```

The majority of the work in developing a UDSF is creating your `processBlock` class function. This is where all of the processing in your function occurs. Your own UDSF should follow the same basic pattern as this example:

- Read in a set of parameters from the `BlockReader` object using data-type-specific class functions.
- Process the data in some manner.
- Output the resulting value using one of the `BlockWriter` class's data-type-specific class functions.
- Advance to the next row of output and input by calling `BlockWriter.next()` and `BlockReader.next()`.

This process continues until there are no more rows data to be read (`BlockReader.next()` returns false).

Notes

- You must make sure that `processBlock` reads all of the rows in its input and outputs a single value for each row. Failure to do so can corrupt the data structures that HP Vertica reads to get the output of your UDSF. The only exception to this rule is if your `processBlock` function uses the `vt_report_error` macro to report an error back to HP Vertica (see Handling Errors for more). In that case, HP Vertica does not attempt to read the incomplete result set generated by the UDSF.
- Writing too many output rows can cause HP Vertica an out of bounds error.

Subclassing ScalarFunctionFactory

The `ScalarFunctionFactory` class tells HP Vertica metadata about your User Defined Scalar Function (UDSF): its number of parameters and their data types, as well as the data type of its return value. It also instantiates a member of the UDSF's `ScalarFunction` subclass for HP Vertica.

After defining your factory class, you need to call the `RegisterFactory` macro. This macro instantiates a member of your factory class, so HP Vertica can interact with it and extract the metadata it contains about your UDSF.

The following example shows the `ScalarFunctionFactory` subclass for the example `ScalarFunction` function subclass shown in Subsclassing ScalarFunction.

```
/*
 * This class provides metadata about the ScalarFunction class, and
 * also instantiates a member of that class when needed.
 */
class Add2IntsFactory : public ScalarFunctionFactory
{
       // return an instance of Add2Ints to perform the actual addition.
       virtual ScalarFunction *createScalarFunction(ServerInterface &interface)
       {
               // Calls the vt_createFuncObj to create the new Add2Ints class instance.
               return vt_createFuncObj(interface.allocator, Add2Ints);
       }
       // This function returns the description of the input and outputs of the
       // Add2Ints class's processBlock function.  It stores this information in
       // two ColumnTypes objects, one for the input parameters, and one for
       // the return value.
       virtual void getPrototype(ServerInterface &interface,
       ColumnTypes &argTypes,
       ColumnTypes &returnType)
       {
               // Takes two ints as inputs, so add ints to the argTypes object
               argTypes.addInt();
               argTypes.addInt();
               // returns a single int, so add a single int to the returnType object.
               // Note that ScalarFunctions *always* return a single value.
               returnType.addInt();
       }
};
```

There are two required class functions you must implement your `ScalarFunctionFactory` subclass:

- `createScalarFunction` instantiates a member of the UDSF's `ScalarFunction` class. The implementation of this function is simple—you just supply the name of the `ScalarFunction` subclass in a call to the `vt_createFuncObj` macro. This macro takes care of allocating and instantiating the class for you.

- `getPrototype` tells HP Vertica about the parameters and return type for your UDSF. In addition to a `ServerInterface` object, this function gets two `ColumnTypes` objects. All you ned to do in this function is to call class functions on these two objects to build the list of parameters and the single return value type.

After you define your `ScalarFunctionFactory` subclass, you need to use the `RegisterFactory` macro to make the factory available to HP Vertica. You just pass this macro the name of your factory class.

The getReturnType Function

If your function returns a sized column (a return data type whose length can vary, such as a varchar) or a value that requires precision, you need to implement a class function named `getReturnType`. This function is called by HP Vertica to find the length or precision of the data being returned in each row of the results. The return value of this function depends on the data type your `processBlock` function returns:

- CHAR or VARCHAR return the maximum length of the string.

- NUMERIC types specify the precision and scale.

- TIME and TIMESTAMP values (with or without timezone) specify precision.

- INTERVAL YEAR TO MONTH specifies range.

- INTERVAL DAY TO SECOND specifies precision and range.

If your UDSF does not return one of these data types, it does not need a `getReturnType` function.

The input to `getReturnType` function is a `SizedColumnTypes` object that contains the input argument types along with their lengths that will be passed to an instance of your `processBlock` function. Your implementation of `getReturnType` has to extract the data types and lengths from this input and determine the length or precision of the output rows. It then saves this information in another instance of the `SizedColumnTypes` class.

The following demonstration comes from one of the UDSF examples that is included with the HP Vertica SDK. This function determines the length of the VARCHAR data being returned by a UDSF that removes all spaces from the input string. It extracts the return value as a `VerticaType` object, then uses the `getVarcharLength` class function to get the length of the string.

```
// Determine the length of the varchar string being returned.
virtual void getReturnType(ServerInterface &srvInterface,
                           const SizedColumnTypes &argTypes,
                           SizedColumnTypes &returnType)
{
    const VerticaType &t = argTypes.getColumnType(0);
    returnType.addVarchar(t.getVarcharLength());
}
```

The RegisterFactory Macro

Once you have completed your `ScalarFunctionFactory` subclass, you need to register it using the `RegisterFactory` macro. This macro instantiates your factory class and makes the metadata it contains available for HP Vertica to access. To call this macro, you just pass it the name of your factory class.

```
// Register the factory with HP Vertica
RegisterFactory(Add2IntsFactory);
```

Setting Null Input and Volatility Behavior

Normally, HP Vertica calls your UDSF for every row of data in the query. There are two cases where HP Vertica could avoid calling your UDSF code:

- If your function returns NULL when any parameter is a NULL, HP Vertica can just return NULL without having to call the function. This optimization is also helpful since you do not need to handle null input parameters in your UDSF code.

- If your function produces the same output value given the same input parameters, HP Vertica can cache the function's return value. If the UDSF is called with the same set of input parameters again, it can return the cached value, rather than calling your UDSF.

Letting HP Vertica know about these behaviors of your function allows it to optimize queries containing your UDSF.

You indicate the volatility and null handling of your function by setting the `vol` and `strict` fields in your `ScalarFunctionFactory` class's constructor.

Volatility Settings

To indicate your function's volatility, set the `vol` field to one of the following values:

Value	Description
VOLATILE	Repeated calls to the function with the same input parameters always result in different values. HP Vertica always calls volatile functions for each invocation.
IMMUTABLE	Calls to the function with the same input parameters always results in the same output.
STABLE	Repeated calls to the function with the same input *within the same statement* returns the same output. For example, a function that returns the current user name would be stable since the user cannot change within a statement, but could change between statements.
DEFAULT_ VOLATILITY	The default volatility. This is the same as VOLATILE.

The following example code shows a version of the Add2ints example factory class that makes the function immutable.

```
class Add2intsImmutableFactory : public Vertica::ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface &srvIn
terface)
    { return vt_createFuncObj(srvInterface.allocator, Add2ints); }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }

public:
    Add2intsImmutableFactory() {vol = IMMUTABLE;}
};
RegisterFactory(Add2intsImmutableFactory);
```

Null Input Behavior

To indicate how your function reacts to NULL input, set the `strictness` field to one of the following values.

Value	Description
CALLED_ON_NULL_INPUT	The function must be called, even if one or more input values are NULL.
RETURN_NULL_ON_ NULL_INPUT	The function always returns a NULL value if any of its inputs are NULL.
STRICT	A synonym for RETURN_NULL_ON_NULL_INPUT

Value	Description
DEFAULT_STRICTNESS	The default strictness setting. This is the same as CALLED_ON_NULL_INPUT.

The following example demonstrates setting the null behavior of Add2ints so HP Vertica does not call the function with NULL values.

```
class Add2intsNullOnNullInputFactory : public Vertica::ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface &srvIn
terface)
    { return vt_createFuncObj(srvInterface.allocator, Add2ints); }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }

public:
    Add2intsNullOnNullInputFactory() {strict = RETURN_NULL_ON_NULL_INPUT;}
};
RegisterFactory(Add2intsNullOnNullInputFactory);
```

Deploying and Using UDSFs

To deploy a UDSF on your HP Vertica database:

1. Copy the UDF shared library file (`.so`) that contains your function to a node on your HP Vertica cluster.
2. Connect to the node where you copied the library (for example, using **vsql**).
3. Use the CREATE LIBRARY statement to load the UDF library into HP Vertica. You pass this statement the location of the UDF library file you copied to the node earlier. HP Vertica distributes the library to each node in the cluster, and each HP Vertica process loads a copy of the library.
4. Use the CREATE FUNCTION statement to add the functions to HP Vertica's catalog. This maps a SQL function name to the name of the UDSF's factory class. If you are not sure of the name of the UDSF's factory class, you can list all of the UDFs in the library (see Listing the UDFs Contained in a Library for details).

The following example demonstrates loading the `Add2ints` UDSF that is included in the SDK examples directory. It assumes that the `ScalarFunctions.so` library that contains the function has been copied to the dbadmin user's home directory on the initiator node.

```
=> CREATE LIBRARY ScalarFunctions AS
```

```
-> '/home/dbadmin/ScalarFunctions.so';
CREATE LIBRARY
=> CREATE FUNCTION Add2Ints AS LANGUAGE 'C++'
-> NAME 'Add2IntsFactory' LIBRARY ScalarFunctions;
CREATE FUNCTION
```

After creating the Add2ints UDSF, it can be used almost everywhere a built-in function can be used:

```
=> SELECT Add2Ints(27,15);
 Add2ints
----------
       42
(1 row)
=> SELECT * FROM MyTable;
  a  | b
-----+----
   7 |  0
  12 |  2
  12 |  6
  18 |  9
   1 |  1
  58 |  4
 450 | 15
(7 rows)
=> SELECT * FROM MyTable WHERE Add2ints(a, b) > 20;
  a  | b
-----+----
  18 |  9
  58 |  4
 450 | 15
(3 rows)
```

See Also

- CREATE LIBRARY
- CREATE FUNCTION (UDF)

Developing a User Defined Transform Function in C++

A User Defined Transform Function (UDTF) reads one or more arguments (treated as a row of data), and returns zero or more rows of data consisting of one or more columns. The schema of the output table does not need to correspond to the schema of the input table—they can be totally different. The UDTF can return any number of output rows for each row of input.

UDTFs can only be used in the SELECT list that contains just the UDTF call and a required OVER clause.

Unlike other types of User Defined Functions, UDTFs do not have a limit on the number of arguments that they can accept. Most other types of UDFs have a maximum of 32 arguments..

The topics in this section guide you through developing a UDTF.

UDTF Requirements

There are several requirements for UDTFs:

- UDTF's are run after 'group by', but before the final 'order by', when used in conjunction with 'group by' and 'order by' in a statement.
- The UDTF can produce as little or as many rows as it wants as output. However, each row it outputs must be complete. Advancing to the next row without having added a value for each column results in incorrect results.
- Your UDTF must not allow an exception to be passed back to Vertica. You should use a top-level try-catch block to catch any stray exceptions that may be thrown by your code or any functions or libraries your code calls.
- If your UDTF allocates its own memory, you must make absolutely sure it properly frees it. Failing to free even a single byte of allocated memory can have huge consequences if your UDF is called to operate on a multi-million row table. Instead of having your code allocate its own memory, you should use the `vt_alloc macro`, which uses Vertica's own memory manager to allocate and track memory. This memory is guaranteed to be properly disposed of when your UDTF finishes executing. See Allocating Resources for UDFs for more information.
- If you intend your UDTF to run in unfenced mode, you need to ensure it is error-free. Any errors in an unfenced UDF can result in database instability or even data loss.

UDTF Class Overview

You create your UDTF by subclassing two classes defined by the HP Vertica SDK: `Vertica::TransformFunction` and `Vertica::TransformFunctionFactory`.

The `TransformFunctionFactory` performs two roles:

- It provides the number of parameters and their and data types accepted by the UDTF and the number of output columns and their data types UDTF's output. HP Vertica uses this data when you call the CREATE FUNCTION SQL statement to add the function to the database catalog.
- It returns an instance of the UDTF function's `TransformFunction` subclass that HP Vertica can call to process data.

The `TransformFunction` class is where you define the `processPartition` function, which performs the data processing that you want your UDTF to perform. When a user calls your UDTF function in a SQL SELECT statement, HP Vertica sends a partition of data to the `processPartition` statement.

The input and output of the `processPartition` function are supplied by objects of the `Vertica::PartitionReader` and `Vertica::PartitionWriter` class. They define functions that you use to readthe input data and write the output data for your UDTF.

In addition to `processPartition`, the `TransformFunction` class defines two optional class functions that you can implement to allocate and free resources: `setup` and `destroy`. You should use these class functions to allocate and deallocate resources that you do not allocate through the UDF API (see Allocating Resources for UDFs for details).

The ServerInterface Class

All of the class functions that you will define in your UDF receive an instance of the `ServerInterface` class as a parameter. This object is used by the underlying HP Vertica SDK code to make calls back into the HP Vertica process. For example, the macro you use to instantiate a member of your `TransformFunction` subclass (`vt_createFuncObj`) needs a pointer to a class function on this object to able able to ask HP Vertica to allocate the memory for the new object. You generally will not interact with this object directly, but instead pass it along to HP Vertica SDK function and macro calls.

Subclassing TransformFunction

Your subclass of `Vertica::TransformFunction` is where you define the processing you want your UDTF to perform. The only required function in this class is `processPartition`, which reads the parameters sent to your UDTF via a `Vertica::PartitionReader` object, and writes output values to a `Vertica::PartitionWriter` object.

The following example shows a subclass of `TransformFunction` named `StringTokenizer` that breaks input strings into individual words, returning each on its own row. For example:

```
=> SELECT * FROM t;
         text
------------------------
 row row row your boat
 gently down the stream
(2 rows)
=> SELECT tokenize(text) OVER (partition by text) FROM t;
 words
```

```
--------
 gently
 down
 the
 stream
 row
 row
 row
 your
 boat
(9 rows)
```

Notice that the number of rows in the result table (and the name of the results column) are different than the input table. This is one of the strengths of a UDTF.

The following code defines the `StringTokenizer` class.

```
#include "Vertica.h"
#include <sstream>
// Use the Vertica namespace to make referring
// to SDK classes easier.
using namespace Vertica;
using namespace std;
// The primary class for the StringTokenizer UDTF.
class StringTokenizer : public TransformFunction {
    // Called for each partition in the table. Recieves the data from
    // The source table and
    virtual void processPartition(ServerInterface &srvInterface,
                                  PartitionReader &inputReader,
                                  PartitionWriter &outputWriter) {
        try {
            // Loop through the input rows
            do {
                // Get a single varchar as input.
                const VString &sentence = inputReader.getStringRef(0);
                // If input string is NULL, then output is NULL as well
                if (sentence.isNull())
                {
                    VString &word = outputWriter.getStringRef(0);
                    word.setNull();
                    outputWriter.next();
                }
                else
                {
                    // Otherwise, let's tokenize the string and output the words
                  std::string tmp = sentence.str();
                    istringstream ss(tmp);
                    do
                    {
                        std::string buffer;
                        ss >> buffer;

                        // Copy to output
                        if (!buffer.empty()) {
                            VString &word = outputWriter.getStringRef(0);
                            word.copy(buffer);
```

```
                              outputWriter.next();
                          }
                      } while (ss);
                  }
              }  while (inputReader.next()); // Loop until no more input rows
          } catch (exception& e) {
              // Standard exception. Quit.
              vt_report_error(0, "Exception while processing partition: [%s]", e.what());
          }
      }
};
```

The `processPartition` function in this example follows a pattern that you will follow in your own UDTF: it loops over all rows in the table partition that HP Vertica sends it, processing each row. For UDTF's you do not have to actually process every row. You can exit your function without having read all of the input without any issues. You may choose to do this if your UDTF is performing some sort search or some other operation where it can determine that the rest of the input is unneeded.

Extracting Parameters

The first task your UDTF function needs to perform in its main loop is to extract its parameters. You call a data-type specific function in the `PartitionReader` object to extract each input parameter. All of these functions take a single parameter: the column number in the input row that you want to read. In this example, `processPartition` extracts the single `VString` input parameter from the `PartitionReader` object. The `VString` class represents an HP Vertica string value (VARCHAR or CHAR).

In more complex UDTFs, you may need to extract multiple values. This is done the same way as shown in the example, calling the data-type specific function to extract the value of each column in the input row.

Note: In some cases, you may want to determine the number and types of parameters using `PartitionReader`'s `getNumCols` and `getTypeMetaData` functions, instead of just hard-coding the data types of the columns in the input row. This is useful if you want your `TransformFunction` to be able to process input tables with different schemas. You can then use different `TransformFunctionFactory` classes to define multiple function signatures that call the same `TransformFunction` class. See Subclassing TransformFunctionFactory for more information.

Handling Null Values

When developing UDTFs, you often need to handle NULL input values in a special manner. In this example, a NULL input value results in a NULL output value, which is handled as a special case. After writing a NULL to the output, `processPartition` moves on to the next input row.

Processing Input Values

After handling any NULL values, the `processPartition` shown in the example moves on to performing the actual processing. It breaks the string into individual words and adds each word to its own row in the output.

Writing Output

After your UDTF has performed its processing, it may need to write output. Unlike a UDSF, outputting data is optional for a UDTF. However, if it does write output, it must supply values for all of the output columns you defined for your UDTF (see Subclassing TransformFunctionFactory for details on how you specify the output columns of your UDTF) . There are no default values for your output. If you want to output a NULL value in one of the columns, you must explicitly set it.

Similarly to reading input columns, there are function on the `PartitionWriter` object for writing each type of data to the output row. In this case, the example calls the `PartitionWriter` object's `getStringRef` function to allocate a new `VString` object to hold the word it needs to output. Once it has copied the buffer containing the word, the example calls `PartitionWriter.next()` to complete the output row.

Advancing to the Next Input Row

In most UDTFs, processing will continue until all of the rows of input have been read. You advance to the next row by calling `ProcessReader.next()`. This function returns true if there is another row of input data to process and false if all the data in the partition has been read. Once the input rows are exhausted, your UDTF usually exits so its results are returned back to HP Vertica.

Subclassing TransformFunctionFactory

Your subclass of the `TransformFunctionFactory` provides metadata about your UDTF to HP Vertica. Included in this information is the function's name, number and data type of parameters, and the number and data types of output columns.

There are three required functions you need to implement in your `TransformFunctionFactory`:

- `getPrototype` returns two `ColumnTypes` objects that describe the columns your UDTF takes as input and returns as output.
- `createTransformFunction` instantiates a member of your `TransformFunction` subclass that HP Vertica can call to process data.
- `getReturnType` tells HP Vertica details about the output values: the width of variable sized data types (such as VARCHAR) and the precision of data types that have settable precision (such as TIMESTAMP). You can also set the names of the output columns using in this function.

Note: The `getReturnType` function is optional for User Defined Scalar Functions since they do not return a table, and therefore do not require column names. It is required for UDTFs.

The following example shows the factory class that corresponds to the `TransformFunction` subclass shown in Subclassing TransformFunction.

```
class TokenFactory : public TransformFunctionFactory {
// Tell Vertica that StringTokenizer reads in a row with 1 string,
    // and returns a row with 1 string
    virtual void getPrototype(ServerInterface &srvInterface, ColumnTypes
                              &argTypes, ColumnTypes &returnType) {
        argTypes.addVarchar();
        returnType.addVarchar();
    }
    // Tell Vertica the maxiumu return string length will be, given the input
    // string length. Also names the output column. This function is only
    // necessary for columns that have a variable size (i.e. strings) or
    // have to report their precision.
    virtual void getReturnType(ServerInterface &srvInterface,
                               const SizedColumnTypes &input_types,
                               SizedColumnTypes &output_types) {
        int input_len = input_types.getColumnType(0).getStringLength();
        // Output size will never be more than the input size
        // Also sets the name of the output column.
        output_types.addVarchar(input_len, "words");
    }
    virtual TransformFunction *createTransformFunction(ServerInterface
               &srvInterface) {
        return vt_createFuncObj(srvInterface.allocator, StringTokenizer);
    }
};
```

The `getPrototype` function is straightforward. You call functions on the `ColumnTypes` objects to set the data types of the input and output columns for your function. In this example, the UDTF takes a single VARCHAR column as input and returns a single VARCHAR column as output, so it calls the `addVarchar()` function on both of the `ColumnTypes` objects. See the `ColumnTypes` entry in the HP Vertica API documentation for a full list of the data type functions you can call to set input and output column types.

The `getReturnType` function is similar to `getPrototype`, but instead of returning just the data types of the output columns, this function returns the precision of data types that require it (INTERVAL, INTERVAL YEAR TO MONTH, TIMESTAMP, TIMESTAMP WITH TIMEZONE, or VNumeric) or the maximum length of variable-width columns (VARCHAR). This example just returns the length of the input string, since the output will never be longer than the input string. It also sets the name of the output column to "words."

Note: You do not have to supply a name for an output column in this function, since the column name has a default value of "". However, if you do not supply a column name here, the SQL statements that call your UDTF must provide aliases for the unnamed columns or they will fail with an error message. From a usability standpoint, its easier for you to supply the column names here once, rather than to force all of the users of your function to supply their own column names for each call to the UDTF.

`createTransformFunction` is essentially boilerplate code. It just calls the `vt_returnFuncObj` macro with the name of the `TransformFunction` class associated with this factory class. This macro takes care of instantiating a copy of the `TransformFunction` class that HP Vertica can use to process data.

Registering the UDTF Factory Subclass

The final step in creating your UDTF is to call the `RegisterFactory` macro. This macro ensures that your factory class is instantiated when HP Vertica loads the shared library containing your UDTF. having your factory class instantiated is the only way that HP Vertica can find your UDTF and determine what its inputs and outputs are.

The `RegisterFactory` macro just takes the name of your factory class:

```
RegisterFactory(TokenFactory);
```

Creating Multi-Phase UDTFs

Multi-phase UDTFs let you break your data processing into multiple steps. Using this feature, your UDTFs can perform processing in a way similar to Hadoop or other MapReduce frameworks. You can use the first phase to break down and gather data, and then use subsequent phases to process the data. For example, the first phase of your UDTF could extract specific types of user interactions from a web server log stored in the column of a table, and subsequent phases could perform analysis on those interactions.

Multi-phase UDTFs also let you decide where processing should should occur: locally on each node, or throughout the cluster. If your multi-phase UDTF is like a MapReduce process, you want the first phase of your multi-phase UDTF to process data that is stored locally on the node where the instance of the UDTF is running. This prevents large segments of data from being copied around the HP Vertica cluster. Depending on the type of processing being performed in later phases, you may choose to have the data segmented and distributed across the HP Vertica cluster.

Each phase of the UDTF is the same as a traditional (single-phase) UDTF: it receives a table as input, and generates a table as output. The schema for each phase's output does not have to match its input, and each phase can output as many or as few rows as it wants. You create a subclass of `TransformFunction` to define the processing performed by each stage. If you already have a `TransformFunction` from a single-phase UDTF that performs the processing you want a phase of your multi-phase UDTF to perform, you can easily adapt it to work within the multi-phase UDTF.

What makes a multi-phase UDTF different from a traditional UDTF is the factory class you use. You define a multi-phase UDTF using a subclass of `MultiPhaseTransformFunctionFactory`, rather than the `TransformFunctionFactory`. This factory class acts as a container for all of the phases in your multi-step UDTF. It provides HP Vertica with the input and output requirements of the entire multi-phase UDTF (through the `getPrototype` function), and a list of all the phases in the UDTF.

Within your subclass of the `MultiPhaseTransformFunctionFactory` class, you define one or more subclasses of `TransformFunctionPhase`. These classes fill the same role as `TransformFunctionFactory` class for each phase in your multi-phase UDTF. They define the input and output of each phase and create instances of their associated `TransformFunction` classes to perform the processing for each phase of the UDTF. In addition to these subclasses, your `MultiPhaseTransformFunctionFactory` includes fields that provide a handle to an instance of each of the `TransformFunctionPhase` subclasses.

The following code fragment is from the InvertedIndex UDTF example distributed with the HP Vertica SDK. It demonstrates subclassing the `MultiPhaseTransformFunctionFactory` including two `TransformFunctionPhase` subclasses that defines the two phases in this UDTF.

```
class InvertedIndexFactory : public MultiPhaseTransformFunctionFactory
{
public:
   /**
    * Extracts terms from documents.
    */
   class ForwardIndexPhase : public TransformFunctionPhase
   {
       virtual void getReturnType(ServerInterface &srvInterface,
                                  const SizedColumnTypes &inputTypes,
                                  SizedColumnTypes &outputTypes)
       {
           // Sanity checks on input we've been given.
           // Expected input: (doc_id INTEGER, text VARCHAR)
           vector<size_t> argCols;
           inputTypes.getArgumentColumns(argCols);
           if (argCols.size() < 2 ||
               !inputTypes.getColumnType(argCols.at(0)).isInt() ||
               !inputTypes.getColumnType(argCols.at(1)).isVarchar())
               vt_report_error(0, "Function only accepts two arguments"
                                 "(INTEGER, VARCHAR))");
           // Output of this phase is:
           //   (term_freq INTEGER) OVER(PBY term VARCHAR OBY doc_id INTEGER)
           // Number of times term appears within a document.
           outputTypes.addInt("term_freq");
           // Add analytic clause columns: (PARTITION BY term ORDER BY doc_id).
           // The length of any term is at most the size of the entire document.
           outputTypes.addVarcharPartitionColumn(
                inputTypes.getColumnType(argCols.at(1)).getStringLength(),
                "term");
           // Add order column on the basis of the document id's data type.
           outputTypes.addOrderColumn(inputTypes.getColumnType(argCols.at(0)),
                                      "doc_id");
       }
       virtual TransformFunction *createTransformFunction(ServerInterface
                &srvInterface)
       { return vt_createFuncObj(srvInterface.allocator, ForwardIndexBuilder); }
   };
   /**
    * Constructs terms' posting lists.
    */
   class InvertedIndexPhase : public TransformFunctionPhase
   {
       virtual void getReturnType(ServerInterface &srvInterface,
                                  const SizedColumnTypes &inputTypes,
                                  SizedColumnTypes &outputTypes)
       {
           // Sanity checks on input we've been given.
```

```
            // Expected input:
            //   (term_freq INTEGER) OVER(PBY term VARCHAR OBY doc_id INTEGER)
            vector<size_t> argCols;
            inputTypes.getArgumentColumns(argCols);
            vector<size_t> pByCols;
            inputTypes.getPartitionByColumns(pByCols);
            vector<size_t> oByCols;
            inputTypes.getOrderByColumns(oByCols);
            if (argCols.size() != 1 || pByCols.size() != 1 || oByCols.size() != 1 ||
                !inputTypes.getColumnType(argCols.at(0)).isInt() ||
                !inputTypes.getColumnType(pByCols.at(0)).isVarchar() ||
                !inputTypes.getColumnType(oByCols.at(0)).isInt())
                vt_report_error(0, "Function expects an argument (INTEGER) with "
                                "analytic clause OVER(PBY VARCHAR OBY INTEGER)");
            // Output of this phase is:
            //   (term VARCHAR, doc_id INTEGER, term_freq INTEGER, corp_freq INTEGER).
            outputTypes.addVarchar(inputTypes.getColumnType(
                                     pByCols.at(0)).getStringLength(),"term");
            outputTypes.addInt("doc_id");
            // Number of times term appears within the document.
            outputTypes.addInt("term_freq");
            // Number of documents where the term appears in.
            outputTypes.addInt("corp_freq");
        }

        virtual TransformFunction *createTransformFunction(ServerInterface
                 &srvInterface)
        { return vt_createFuncObj(srvInterface.allocator, InvertedIndexBuilder); }
    };
    ForwardIndexPhase fwardIdxPh;
    InvertedIndexPhase invIdxPh;
    virtual void getPhases(ServerInterface &srvInterface,
         std::vector<TransformFunctionPhase *> &phases)
    {
        fwardIdxPh.setPrepass(); // Process documents wherever they're originally stored.
        phases.push_back(&fwardIdxPh);
        phases.push_back(&invIdxPh);
    }
    virtual void getPrototype(ServerInterface &srvInterface,
                              ColumnTypes &argTypes,
                              ColumnTypes &returnType)
    {
        // Expected input: (doc_id INTEGER, text VARCHAR).
        argTypes.addInt();
        argTypes.addVarchar();
        // Output is: (term VARCHAR, doc_id INTEGER, term_freq INTEGER, corp_freq INTEGER)
        returnType.addVarchar();
        returnType.addInt();
        returnType.addInt();
        returnType.addInt();
    }
};
RegisterFactory(InvertedIndexFactory);
```

Most of the code in this example is similar to the the code in a `TransformFunctionFactory` class:

- Both `TransformFunctionPhase` subclasses implement the `getReturnType` function, which describes the output of each stage. This is the similar to the `getPrototype` function from the `TransformFunctionFactory` class. However, this function also lets you control how the data is partitioned and ordered between each phase of your multi-phase UDTF.

  The first phase calls `SizedColumnTypes::addVarcharPartitionColumn` (rather than just `addVarcharColumn`) to set the phase's output table to be partitioned by the column containing the extracted words. It also calls `SizedColumnTypes::addOrderColumn` to order the output table by the document ID column. It calls this function instead of one of the data-type-specific functions (such as `addIntOrderColumn`) so it can pass the data type of the original column through to the output column.

  Note: Any order by column or partition by column set by the final phase of the UDTF in its `getReturnType` function is ignored. Its output is returned to the initiator node rather than partitioned and reordered then sent to another phase.

- The `MultiPhaseTransformFunctionFactory` class implements the `getPrototype` function, that defines the schemas for the input and output of the multi-phase UDTF. This function is the same as the `TransformFunctionFactory::getPrototype` function.

The unique function implemented by the `MultiPhaseTransformFunctionFactory` class is `getPhases`. This function defines the order in which the phases are executed. The fields that represent the phases are pushed into this vector in the order they should execute.

The `MultiPhaseTransformFunctionFactory.getPhase` function is also where you flag the first phase of the UDTF as operating on data stored locally on the node (called a "pre-pass" phase) rather than on data partitioned across all nodes. Using this option increases the efficiency of your multi-phase UDTF by avoiding having to move significant amounts of data around the HP Vertica cluster.

Note: Only the first phase of your UDTF can be a pre-pass phase. You cannot have multiple pre-pass phases, and no later phase can be a pre-pass phase.

To mark the first phase as pre-pass, you call the `TransformFunctionPhase::setPrepass` function of the first phase's TransformFunctionPhase instance from within the `getPhase` function.

Notes

- You need to ensure that the output schema of each phase matches the input schema expected by the next phase. In the example code, each `TransformFunctionPhase::getReturnType` implementation performs a sanity check on its input and output schemas. Your `TransformFunction` subclasses can also perform these checks in their `processPartition` function.

- There is no built-in limit on the number of phases that your multi-phase UDTF can have. However, more phases use more resources. When running in fenced mode, HP Vertica may terminate UDTFs that use too much memory. See UDF Resource Use.

Deploying and Using User Defined Transforms

To deploy a UDTF on your HP Vertica database:

1. Copy the UDF shared library file (`.so`) that contains your function to a node on your HP Vertica cluster.

2. Connect to the node where you copied the library (for example, using **vsql**).

3. Use the CREATE LIBRARY statement to load the UDF library into HP Vertica. You pass this statement the location of the UDF library file you copied to the node earlier. HP Vertica distributes the library to each node in the cluster.

4. Use the CREATE TRANSFORM FUNCTION statement to add the function to the HP Vertica catalog. This maps a SQL function name to the name of the UDF's factory class. If you are not sure of the name of the UDF's factory class, you can list all of the UDFs in the library (see Listing the UDFs Contained in a Library for details).

The following example demonstrates loading the Tokenize UDTF that is included in the SDK examples directory. It assumes that the `TransformFunctions.so` library that contains the function has been copied to the dbadmin user's home directory on the initiator node.

```
=> CREATE LIBRARY TransformFunctions AS
-> '/home/dbadmin/TransformFunctions.so';
CREATE LIBRARY
=> CREATE TRANSFORM FUNCTION tokenize
-> AS LANGUAGE 'C++' NAME 'TokenFactory' LIBRARY TransformFunctions;
CREATE TRANSFORM FUNCTION
=> CREATE TABLE T (url varchar(30), description varchar(2000));
CREATE TABLE
=> INSERT INTO T VALUES ('www.amazon.com','Online retail merchant and provider of cloud s
ervices');
 OUTPUT
--------
      1
(1 row)
=> INSERT INTO T VALUES ('www.hp.com','Leading provider of computer hardware and imaging
solutions');
 OUTPUT
--------
      1
(1 row)
=> INSERT INTO T VALUES ('www.vertica.com','World''s fastest analytic database');
 OUTPUT
--------
      1
(1 row)
=> COMMIT;
COMMIT
=> -- Invoke the UDT
=> SELECT url, tokenize(description) OVER (partition by url) FROM T;
       url       |   words
```

```
-----------------+-----------
 www.amazon.com  | Online
 www.amazon.com  | retail
 www.amazon.com  | merchant
 www.amazon.com  | and
 www.amazon.com  | provider
 www.amazon.com  | of
 www.amazon.com  | c
 www.amazon.com  | loud
 www.amazon.com  | services
 www.hp.com      | Leading
 www.hp.com      | provider
 www.hp.com      | of
 www.hp.com      | computer
 www.hp.com      | hardware
 www.hp.com      | and
 www.hp.com      | im
 www.hp.com      | aging
 www.hp.com      | solutions
 www.vertica.com | World's
 www.vertica.com | fastest
 www.vertica.com | analytic
 www.vertica.com | database
(22 rows)
```

UDTF Query Restrictions

A query that includes a UDTF cannot contain:

- Any statements other than the SELECT statement containing the call to the UDTF and a PARTITION BY expression
- Any other analytic function
- A call to another UDTF
- A TIMESERIES clause
- A pattern matching clause
- A gap filling and interpolation clause

Partitioning By Data Stored on Nodes

UDTFs usually need to process data partitioned in a specific way. For example, a UDTF may process a web server log file to determine how many hits were referred from each partner web site. This UDTF needs to have its input partitioned by a referrer column, so that each instance of the UDTF sees the hits generated by a particular partner so it can total the number of hits. When you execute this UDTF, you supply a PARTITION BY clause to partition data in this way—each node in the HP Vertica database partitions the the data it stores, sends some of these partitions off to other nodes, and then consolidates the partitions it receives from other nodes runs an instance of the UDTF to process them.

If your UDTF does not need to process data partitioned in a particular way, you can make its processing much more efficient by eliminating the overhead of partitioning the data. Instead, each instance of the UDTF processes just the data that is stored locally by the node on which it is running. As long as your UDTF does not need to see data partitioned in any particular manner (for example, a UDTF that parses data out of an Apache log file), you can tremendously speed up its processing using this option.

You tell your UDTF to only process local data by using the PARTITION AUTO clause, rather than specifying a column or expression to use to partition the data. For example, to call a UDTF that parses a locally-stored Apache log file, you could use the following statement:

```
SELECT ParseLogFile('/data/apache/log*') OVER (PARTITION AUTO);
```

Using PARTITION AUTO to Process Local Data

UDTFs usually need to process data partitioned in a specific way. For example, a UDTF that processes a web server log file to count the number of hits referred by each partner web site UDTF needs to have its input partitioned by a referrer column. Each instance of the UDTF sees the hits referred by a particular partner site so it can count them. When you execute this UDTF, you supply a PARTITION BY clause to partition data in this way—each node in the HP Vertica database partitions the the data it stores, sends some of these partitions off to other nodes, and then consolidates the partitions it receives from other nodes runs an instance of the UDTF to process them.

If your UDTF does not need its input data partitioned in a particular way, you can make its processing much more efficient by eliminating the overhead of partitioning the data. Instead, you have each instance of the UDTF process just the data that is stored locally by the node on which it is running. As long as your UDTF does not need to see data partitioned in any particular manner (for example, a UDTF that parses data out of an Apache log file), you can tremendously speed up its processing using this option.

You tell your UDTF to only process local data by using the PARTITION AUTO clause, rather than specifying a column or expression to use to partition the data. You need to supply a source table that is replicated across all nodes and contains a single row (similar to the DUAL table). For example, to call a UDTF that parses locally-stored Apache log files, you could use the following statements:

```
=> CREATE TABLE rep (dummy INTEGER) UNSEGMENTED ALL NODES;
CREATE TABLE
=> INSERT INTO rep VALUES (1);
 OUTPUT
--------
      1
(1 row)
=> SELECT ParseLogFile('/data/apache/log*') OVER (PARTITION AUTO) FROM rep;
```

Developing a User Defined Aggregate Function

Aggregate Functions perform an operation on a set of values and return one value. HP Vertica provides standard built-in Aggregate Functions such as AVG, MAX, and MIN. User Defined Aggregate Functions (UDAF) allow you to create custom aggregate functions specific to your needs.

UDAF's perform operations on a set of rows and reads one input argument and returns one output column.

Example code for User Defined Aggregates is available in `/opt/vertica/sdk/examples/AggregateFunctions`. See Setting up a C++ UDF Development Environment for details on setting up your environment and The HP Vertica SDK for details on building the examples.

User Defined Aggregate Function Requirements

User Defined Aggregates work similarly to the built in HP Vertica aggregate functions.

User Defined Aggregates:

- Support a single input column (or set) of values.
- Provide a single output column.
- Supports automatic **RLE** decompression. RLE input is decompressed before it is sent to a User Defined Aggregate.
- Can be used with the GROUP BY and HAVING clauses. Only columns appearing in the GROUP BY clause can be selected.
- Correlated subquery with User Defined Aggregates is not supported.
- Cannot be used in a query containing multiple distinct User Defined Aggregate functions.
- Cannot be used in a query containing a single User Defined Aggregate and one or more non-distinct User Defined Aggregate functions.
- Must not allow an exception to be passed back to HP Vertica. Doing so could result in a memory leak, because any memory allocated by the exception is not reclaimed. It is a best practice to use a top-level try-catch block to catch any stray exceptions that may be thrown by your code or any functions or libraries your code calls.

UDAF Class Overview

You create your UDAF by subclassing two classes defined by the Vertica SDK:

- Vertica::AggregateFunctionFactory
- Vertica::AggregateFunction

The AggregateFunctionFactory Class

The AggregateFunctionFactory class specifies metadata information such as the argument and return types of your aggregate function. It provides these methods for you to customize:

- getPrototype() - Defines the number of parameters and data types accepted by the function. There is a single parameter for aggregate functions.
- getIntermediateTypes() - Defines the intermediate variable(s) used by the function.
- getParameterType() - Defines the names and types of parameters that this function uses (optional).
- getReturnType() - Defines the type of the output column.

HP Vertica uses this data when you call the CREATE AGGREGATE FUNCTION SQL statement to add the function to the database catalog.

The AggregateFunctionFactory returns an AggregateFunction instance that HP Vertica can call to process data.

The AggregateFunction Class

The AggregateFunction class provides these methods for you to customize:

- initAggregate() - Initializes the class, defines variables, and sets the starting value for the variables. This function must be idempotent.
- aggregate() - The main aggregation operation.
- combine() - If multiple instances of aggregate is run, then combine is called to combine all the sub-aggregations into a final aggregation. Although this method may not be called, it must be defined.
- terminate() - Terminates the function and returns the result as a column.

Important: The aggregate() function may be not operate on the complete input set all at once. Depending on how the data is stored, multiple instances of aggregate may run on the data set. For this reason, initAggregate() must be idempotent.

The AggregateFunction class also provides optional methods that you can implement to allocate and free resources: `setup` and `destroy`. You should use these class functions to allocate and deallocate resources that you do not allocate through the UDAF API (see Allocating Resources for UDFs for details).

The ServerInterface Class

All of the class functions that you will define in your UDAF receive an instance of the ServerInterface class as a parameter. This object is used by the underlying HP Vertica SDK code to make calls back into the Vertica process. For example, the macro you use to instantiate a member of your AggregateFunction subclass (vt_createFuncObj) needs a pointer to a class function on this object to able able to ask HP Vertica to allocate the memory for the new object. You generally will not interact with this object directly, but instead pass it along to HP Vertica SDK function and macro calls.

Subclassing Aggregate Function

Example Subclass of AggregateFunction

The following shows a subclass of AggregateFunction named ag_max that returns the highest value from a column of numbers.

The **initAggregate function** in this example gets the first argument to the function as a vfloat and sets the initial value to zero. Note that the example, as written for simplicity, does not take into account negative input values. Any negative values are returned as 0.

```
virtual void initAggregate(ServerInterface &srvInterface,
                   IntermediateAggs &aggs)
{
    vfloat &max = aggs.getFloatRef(0);
    max = 0;
}
```

The **aggregate function** compares the current maximum with the maximum value it has seen so far, and if the new value is higher, then the new value becomes the highest value.

```
void aggregate(ServerInterface &srvInterface,
               BlockReader &arg_reader,
               IntermediateAggs &aggs)
{
    vfloat &max = aggs.getFloatRef(0);
    do {
        const vfloat &input = arg_reader.getFloatRef(0);
        // if input is bigger than the current max, make the max = input
        if (input > max) {
           max = input;
        }
    } while (arg_reader.next());
}
```

The **combine function** takes as its input the intermediate results of other invocations of aggregate method(s) and determines the overall max value from the complete input set.

```
virtual void combine(ServerInterface &srvInterface,
                        IntermediateAggs &aggs,
                        MultipleIntermediateAggs &aggs_other)
    {
        vfloat &myMax = aggs.getFloatRef(0);
        // Combine all the other intermediate aggregates
        do {
            const vfloat &otherMax   = aggs_other.getFloatRef(0);
               // if input is bigger than the current max, make the max = input
               if (otherMax > myMax) {
                       myMax = otherMax;
               }
        } while (aggs_other.next());
    }
```

The **terminate function** is called when all input has been evaluated by the aggregate method. It returns the max value to the result writer.

```
    virtual void terminate(ServerInterface &srvInterface,
        BlockWriter &res_writer,
        IntermediateAggs &aggs)
        {
            // Metadata about the type (to allow creation)
            const vfloat max = aggs.getFloatRef(0);
            res_writer.setFloat(max);
        }
```

Example Code:

```
class Max : public AggregateFunction
{
    virtual void initAggregate(ServerInterface &srvInterface,
    IntermediateAggs &aggs)
    {
        vfloat &max = aggs.getFloatRef(0);
        max = 0;
    }
    void aggregate(ServerInterface &srvInterface,
    BlockReader &arg_reader,
    IntermediateAggs &aggs)
    {
        vfloat &max = aggs.getFloatRef(0);
        do {
            const vfloat &input = arg_reader.getFloatRef(0);
            // if input is bigger than the current max, make the max = input
            if (input > max) {
                max = input;
            }
        } while (arg_reader.next());
    }
    virtual void combine(ServerInterface &srvInterface,
    IntermediateAggs &aggs,
    MultipleIntermediateAggs &aggs_other)
    {
        vfloat
        &myMax = aggs.getFloatRef(0);
        // Combine all the other intermediate aggregates
        do {
            const vfloat &otherMax   = aggs_other.getFloatRef(0);
            // if input is bigger than the current max, make the max = input
            if (otherMax > myMax) {
                myMax = otherMax;
            }
        } while (aggs_other.next());
    }
    virtual void terminate(ServerInterface &srvInterface,
    BlockWriter &res_writer,
    IntermediateAggs &aggs)
    {
        // Metadata about the type (to allow creation)
        const vfloat max = aggs.getFloatRef(0);
        res_writer.setFloat(max);
    }
    InlineAggregate()
};
```

InlineAggregate() is called at the end of the class. You do not need to implement this function, but it automatically optimizes aggregates and speeds up calculations when called on your code.

Subclassing AggregateFunctionFactory

Example Subclass of AggregateFunctionFactory

The following shows a subclass of AggregateFunctionFactory named ag_maxFactory.

The **getPrototype** method allows you to define the variables that are sent to your aggregate function and returned to HP Vertica after your aggregate function runs. The example below accepts and returns a float from/to HP Vertica:

```
    virtual void getPrototype(ServerInterface &srvfloaterface, ColumnTypes &argTypes,
                              ColumnTypes &returnType)
    {
        argTypes.addFloat();
        returnType.addFloat();
    }
```

The **getIntermediateTypes** method defines any intermediate variables that you use in your aggregate function. Intermediate variables are used to pass data between multiple invocations of an aggregate function ("max" in the example below) to combine results until a final result can be computed:

```
    virtual void getIntermediateTypes(ServerInterface &srvInterface,
                                      const SizedColumnTypes &input_types,
                                      SizedColumnTypes &intermediateTypeMetaData)
    {
        intermediateTypeMetaData.addFloat("max");
    }
```

The **getParameterType** method defines the name and types of parameters that the aggregate function uses:

```
    virtual void getParameterType(ServerInterface &srvInterface, SizedColumnTypes
                                  &parameter_types)
    {
        parameter_types.addFloat();
    }
```

The **getReturnType** method defines and assigns a value to the variable that is sent back to HP Vertica when the aggregate function completes:

```
    virtual void getReturnType(ServerInterface &srvfloaterface,
                               const SizedColumnTypes &input_types,
                               SizedColumnTypes &output_types)
    {
        output_types.addFloat("max");
    }
```

Example Code:

```
class MaxFactory : public AggregateFunctionFactory
{
    virtual void getPrototype(ServerInterface &srvfloaterface, ColumnTypes &argTypes,
                              ColumnTypes &returnType)
    {
        argTypes.addFloat();
        returnType.addFloat();
    }
    virtual void getIntermediateTypes(ServerInterface &srvInterface,
                                      const SizedColumnTypes &input_types,
                                      SizedColumnTypes &intermediateTypeMetaData)
    {
        intermediateTypeMetaData.addFloat("max");
    }
    virtual void getReturnType(ServerInterface &srvfloaterface,
                               const SizedColumnTypes &input_types,
                               SizedColumnTypes &output_types)
    {
        output_types.addFloat("max");
    }
    virtual AggregateFunction *createAggregateFunction(ServerInterface &srvfloaterface)
    { return vt_createFuncObj(srvfloaterface.allocator, Max); }
};
```

User Defined Aggregate - Complete Example

This is a complete working example of a User Defined Aggregate:

```
#include "Vertica.h"
#include <sstream>
#include <iostream>
using namespace Vertica;
using namespace std;
/****
 * Example implementation of Aggregate "Max" Function
 ***/
class Max : public AggregateFunction
{
    virtual void initAggregate(ServerInterface &srvInterface,
                       IntermediateAggs &aggs)
    {
        vfloat &max = aggs.getFloatRef(0);
        max = 0;
    }

    void aggregate(ServerInterface &srvInterface,
                   BlockReader &arg_reader,
                   IntermediateAggs &aggs)
    {
        vfloat &max = aggs.getFloatRef(0);
        do {
            const vfloat &input = arg_reader.getFloatRef(0);
        // if input is bigger than the current max, make the max = input
       if (input > max) {
               max = input;
            }
        } while (arg_reader.next());
    }
    virtual void combine(ServerInterface &srvInterface,
                         IntermediateAggs &aggs,
                         MultipleIntermediateAggs &aggs_other)
    {
        vfloat       &myMax      = aggs.getFloatRef(0);
        // Combine all the other intermediate aggregates
        do {
            const vfloat &otherMax   = aggs_other.getFloatRef(0);
               // if input is bigger than the current max, make the max = input
               if (otherMax > myMax) {
                       myMax = otherMax;
               }
        } while (aggs_other.next());
    }
    virtual void terminate(ServerInterface &srvInterface,
                           BlockWriter &res_writer,
                           IntermediateAggs &aggs)
    {
        // Metadata about the type (to allow creation)
        const vfloat max = aggs.getFloatRef(0);
        res_writer.setFloat(max);
    }
    InlineAggregate()
};
class MaxFactory : public AggregateFunctionFactory
{
    virtual void getIntermediateTypes(ServerInterface &srvInterface,
                                      const SizedColumnTypes &input_types,
                                      SizedColumnTypes &intermediateTypeMetaData)
    {
        intermediateTypeMetaData.addFloat("max");
    }
    virtual void getPrototype(ServerInterface &srvfloaterface,
                              ColumnTypes &argTypes,
                              ColumnTypes &returnType)
    {
        argTypes.addFloat();
        returnType.addFloat();
    }
    virtual void getReturnType(ServerInterface &srvfloaterface,
                               const SizedColumnTypes &input_types,
                               SizedColumnTypes &output_types)
    {
        output_types.addFloat("max");
    }
    virtual AggregateFunction *createAggregateFunction(ServerInterface &srvfloaterface)
    { return vt_createFuncObj(srvfloaterface.allocator, Max); }
};
RegisterFactory(MaxFactory);
```

Example usage:

```
=> CREATE LIBRARY AggregateFunctions AS
-> '/opt/vertica/sdk/examples/build/AggregateFunctions.so';
CREATE LIBRARY
=> create aggregate function ag_max as LANGUAGE 'C++'
-> name 'MaxFactory' library AggregateFunctions;
CREATE AGGREGATE FUNCTION
=> select * from example;
 i
----
 1
 1
 8
 2
 3
 0
 5
 13
 21
(9 rows)
=> select ag_max(i) from example;
 ag_max
--------
   21
(1 row)
```

Developing a User Defined Analytic Function

User Defined Analytic Functions (UDAnFs) are User Defined Functions that are used for analytics. See Using SQL Analytics for an overview of HP Vertica's built-in analytics. Like User Defined Scalar Functions (UDSFs), UDAnFs must output a single value for each row of data read. Unlike UDSFs, the UDAnF's input reader and output reader can be advanced independently. This feature lets you create UDAnF's where the output value is calculated over multiple rows of data. By advancing the reader and writer independently, you can create functions similar to the built-in analytic functions such as LAG, which uses data from prior rows to output a value for the current row.

Note: Analytic functions are only supported in C++. They cannot be developed in R or Java.

User Defined Analytic Function Requirements

User Defined Analytic Functions (UDAnFs) must meet several requirements:

- They must produce a single output value for each row in the partition of data they are given to process.
- They must have 32 or fewer arguments.
- Like all UDF's, they must not allow an exception to be passed back to HP Vertica. Doing so could lead to issues such as memory leaks (caused by the memory allocated by the exception never being freed). Your UDAnF should always contain a top-level try-catch block to catch any stray exceptions caused by your code or libraries your code calls.
- If they allocate resources on their own, they must properly free it. Even a single byte of allocated memory that is not freed can become an issue in a UDAnF that is called over millions of rows. This is especially true if your UDAnF runs in unfenced mode (directly within the HP Vertica process) since it could destabilize the database. Instead of directly allocating memory, your function should use the memory allocation macros in the HP Vertica SDK. See Allocating Resources for UDFs for details.

UDAnF Class Overview

To create a UDAnF, you need to subclass two classes defined by the HP Vertica SDK:

- `Vertica::AnalyticFunction` which carries out the processing you want your function to perform.
- `Vertica::AnalyticFunctionFactory` which defines the input and output data types of your function. It instantiates your AnalyticFunction subclass when needed.

The `AnalyticFunction` class is where you define the `processPartition` function that performs the analytic processing you want your function to perform. When a user calls your UDAnF function in a SQL SELECT statement, HP Vertica breaks the data into partitions as specified in the OVER clause of the query, and calls the `processPartition` function to process them. If the query does not a partition by column, the

The input and output of the `processPartition` function are supplied by objects of the `Vertica::AnalyticPartitionReader` and `Vertica::AnalyticPartitionWriter` class. They define functions that you use to read the input data and write the output data for your UDAnF.

In addition to `processPartition`, the `AnalyticFunction` class defines two optional class functions that you can implement to allocate and free resources: `setup` and `destroy`. You use these class functions to allocate and deallocate resources that you do not allocate through the UDF API (see Allocating Resources for UDFs for details).

The ServerInterface Class

All of the class functions that you define in your UDAnF receive a `ServerInterface` object as a parameter. It is used by the underlying HP Vertica SDK code to make calls back into the HP Vertica process. For example, the macro you use to instantiate a member of your `AnalyticFunction` subclass (`vt_createFuncObj`) needs a pointer to this object to able able to ask HP Vertica to allocate the memory for the new object. You generally will not interact with this object directly, but instead pass it along to HP Vertica SDK function and macro calls.

Subclassing AnalyticFunction

Your subclass of `Vertica::AnalyticFunction` is where your User Defined Analytic Function (UDAnF) performs its processing. The one function in the class that you must implement, `processPartition`, reads a partition of data, performs some sort of processing, and outputs single value for each input row.

For example, a UDAnF that ranks rows based on how they are ordered has a `processPartition` function that reads each row, determines how to rank the row, and outputs the rank value. An example of running a rank function, named `an_rank` is:

```
=> SELECT * FROM hits;
      site       |    date    | num_hits
-----------------+------------+----------
 www.example.com | 2012-01-02 |       97
 www.vertica.com | 2012-01-01 |   343435
 www.example.com | 2012-01-01 |      123
 www.example.com | 2012-01-04 |      112
 www.vertica.com | 2012-01-02 |   503695
 www.vertica.com | 2012-01-03 |   490387
 www.example.com | 2012-01-03 |      123
(7 rows)
=> SELECT site,date,num_hits,an_rank()
-> OVER (PARTITION BY site ORDER BY num_hits DESC)
-> AS an_rank FROM hits;
      site       |    date    | num_hits | an_rank
-----------------+------------+----------+---------
 www.example.com | 2012-01-03 |      123 |       1
 www.example.com | 2012-01-01 |      123 |       1
 www.example.com | 2012-01-04 |      112 |       3
 www.example.com | 2012-01-02 |       97 |        4
 www.vertica.com | 2012-01-02 |   503695 |        1
 www.vertica.com | 2012-01-03 |   490387 |        2
 www.vertica.com | 2012-01-01 |   343435 |        3
(7 rows)
```

As with the built-in RANK analytic function, rows that have the same value for the ORDER BY column (num_hits in this example) have the same rank, but the rank continues to increase, so that next row that has a different ORDER BY key gets a rank value based on the number of rows that preceded it.

HP Vertica calls the `processPartition()` function once for each partition of data. It supplies the partition using an `AnalyticPartitionReader` object from which your function reads its input data. In addition, there is a unique function on this object named `isNewOrderByKey`, which returns a Boolean value indicating whether your function has seen a row with the same ORDER BY key (or keys). This function is very useful for analytic functions (such as the example RANK function) which need to to handle rows with identical ORDER BY keys differently than rows with different ORDER BY keys.

Note: You can specify multiple ORDER BY columns in the SQL query you use to call your UDAnF. The `isNewOrderByKey` function returns true if any of the ORDER BY keys are different than the previous row.

Once your function has finished processing the row of data, you advance it to the next row of input by calling the `AnalyticPartitionReader::next` function.

Your function writes its output value using a `AnalyticPartitionWriter` object that HP Vertica supplies as a parameter to the `processPartition` function. This object has data-type-specific functions to write the output value (such as `setInt`). After setting the output value, call the `AnalyticPartitionWriter::next` function to advance to the next row in the output.

Note: You must be sure that your function produces a row of output for each row of input in the partition. You must also not output more rows than are in the partition, otherwise the zygote size process (if running in Fenced Mode) or HP Vertica itself could generate an out of bounds error.

The following example code defines a `AnalyticFunction` subclass named `Rank`, which implements the ranking function demonstrated earlier. It is based on example code distributed in the examples directory of the SDK.

```
/**
 * User defined analytic function: Rank - works mostly the same as SQL-99 rank
 * with the ability to define as many order by columns as desired
 *
 */
class Rank : public AnalyticFunction
{
    virtual void processPartition(ServerInterface &srvInterface,
                                  AnalyticPartitionReader &inputReader,
                                    AnalyticPartitionWriter &outputWriter)
    {
        // Always use a top-level try-catch block to prevent exceptions from
        // leaking back to Vertica or the fenced-mode side process.
        try {
            rank = 1; // The rank to assign a row
            rowCount = 0; // Number of rows processed so far
            do {
                rowCount++;
                // Do we have a new order by row?
                if (inputReader.isNewOrderByKey()) {
                    // Yes, so set rank to the total number of rows that have been
                    // processed. Otherwise, the rank remains the same value as
                    // the previous iteration.
                    rank = rowCount;
                }
                // Write the rank
                outputWriter.setInt(0, rank);
                // Move to the next row of the output
                outputWriter.next();
            } while (inputReader.next()); // Loop until no more input
        } catch(exception& e) {
            // Standard exception. Quit.
            vt_report_error(0, "Exception while processing partition: %s", e.what());
        }
    }
private:
    vint rank, rowCount;
};
```

In this example, the `processPartition` function does not actually read any of the data from the input row, it just advances through the rows. It does not need to read data since just needs to count the number of rows that have been read and determine whether those rows have the same ORDER BY key as the previous row. If the current row is a new ORDER BY key, then the rank is set to the total number of rows that have been processed. If the current row has the same ORDER BY value as the previous row, then the rank remains the same.

Note that the function has a top-level try-catch block. All of your UDF functions should always have one to prevent stray exceptions from being passed back to HP Vertica (if you run the function unfenced) or the side process.

Subclassing AnalyticFunctionFactory

Your subclass of the `AnalyticFunctionFactory` class provides the following metadata about your UDAnF to HP Vertica:

- The number and data types of your function's input arguments
- Your function's output data type (and its width or precision, if it returns a variable-width data type such as VARCHAR or a data type that has settable precision such as TIMESTAMP)
- The `AnalyticFunction` subclass that implements your function. HP Vertica calls your factory class to instantiate members of this class when it needs to execute your UDAnF.

There are three required functions that your `AnalyticFunctionFactory` subclass must implement:

- `getPrototype` describes the input parameters and output value of your function. You set these values by calling functions on two `ColumnTypes` objects that provided to the function via parameters.
- `createAnalyticFunction` supplies an instance of your `AnalyticFunction` factory that HP Vertica can call to process a UDAnF function call.
- `getReturnType` provides details about your function's output. This function is where you set the width of the output value if your function returns a variable-width value (such as VARCHAR) or the precision of the output value if it has a settable precision (such as TIMESTAMP).

The following example code defines the `AnalyticFunctionFactory` that corresponds with the example `Rank` class shown in Subclassing AnalyticFunction.

```
class RankFactory : public AnalyticFunctionFactory
{
    virtual void getPrototype(ServerInterface &srvInterface,
                                ColumnTypes &argTypes, ColumnTypes &returnType)
    {
        returnType.addInt();
    }
    virtual void getReturnType(ServerInterface &srvInterface,
                               const SizedColumnTypes &inputTypes,
                               SizedColumnTypes &outputTypes)
    {
        outputTypes.addInt();
    }
    virtual AnalyticFunction *createAnalyticFunction(ServerInterface
                                                         &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, Rank); }
};
```

The first function defined by the `RankFactory` subclass, `getPrototype`, sets the data type of the return value. Since the Rank UDAnF does not read input, it does not define any arguments by calling functions on the `ColumnTypes` object passed in the `argTypes` parameter.

The next function is `getReturnType`. If your function returns a data type that needs to define a width or precision, your implementation of the `getReturnType` function calls a function on the `SizedColumnType` object passed in as a parameter to tell HP Vertica the width or precision. See the SDK entry for `SizedColumnTypes` for a list of these functions. `Rank` returns a fixed-width data type (an INTEGER) so it does not need to set the precision or width of its output; it just calls `addInt` to report its output data type.

Finally, `RankFactory` defines the `createAnalyticFunction` function that returns an instance of the `AnalyticFunction` class that HP Vertica can call. This code is mostly boilerplate. All you need to do is add the name of your analytic function class in the call to `vt_createFuncObj()` which takes care of allocating the object for you.

Deploying and Using User Defined Analytic Functions

To deploy a UDAnF on your HP Vertica database:

1. Copy the UDF shared library file (`.so`) that contains your function to a node on your HP Vertica cluster.

2. Connect to the node where you copied the library (for example, using **vsql**).

3. Use the CREATE LIBRARY statement to load the UDF library into HP Vertica. You pass this statement the location of the UDF library file you copied to the node earlier. HP Vertica distributes the library to each node in the cluster.

4. Use the CREATE ANALYTIC FUNCTION statement to add the function to the HP Vertica catalog. This maps a SQL function name to the name of the UDF's factory class. If you are not sure of the name of the UDF's factory class, you can list all of the UDFs in the library (see Listing the UDFs Contained in a Library for details).

The following example demonstrates loading and using the Rank UDAnF that is included in the SDK examples directory. It assumes that the `AnalyticFunctions.so` library that contains the function has been copied to the dbadmin user's home directory on the initiator node.

```
=> CREATE LIBRARY AnalyticFunctions AS '/home/dbadmin/AnalyticFunctions.so';
CREATE LIBRARY
=> CREATE ANALYTIC FUNCTION an_rank AS LANGUAGE 'C++'
-> NAME 'RankFactory' LIBRARY AnalyticFunctions;
CREATE ANALYTIC FUNCTION
=> SELECT * FROM hits;
      site       |    date    | num_hits
-----------------+------------+----------
 www.example.com | 2012-01-02 |       97
 www.vertica.com | 2012-01-01 |   343435
 www.example.com | 2012-01-01 |      123
 www.example.com | 2012-01-04 |      112
 www.vertica.com | 2012-01-02 |   503695
 www.vertica.com | 2012-01-03 |   490387
 www.example.com | 2012-01-03 |      123
(7 rows)
=> SELECT site,date,num_hits,an_rank() over (partition by site order by num_hits desc)
-> from hits;
      site       |    date    | num_hits | ?column?
-----------------+------------+----------+----------
 www.example.com | 2012-01-03 |      123 |        1
 www.example.com | 2012-01-01 |      123 |        1
 www.example.com | 2012-01-04 |      112 |        3
 www.example.com | 2012-01-02 |       97 |        4
 www.vertica.com | 2012-01-02 |   503695 |        1
 www.vertica.com | 2012-01-03 |   490387 |        2
 www.vertica.com | 2012-01-01 |   343435 |        3
(7 rows)
```

Notes

- UDAnFs do not support framing windows using ROWS.
- As with HP Vertica's built-in analytic functions, UDAnFs cannot be used with Pattern Matching Functions.

See Also

- CREATE LIBRARY
- CREATE ANALYTIC FUNCTION

Compiling Your C++ UDF

g++ is the only supported compiler for compiling User Defined Function libraries (see Setting up a C++ UDF Development Environment for details). You should compile your UDF code on the same version of Linux that you use on your HP Vertica cluster.

There are several requirements for compiling your library:

- You must pass the `-shared` and `-fPIC` flags to the linker. The easiest method is to just pass these flags to g++ when you compile and link your library.
- You should also use the `-Wno-unused-value` flag to suppress warnings when macro arguments are not used. Otherwise, you may get "left-hand operand of comma has no effect" warnings.
- You must compile `sdk/include/Vertica.cpp` and link it into your library. The easiest way to do this is to include it in the g++ command to compile your library. This file contains support routines that help your UDF communicate with HP Vertica. Supplying this file as C++ source rather than a library limits library compatibility issues.
- Add the HP Vertica SDK include directory in the include search path using the g++ `-I` flag.

The following command line compiles a UDF contained in a single source file named `MyUDF.cpp` into a shared library named `MyUDF.so`:

```
g++ -D HAVE_LONG_INT_64   -I /opt/vertica/sdk/include -Wall -shared -Wno-unused-value \
      -fPIC -o MyUDF.so MyUDF.cpp /opt/vertica/sdk/include/Vertica.cpp
```

The above command line assumes that the HP Vertica SDK directory is located at `/opt/vertica/sdk/include` (the default location).

Note: HP only supports UDF development on 64-bit architectures. If you must compile your UDF code on a 32-bit system, add the flag `-D__Linux32__` to your compiler command line.

Once you have debugged your UDF and are ready to deploy it, you should recompile using the `-O3` flag to enable compiler optimization.

You can add additional source files to your library by adding them to the command line. You can also compile them separately and then link them together on your own.

Note: The examples subdirectory in the HP Vertica SDK directory contains a make file that you can use as starting point for your own UDF project.

Handling External Libraries

If your UDF code relies on additional libraries (either ones you have developed, or provided by third-parties) you have two options on how you link them to your UDF library:

- Statically link them into your UDF. This is the best option, since your UDF library will not rely on any external files. Since HP Vertica takes care of distributing your library to each node in your cluster, bundling the additional library into your UDF library eliminates any additional work to deploy your UDF.
- Dynamically link the library to your UDF. You may need to use dynamic linking for some third-party libraries that do not allow static linking. In this case, you will need to manually install this external library on each of your HP Vertica nodes. This increases the maintenance you need to perform. It also adds a new step when adding new nodes to the cluster, since you need remember to install the library before adding the node. In addition, you need to ensure the version of the library is the same on each node.

Handling Different Numbers and Types of Arguments

Usually, your UDFs accept a set number of arguments that are a specific data type (called its signature). You can create UDFs that handle multiple signatures, or even accept all arguments supplied to them by the user, using either of these techniques:

- Overloading your UDF by creating multiple factory classes, each of which defines a unique function signature. This technique is best if your UDF just needs to accept a few different signatures (for example, accepting two required and one optional argument).
- Using the special "Any" argument type that tells HP Vertica to send all arguments that the user supplies to your function. Your UDF decides whether it can handle the arguments or not.

The following topics explain each of these techniques.

User Defined Function Overloading

You may want your UDF to accept several different signatures (sets of arguments). For example, you might want your UDF to accept:

- One or more optional arguments.
- One or more argument that can be one of several data types.
- Completely distinct signatures (either all INTEGER or all VARCHAR, for example).

You can create a function with this behavior by creating several factory classes each of which accept a different signature (the number and data types of arguments), and associate a single SQL function name with all of them. You can use the same SQL function name to refer to multiple factory classes as long as the signature defined by each factory is unique. When a user calls your UDF, HP Vertica matches the number and types of arguments supplied by the user to the arguments accepted by each of your function's factory classes. If one matches, HP Vertica uses it to instantiate a function class to process the data.

Multiple factory classes can instantiate the same function class, so you can re-use one function class that is able to process multiple sets of arguments and then create factory classes for each of the function signatures. You can also create multiple function classes if you want.

The following example code demonstrates creating a User Defined Scalar Function (UDSF) that adds two or three integers together. The `Add2or3ints` class is prepared to handle two or three arguments. The `processBlock` function checks the number of arguments that have been passed to it, and adds all two or three of them together. It also exits with an error message if it has been called with less than 2 or more than 3 arguments. In theory, this should never happen, since HP Vertica only calls the UDSF if the user's function call matches a signature on one of the factory classes you create for your function. In practice, it is a good idea to perform this sanity checking, in case your (or someone else's) factory class inaccurately reports a set of arguments your function class cannot handle.

```
#include "Vertica.h"
using namespace Vertica;
using namespace std;
// a ScalarFunction that accepts two or three
// integers and adds them together.
class Add2or3ints : public Vertica::ScalarFunction
{
public:
    virtual void processBlock(Vertica::ServerInterface &srvInterface,
                              Vertica::BlockReader &arg_reader,
                              Vertica::BlockWriter &res_writer)
    {
        const size_t numCols = arg_reader.getNumCols();

        // Ensure that only two or three parameters are passed in
        if ( numCols < 2 || numCols > 3)
            vt_report_error(0, "Function only accept 2 or 3 arguments, "
                                "but %zu provided", arg_reader.getNumCols());
         // Add two integers together
        do {
            const vint a = arg_reader.getIntRef(0);
            const vint b = arg_reader.getIntRef(1);
            vint c = 0;
               // Check for third argument, add it in if it exists.
            if (numCols == 3)
                c = arg_reader.getIntRef(2);
            res_writer.setInt(a+b+c);
            res_writer.next();
        } while (arg_reader.next());
    }
};
// This factory accepts function calls with two integer arguments.
class Add2intsFactory : public Vertica::ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface
                &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, Add2or3ints); }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {   // Accept 2 integer values
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }
};
RegisterFactory(Add2intsFactory);
// This factory defines a function that accepts 3 ints.
class Add3intsFactory : public Vertica::ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface
                &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, Add2or3ints); }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {   // accept 3 integer values
        argTypes.addInt();
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }
};
RegisterFactory(Add3intsFactory);
```

The example has two `ScalarFunctionFactory` classes, one for each signature that the function accepts (two integers and three integers). There is nothing unusual about these factory classes, except that their implementation of `ScalarFunctionFactory::createScalarFunction` both create `Add2or3ints` objects.

The final step is to bind the same SQL function name to both factory classes. You can assign multiple factories to the same SQL function, as long as the signatures defined by each factory's `getPrototype` implementation are different.

```
=> CREATE LIBRARY add2or3IntsLib AS '/home/dbadmin/Add2or3Ints.so';
CREATE LIBRARY
=> CREATE FUNCTION add2or3Ints as NAME 'Add2intsFactory' LIBRARY add2or3IntsLib FENCED;
CREATE FUNCTION
=> CREATE FUNCTION add2or3Ints as NAME 'Add3intsFactory' LIBRARY add2or3IntsLib FENCED;
CREATE FUNCTION
=> SELECT add2or3Ints(1,2);
 add2or3Ints
-------------
           3
(1 row)
=> SELECT add2or3Ints(1,2,4);
 add2or3Ints
-------------
           7
(1 row)
=> SELECT add2or3Ints(1,2,3,4); -- Will generate an error
ERROR 3467:  Function add2or3Ints(int, int, int, int) does not exist, or
permission is denied for add2or3Ints(int, int, int, int)
HINT:  No function matches the given name and argument types. You may
need to add explicit type casts
```

The error message in response to the final call to the add2or3Ints function was generated by HP Vertica, since it could not find a factory class associated with add2or3Ints that accepted four integer arguments. To expand add2or3Ints further, you could create another factory class that accepted this signature, and either change the Add2or3ints ScalarFunction class or create a totally different class to handle adding more integers together. However, adding more classes to accept a each variation in the arguments quickly becomes overwhelming. In that case, you should consider creating a polymorphic UDF (see Creating a Polymorphic UDF for more information).

Creating a Polymorphic UDF

Polymorphic UDFs accept any number and type of argument that the user supplies. HP Vertica does not check the number or types of argument that the user passes to the UDF—it just passes the UDF all of the arguments supplied by the user. It is up to your polymorphic UDF's main processing function (for example, `processBlock` in User Defined Scalar Functions) to examine the number and types of arguments it received and determine if it can handle them.

> **Note:** User Defined Transform Functions (UDTFs) can have an unlimited number of arguments. All other UDFs except UDTFs are limited to a maximum number of 32 arguments.

Polymorphic UDFs are more flexible than using multiple factory classes for your function (see User Defined Function Overloading), since you function can determine at run time if it can process the arguments rather than accepting specific sets of arguments. However, your polymorphic function needs to perform more work to determine whether it can process the arguments that it has been given.

Your polymorphic UDF declares it accepts any number of arguments in its factory's `getPrototype` function by calling the `addAny` function on the `ColumnTypes` object that defines its arguments. This "any parameter" argument type is the only one that your function can declare. You cannot define required arguments and then call `addAny` to declare the rest of the signature as optional. If your function has requirements for the arguments it accepts, your process function must enforce them.

The following example shows an implementation of a `ScalarFunction` that adds together two or more integers.

```
#include "Vertica.h"
using namespace Vertica;
using namespace std;
// Adds two or more integers together.
class AddManyInts : public Vertica::ScalarFunction
{
public:
    virtual void processBlock(Vertica::ServerInterface &srvInterface,
                              Vertica::BlockReader &arg_reader,
                              Vertica::BlockWriter &res_writer)
    {
        // Always catch exceptions to prevent causing the side process or
        // Vertica itself from crashing.
        try
        {
            // Find the number of arguments sent.
            size_t numCols = arg_reader.getNumCols();

            // Make sure at least 2 arguments were supplied
            if (numCols < 2)
                vt_report_error(0, "Function expects at least 2 integer parameters");

            // Make sure all types are ints
            const SizedColumnTypes &inTypes = arg_reader.getTypeMetaData();
            for (int param=0; param < (int)numCols; param++) {
                const VerticaType &t = inTypes.getColumnType(param);
                if (!t.isInt())
                {
                    string typeDesc = t.getPrettyPrintStr();
                    // Report that the user supplied a non-integer value.
                    vt_report_error(0, "Function expects all arguments to be "
                                    "INTEGER. Argument %d was %s", param+1,
                                    typeDesc.c_str());
                }
            }
            do
            { // total up the arguments and write out the total.
                vint total = 0;
                int x;
                // Loop over all params, adding them up.
                for (x=0; x<(int)numCols; x++) {
                    total += arg_reader.getIntRef(x);
                }
                res_writer.setInt(total);
                res_writer.next();
            } while (arg_reader.next());
        }  catch(exception& e) {
           // Standard exception. Quit.
           vt_report_error(0, "Exception while processing partition: [%s]",
                       e.what());
        }
    }
};
// Defines the AddMany function.
class AddManyIntsFactory : public Vertica::ScalarFunctionFactory
{
    // Return the function object to process the data.
    virtual Vertica::ScalarFunction *createScalarFunction(
               Vertica::ServerInterface &srvInterface)
    { return vt_createFuncObj(srvInterface.allocator, AddManyInts); }
    // Define the number and types of arguments that this function accepts
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {
        argTypes.addAny(); // Must be only argument type.
        returnType.addInt();
    }
};
RegisterFactory(AddManyIntsFactory);
```

Most of the work in the example is done by the `ScalarFunction.processBlock` function. It performs two checks on the arguments that have been passed in through the `BlockReader` object:

- Ensures there are at least two arguments
- Checks the data type of all arguments to ensure they are all integers

Once the checks are performed, the example processes the block of data by looping over the arguments and adding them together.

You assign a SQL name to your polymorphic UDF using the same statement you use to assign one to a non-polymorphic UDF. The following demonstration shows how you load and call the polymorphic function from the example.

```
=> CREATE LIBRARY addManyIntsLib AS '/home/dbadmin/AddManyInts.so';
CREATE LIBRARY
=> CREATE FUNCTION addManyInts AS NAME 'AddManyIntsFactory' LIBRARY addManyIntsLib
```

```
FENCED;
CREATE FUNCTION
=> SELECT addManyInts(1,2);
 addManyInts
-------------
           3
(1 row)
=> SELECT addManyInts(1,2,3,40,50,60,70,80,900);
 addManyInts
-------------
        1206
(1 row)
=> SELECT addManyInts(1); -- Too few parameters
ERROR 3412:  Failure in UDx RPC call InvokeProcessBlock(): Error calling
processBlock() in User Defined Object [addManyInts] at
[AddManyInts.cpp:51], error code: 0, message: Exception while processing
partition: [Function expects at least 2 integer parameters]
=> SELECT addManyInts(1,2.232343); -- Wrong data type
ERROR 3412:  Failure in UDx RPC call InvokeProcessBlock(): Error
calling processBlock() in User Defined Object [addManyInts] at
[AddManyInts.cpp:51], error code: 0, message: Exception while
processing partition: [Function expects all arguments to be INTEGER.
Argument 2 was Numeric(7,6)]
```

Notice that the errors returned by last two calls to the function were generated by the `processBlock` function. It is up to your UDF to ensure that the user supplies the correct number and types of arguments to your function and exit with an error if it cannot process them.

Polymorphic UDFs and Schema Search Paths

If a user does not supply a schema name as part of a function call, HP Vertica searches each schema in the schema search path for a function whose name and signature match the function call. See Setting Schema Search Paths in the Administrator's Guide for more information about schema search paths.

Since polymorphic functions do not have a specific signature associated with them, HP Vertica initially skips them when searching for a function to handle the function call. If none of the schemas in the search path contain a function whose name and signature match the function call, HP Vertica searches the schema search path again for a polymorphic function whose name matches the function name in the function call.

This behavior gives precedence to functions whose signature exactly matches the function call. It allows you to create a "catch all" polymorphic function that is called only if none of the non-polymorphic functions with the same name have matching signatures.

This behavior may cause confusion if your users expect the the first polymorphic function in the schema search path to handle a function call. To avoid confusion, you should:

- Avoid using the same name for different functions. You should always uniquely name functions unless you intend to create an overloaded function with multiple signatures.

- When you cannot avoid having functions with the same name in different schemas, always supply the schema name as part of the function call. Using the schema name prevents ambiguity and ensures that HP Vertica uses the correct function to process your function calls.

UDF Parameters

Parameters let you define arguments for your UDFs that remain constant across all of the rows processed by the SQL statement that calls you UDF. Typically, your UDFs accept arguments that come from columns in a SQL statement. For example, in the following SQL statement, the arguments a and b to the add2ints UDSF change value for each row processed by the SELECT statement:

```
=> SELECT a, b, add2ints(a,b) AS 'sum' FROM example;
 a | b  | sum
---+----+-----
 1 |  2 |   3
 3 |  4 |   7
 5 |  6 |  11
 7 |  8 |  15
 9 | 10 |  19
(5 rows)
```

Parameters remain constant for all the rows your UDF processes. You can also make parameters optional so that if the user does not supply it, your UDF uses a default value. For example, the following example demonstrates calling a UDSF named add2intsWithConstant that has a single parameter value named constant whose value is added to each the arguments supplied in each row of input:

```
=> SELECT a, b, add2intsWithConstant(a, b USING PARAMETERS constant=42)
-> AS 'a+b+42' from example;
 a | b  | a+b+42
---+----+--------
 1 |  2 |     45
 3 |  4 |     49
 5 |  6 |     53
 7 |  8 |     57
 9 | 10 |     61
(5 rows)
```

Note: When calling a UDF with parameters, there is no comma between the last argument and the USING PARAMETERS clause.

The topics in this section explain how develop UDFs that accept parameters.

Defining the Parameters Your UDF Accepts

You define the parameters that your UDF accepts in its factory class (`ScalarFunctionFactory`, `AggregateFunctionFactory`, etc.) by implementing the `getParameterType` function. This function is similar to the `getReturnType` function: you call data-type-specific functions on a `SizedColumnTypes` object that is passed in as a parameter. Each function call sets the name, data type, and width or precision (if the data type requires it) of the parameter.

The following code fragment demonstrates adding a single parameter to the add2ints UDSF example. The `getParameterType` function defines a single integer parameter that is named constant.

```
class Add2intsWithConstantFactory : public ScalarFunctionFactory
{
    // return an instance of Add2ints to perform the actual addition.
    virtual ScalarFunction *createScalarFunction(ServerInterface &interface)
    {
        // Calls the vt_createFuncObj to create the new Add2ints class instance.
        return vt_createFuncObj(interface.allocator, Add2intsWithConstant);
    }
    // Report the argument and return types to Vertica
    virtual void getPrototype(ServerInterface &interface,
                              ColumnTypes &argTypes,
                              ColumnTypes &returnType)
    {
        // Takes two ints as inputs, so add ints to the argTypes object
        argTypes.addInt();
        argTypes.addInt();
        // returns a single int.
        returnType.addInt();
    }
    // Defines the parameters for this UDSF. Works similarly to defining
    // arguments and return types.
    virtual void getParameterType(ServerInterface &srvInterface,
                                  SizedColumnTypes &parameterTypes)
    {
        // One INTEGER parameter named constant
        parameterTypes.addInt("constant");
    }
};
RegisterFactory(Add2intsWithConstantFactory);
```

See the HP Vertica SDK entry for `SizedColumnTypes` for a full list of the data-type-specific functions you can call to define parameters.

Getting Parameter Values in UDFs

Your UDF uses the parameter values it declared in its factory class (see Defining the Parameters Your UDF Accepts) in its function class's process function (for example, `processBlock` or `processPartition`). It gets its parameter values from a `ParamReader` object, which is available from the `ServerInterface` object that is passed to your process function. Reading parameters from this object is similar to reading argument values from `BlockReader` or `PartitionReader` objects: you call a data-type-specific function with the name of the parameter to retrieve its value. For example:

```
// Get the parameter reader from the ServerInterface to see if
// there are supplied parameters
ParamReader paramReader = srvInterface.getParamReader();
// Get the value of an integer parameter named constant
const vint constant = paramReader.getIntRef("constant");
```

Note: String data values do not have any of their escape characters processed before they are passed to your function. Therefore, your function may need to process the escape sequences itself if it needs to operate on unescaped character values.

Testing Whether the User Supplied Parameter Values

Unlike arguments, HP Vertica does not immediately return an error if a user's function call does not include a value for a parameter defined by your UDF's factory class. This means that your function can attempt to read a parameter value that the user did not supply. If it does so, HP Vertica returns a non-existent parameter error to the user, and the query containing the function call is canceled. This behavior is fine if you want a parameter to be required by your UDF—just attempt to access its value. If the user didn't supply a value, HP Vertica reports the resulting error about a missing parameter to the user.

If you want your parameter to be optional, you can test whether the user supplied a value for the parameter before attempting to access its value. Your function determines if a value exists for a particular parameter by calling the `ParamReader::containsParameter` function with the parameter's name. If this function returns true, your function can safely retrieve the value. If this function returns false, your UDF can use a default value or change its processing in some other way to compensate for not having the parameter value. As long as your UDF does not try to access the non-existent parameter value, HP Vertica does not generate an error or warning about missing parameters.

Note: If the user passes your UDF a parameter that it has not defined, HP Vertica issues a warning that the parameter is not used. It still executes the SQL statement, ignoring the parameter.

The following code fragment demonstrates using the parameter value that was defined in the example shown in Defining the Parameters Your UDF Accepts. The `Add2intsWithConstant` class defines a function that adds two integer values. If the user supplies it, the function also adds the value of the optional integer parameter named constant.

```
/**
 * A UDSF that adds two numbers together with a constant value.
 *
 */
class Add2intsWithConstant : public ScalarFunction
{
public:
    // Processes a block of data sent by Vertica
    virtual void processBlock(ServerInterface &srvInterface,
                              BlockReader &arg_reader,
                              BlockWriter &res_writer)
    {
        try
            {
                // The default value for the constant parameter is 0.
                vint constant = 0;
```

```
                 // Get the parameter reader from the ServerInterface to see if
                 // there are supplied parameters
                 ParamReader paramReader = srvInterface.getParamReader();
                 // See if the user supplied the constant parameter
                 if (paramReader.containsParameter("constant"))
                     // There is a parameter, so get its value.
                     constant = paramReader.getIntRef("constant");
                 // While we have input to process
                 do
                     {
                         // Read the two integer input parameters by calling the
                         // BlockReader.getIntRef class function
                         const vint a = arg_reader.getIntRef(0);
                         const vint b = arg_reader.getIntRef(1);
                         // Add arguments plus constant
                         res_writer.setInt(a+b+constant);
                         // Finish writing the row, and advance to the next
                         // output row
                         res_writer.next();
                         // Continue looping until there are no more input rows
                     }
                 while (arg_reader.next());
             }
         catch (exception& e)
             {
                 // Standard exception. Quit.
                 vt_report_error(0, "Exception while processing partition: %s",
                     e.what());
             }
    }
};
```

Using Parameters in the Factory Class

In addition to using parameters in your UDF function class, you can also access the parameters in the factory class. You may want to access the parameters to let the user control the input or output values of your function in some way. For example, your UDF can have a parameter that lets the user choose to have your UDF return a single or double-precision value. The process of accessing parameters in the factory class is the same as accessing it in the function class: get a `ParamReader` object from the `ServerInterface::getParamReader` function, them read the parameter values.

Calling UDFs with Parameters

You pass parameters to a UDF by adding a USING PARAMETERS clause in the function call after the last argument. There is no comma between the last argument and the USING PARAMETERS clause. After the USING PARAMETERS clause you add one or more parameter definitions which contains the parameter name, followed by an equal sign, then the parameter's value. Multiple parameter definitions are separated by commas.

Note: Parameter values can be a constant expression (for example `1234 + SQRT(5678)`). You cannot use volatile functions (such as RANDOM) in the expression, since they do not return a constant value. If you do supply a volatile expression as a parameter value, HP Vertica returns an incorrect parameter type warning, and tries to run the UDF without the parameter value. If the UDF requires the parameter, it returns its own error which cancels the query.

The following example demonstrates calling the add2intsWithConstant UDSF example from Defining the Parameters Your UDF Accepts and Getting Parameter Values in UDFs:

```
=> SELECT a, b, add2intsWithConstant(a, b USING PARAMETERS constant=42)
-> AS 'a+b+42' from example;
 a | b  | a+b+42
---+----+--------
 1 |  2 |     45
 3 |  4 |     49
 5 |  6 |     53
 7 |  8 |     57
 9 | 10 |     61
(5 rows)
```

Multiple parameters are separated by commas. The following example calls a version of the tokenize UDTF that has parameters to limit the shortest allowed word and force the words to be output in uppercase.

```
=> SELECT url, tokenize(description USING PARAMETERS
-> minLength=4, uppercase=true) OVER (partition by url) FROM T;
       url       |   words
-----------------+-----------
 www.amazon.com  | ONLINE
 www.amazon.com  | RETAIL
 www.amazon.com  | MERCHANT
 www.amazon.com  | PROVIDER
 www.amazon.com  | CLOUD
 www.amazon.com  | SERVICES
 www.hp.com      | LEADING
 www.hp.com      | PROVIDER
 www.hp.com      | COMPUTER
 www.hp.com      | HARDWARE
 www.hp.com      | IMAGING
 www.hp.com      | SOLUTIONS
 www.vertica.com | WORLD'S
 www.vertica.com | FASTEST
 www.vertica.com | ANALYTIC
 www.vertica.com | DATABASE
(16 rows)
```

The add2intsWithConstant UDSF's constant parameter is optional; calling it without the parameter does not return an error or warning:

```
=> SELECT a,b,add2intsWithConstant(a, b) AS 'sum' FROM example;
 a | b  | sum
---+----+-----
 1 |  2 |   3
 3 |  4 |   7
 5 |  6 |  11
 7 |  8 |  15
 9 | 10 |  19
(5 rows)
```

Calling a UDF with incorrect parameters does generate a warning, but the query still runs:

```
=> SELECT a, b,  add2intsWithConstant(a, b USING PARAMETERS wrongparam=42)
-> AS 'result' from example;
WARNING 4332:  Parameter wrongparam was not registered by the function and cannot
be coerced to a definite data type
 a | b  | result
---+----+--------
 1 |  2 |      3
 3 |  4 |      7
 5 |  6 |     11
 7 |  8 |     15
 9 | 10 |     19
(5 rows)
```

UDF Resource Use

Your UDFs consume at least a small amount of memory by instantiating classes and creating local variables. This basic memory usage by UDFs is small enough that you do not need to be concerned about it.

If your UDF needs to allocate more than one or two megabytes of memory for data structures, or requires access additional resources such as files, you must inform HP Vertica about its resource use. HP Vertica can then ensure that the resources your UDF requires are available before running a query that uses it. Even moderate memory use (10MB per invocation of a UDF, for example) can become an issue if there are many simultaneous queries that call it.

Allocating Resources for UDFs

You have two options for allocating memory and file handles for your User Defined Functions (UDFs):

- Use HP Vertica SDK macros to allocate resources. This is the best method, since it uses HP Vertica's own resource manager, and guarantees that resources used by your UDF are reclaimed. See Allocating Resources with the SDK Macros.

- Allocate resources in your UDFs yourself using standard C++ methods (instantiating objects using `new`, allocating memory blocks using `malloc`, etc.). You must manually free these resources before your UDF exits.

Note: You must be extremely careful if you choose to allocate your own resources in your UDF. Failing to free resources properly will have significant negative impact, especially if your UDF is running in unfenced mode.

Whichever method you choose, you usually allocate resources in a function named `setup` (subclassed from (`Vertica::UDXObject::setup`) in your function class. This function is called after your UDF function object is instantiated, but before HP Vertica calls it to process data.

If you allocate memory on your own in `setup` function, you must free it in a function named destroy (subclassed from `Vertica::UDXObject::destroy`) in your function class. This functions is called after your UDF has performed all of its processing. This function is also called if your UDF returns an error (see Handling Errors).

Note: Always use the `setup` and `destroy` functions to allocate and free resources instead your own constructors and destructors. The memory for your UDF object is allocated from one of HP Vertica's own memory pools. HP Vertica always calls your UDF's destroy function before the it deallocates the object's memory. There is no guarantee that your UDF's destructor is will be called before the object is deallocated. Using the `destroy` function ensures that your UDF has a chance to free its allocated resources before it is destroyed.

The following code fragment demonstrates allocating and freeing memory using a setup and destroy function.

```
class MemoryAllocationExample : public ScalarFunction
{
public:
    uint64* myarray;
    // Called before running the UDF to allocate memory used throughout
    // the entire UDF processing.
    virtual void setup(ServerInterface &srvInterface, const SizedColumnTypes
                        &argTypes)
    {
        try
        {
            // Allocate an array. This memory is directly allocated, rather than
            // letting Vertica do it. Remember to properly calculate the amount
            // of memory you need based on the data type you are allocating.
            // This example divides 500MB by 8, since that's the number of
            // bytes in a 64-bit unsigned integer.
            myarray = new uint64[1024 * 1024 * 500 / 8];
        }
        catch (std::bad_alloc &ba)
        {
            // Always check for exceptions caused by failed memory
            // allocations.
            vt_report_error(1, "Couldn't allocate memory :[%s]", ba.what());
        }

    }

    // Called after the UDF has processed all of its information. Use to free
    // any allocated resources.
    virtual void destroy(ServerInterface &srvInterface, const SizedColumnTypes
                          &argTypes)
    {
        // srvInterface.log("RowNumber processed %d records", *count_ptr);
        try
        {
            // Properly dispose of the allocated memory.
            delete[] myarray;
        }
        catch (std::bad_alloc &ba)
        {
            // Always check for exceptions caused by failed memory
            // allocations.
            vt_report_error(1, "Couldn't free memory :[%s]", ba.what());
        }

    }
```

Allocating Resources with the SDK Macros

The HP Vertica SDK provides three macros to allocate memory:

- `vt_alloc` allocates a block of memory to fit a specific data type (vint, struct, etc.).
- `vt_allocArray` allocates a block of memory to hold an array of a specific data type.

- `vt_allocSize` allocates an arbitrarily-sized block of memory.

All of these macros allocate their memory from memory pools managed by HP Vertica. The main benefit of allowing HP Vertica to manage your UDF's memory is that the memory is automatically reclaimed after your UDF has finished. This ensures there is no memory leaks in your UDF.

Because Vertica Analytics Platform frees this memory automatically, do not attempt to free any of the memory you allocate through any of these macros. Attempting to free this memory results in run-time errors.

Informing HP Vertica of Resource Requirements

When you run your UDF in fenced mode, HP Vertica monitors its use of memory and file handles. If your UDF uses more than a few megabytes of memory or any file handles, it should tell HP Vertica about its resource requirements. Knowing the resource requirements of your UDF allows HP Vertica to determine whether it can run the UDF immediately or needs to queue the request until enough resources become available to run it.

Determining how much memory your UDF requires can be difficult in some case. For example, if your UDF extracts unique data elements from a data set, and there is potentially no bounds on the number of data items. In this case, a useful technique is to run your UDF in a test environment and monitor its memory use on a node as it handles several differently-sized queries, then extrapolate its memory use based on the worst case scenario it may face in your production environment. In all cases, it's usually a good idea to add a safety margin to the amount of memory you tell HP Vertica your UDF uses.

Note: The information on your UDF's resource needs that you pass to HP Vertica is used when planning the query execution. There is no way to change the amount of resources your UDF requests from HP Vertica while the UDF is actually running.

Your UDF informs HP Vertica of its resource needs by implementing the `getPerInstanceResources` function in its factory class (see `Vertica::UDXFactory::getPerInstanceResources` in the SDK documentation). If your UDF's factory class implements this function, HP Vertica calls it to determine the resources your UDF requires.

The `getPerInstanceResources` function receives an instance of the Vertica::VResources struct, which contains fields setting the amount of memory and the number of file handles your UDF may need to use. Your implementation of this function sets these fields based on the maximum resources your UDF may consume for each instance of the UDF function. So, if your UDF's `ProcessBlock` function creates a data structure that uses at most 100MB of memory, your UDF should set the `VResources.scratchMemory` field to at least 104857600 (the number of bytes in 100MB). Rounding up to a number like 115000000 (just under 110MB) is a good idea.

The following `ScalarFunctionFactory` class demonstrates calling `getPerInstanceResources` to inform HP Vertica about the memory requirements of the `MemoryAllocationExample` class shown in Allocating Resources for UDFs. It tells HP Vertica that the UDSF requires 510MB of memory (which is a bit more than the UDSF actually allocates, to be on the safe size).

```
class MemoryAllocationExampleFactory : public ScalarFunctionFactory
{
    virtual Vertica::ScalarFunction *createScalarFunction(Vertica::ServerInterface
                                                           &srvInterface)
    {
        return vt_createFuncObj(srvInterface.allocator, MemoryAllocationExample);
    }
    virtual void getPrototype(Vertica::ServerInterface &srvInterface,
                              Vertica::ColumnTypes &argTypes,
                              Vertica::ColumnTypes &returnType)
    {
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }
    // Tells Vertica the amount of resources that this UDF uses.
    virtual void getPerInstanceResources(ServerInterface &srvInterface,
                                          VResources &res)
    {
        res.scratchMemory += 1024LL * 1024 * 510; // request 510MB of memory
    }
};
```

Setting Memory Limits for Fenced Mode UDFs

HP Vertica calls fenced-mode UDF's implementation of `Vertica::UDXFactory::getPerInstanceResources` to determine if there are enough free resources to run the query containing the UDF (see Informing HP Vertica of Resource Requirements). Since these reports are not generated by actual memory use, they can be inaccurate. Once started by HP Vertica, a UDF could allocate far more memory or file handles than it reported it needs.

The FencedUDxMemoryLimitMB configuration parameter lets you create an absolute memory limit for UDFs. Any attempt by a UDF to allocate more memory than this limit results in a `bad_alloc` exception. For more information on configuration parameters, see Configuration Parameters in the Administrator's Guide. For an example of setting FencedUDxMemoryLimitMB, see How Resource Limits Are Enforced.

How Resource Limits Are Enforced

Before running a query, HP Vertica determines how much memory it requires to run. If the query contains a fenced-mode UDF which implements the `getPerInstanceResources` function in its factory class, HP Vertica calls it to determine the amount of memory the UDF needs and adds this to the total required for the query. Based on these requirements, HP Vertica decides how to handle the query:

- If the total amount of memory required (including the amount that the UDFs report that they need) is larger than the session's MEMORYCAP or **resource pool's** MAXMEMORYSIZE setting, HP Vertica rejects the query. For more information about resource pools, see Resource Pool Architecture in the Administrator's Guide.

- If the amount of memory is below the limit set by the session and resource pool limits, but there is currently not enough free memory to run the query, HP Vertica queues it until enough resources become available.
- If there is enough free resources to run the query, HP Vertica executes it.

Note: HP Vertica has no other way to determine the amount of resources a UDF requires other than the values it reports using the `getPerInstanceResources` function. A UDF could use more resources than it claims, which could cause performance issues for other queries that are denied resources. You can set an absolute limit on the amount of memory UDFs can allocate. See Setting Memory Limits for Fenced Mode UDFs for more information.

If the process executing your UDF attempts to allocate more memory than the limit set by the FencedUDxMemoryLimitMB configuration parameter, it receives a bad_alloc exception. For more information about FencedUDxMemoryLimitMB, see Setting Memory Limits for Fenced Mode UDFs.

Below is the output of loading a UDSF that consumes 500MB of memory, then changing the memory settings to cause out of memory errors. The MemoryAllocationExample UDSF in the following example is just the Add2Ints UDSF example altered as shown in Allocating Resources for UDFs and Informing HP Vertica of Resource Requirements to allocate 500MB of RAM.

```
=> CREATE LIBRARY mylib AS '/home/dbadmin/MemoryAllocationExample.so';
CREATE LIBRARY
=> CREATE FUNCTION usemem AS NAME 'MemoryAllocationExampleFactory' LIBRARY mylib
-> FENCED;
CREATE FUNCTION
=> SELECT usemem(1,2);
 usemem
--------
      3
(1 row)
```

The following statements demonstrate setting the session's MEMORYCAP to lower than the amount of memory that the UDSF reports it uses. This causes HP Vertica to return an error before it executes the UDSF.

```
=> SET SESSION MEMORYCAP '100M';
SET
=> SELECT usemem(1,2);
ERROR 3596:  Insufficient resources to execute plan on pool sysquery
[Request exceeds session memory cap: 520328KB > 102400KB]
=> SET SESSION MEMORYCAP = default;
SET
```

The **resource pool** can also prevent a UDF from running if it requires more memory than is available in the pool. The following statements demonstrate the effect of creating and using a resource pool that has too little memory for the UDSF to run. Similar to the session's MAXMEMORYCAP limit, the pool's MAXMEMORYSIZE setting prevents HP Vertica from executing the query containing the UDSF.

```
=> CREATE RESOURCE POOL small MEMORYSIZE '100M' MAXMEMORYSIZE '100M';
CREATE RESOURCE POOL
=> SET SESSION RESOURCE POOL small;
SET
=> CREATE TABLE ExampleTable(a int, b int);
CREATE TABLE
=> INSERT /*+direct*/ INTO ExampleTable VALUES (1,2);
 OUTPUT
--------
      1
(1 row)
=> SELECT usemem(a, b) FROM ExampleTable;
ERROR 3596:  Insufficient resources to execute plan on pool small
[Request Too Large:Memory(KB) Exceeded: Requested = 523136, Free = 102400 (Limit = 10240
0, Used = 0)]
=> DROP RESOURCE POOL small CASCADE; --Dropping the pool resets the session's pool
DROP RESOURCE POOL
```

Finally, setting the FencedUDxMemoryLimitMB configuration parameter to lower than the UDF actually allocates results in the UDF throwing an exception. This is a different case than either of the previous two examples, since the query actually executes. The UDF's code needs to catch and handle the exception. In this example, it uses the `vt_report_error` macro to report the error back to HP Vertica and exit.

```
=> SELECT set_config_parameter('FencedUDxMemoryLimitMB','300');
    set_config_parameter
----------------------------
 Parameter set successfully
(1 row)
=> SELECT usemem(1,2);
ERROR 3412:  Failure in UDx RPC call InvokeSetup(): Error calling setup() in
User Defined Object [usemem] at [MemoryAllocationExample.cpp:32], error code:
 1, message: Couldn't allocate memory :[std::bad_alloc]
=> SELECT set_config_parameter('FencedUDxMemoryLimitMB','-1');
    set_config_parameter
----------------------------
 Parameter set successfully
(1 row)
=> SELECT usemem(1,2);
 usemem
--------
      3
(1 row)
```

See Also

- SET SESSION RESOURCE_POOL
- SET SESSION MEMORYCAP

- SET_CONFIG_PARAMETER

Handling Errors

If your UDF encounters some sort of error, it can report it back to HP Vertica using the `vt_report_error` macro. When called, this macro halts the execution of the UDF and causes the statement that called the function to fail. The macro takes two parameters: an error number and a error message string. Both the error number and message appear in the error that HP Vertica reports to the user. The error number is not defined by HP Vertica. You can use whatever value that you wish.

For example, the following `ScalarFunction` class divides two integers. To prevent division by zero, it tests the second parameter. If it is zero, the function reports the error back to HP Vertica.

```
/*
 * Demonstrate reporting an error
 */
class Div2ints : public ScalarFunction
{
public:
  virtual void processBlock(ServerInterface &srvInterface,
                            BlockReader &arg_reader,
                            BlockWriter &res_writer)
  {
    // While we have inputs to process
    do
      {
        const vint a = arg_reader.getIntRef(0);
        const vint b = arg_reader.getIntRef(1);
        if (b == 0)
          {
            vt_report_error(1,"Attempted divide by zero");
          }
        res_writer.setInt(a/b);
        res_writer.next();
      }
    while (arg_reader.next());
  }
};
```

Loading and invoking the function demonstrates how the error appears to the user.

```
=> CREATE LIBRARY Div2IntsLib AS '/home/dbadmin/Div2ints.so';
CREATE LIBRARY
=> CREATE FUNCTION div2ints AS LANGUAGE 'C++' NAME 'Div2intsInfo' LIBRARY Div2IntsLib;
CREATE FUNCTION
=> SELECT div2ints(25, 5);
 div2ints
----------
        5
(1 row)
=> SELECT * FROM MyTable;
 a  | b
----+---
```

```
 12 | 6
  7 | 0
 12 | 2
 18 | 9
(4 rows)
=> SELECT * FROM MyTable WHERE div2ints(a, b) > 2;
ERROR:  Error in calling processBlock() for User Defined Scalar Function
div2ints at Div2ints.cpp:21, error code: 1, message: Attempted divide by zero
```

Handling Cancel Requests

You can cancel a query that calls your UDF (usually, by by pressing CTRL+C in **vsql**). How HP Vertica handles the cancelation of the query and your UDF depends on whether your UDF is running in fenced or unfenced mode:

- If your UDF is running in unfenced mode, HP Vertica either stops the function when it requests a new block of input or output, or waits until your function completes running and discards the results.
- If your UDF is running in Fenced Mode, HP Vertica kills the zygote process that is running your function if it continues processing past a timeout.

See Fenced Mode for more information about running functions in fenced mode.

To give you more control over what happens to your function when the user cancels its query, the HP Vertica SDK includes an API for some UDFs to handle cancelation. Any function class that inherits from the `Vertica::UDXObjectCancelable` class can test whether the query calling it has been canceled using a function named `isCanceled`. Your function can also implement a callback function named `cancel` that HP Vertica calls when the function's query is canceled. Currently, the two classes that inherit from `UDXObjectCancelable` are `TransformFunction` and `AnalyticFunction`.

The topics in this section explain how to use the cancel API.

Exiting When the Calling Query Has Been Canceled

The `processPartition` function in your User Defined Transform Function (UDTF) or Analytic Function (UDAnF) can call `Vertica::UDXObjectCancelable.isCancled` to determine if the user has canceled the query that called it. If `isCanceled` returns true, the query has been canceled and your `processPartition` function should exit immediately to prevent it from wasting CPU time. If your UDF is not running in Fenced Mode, HP Vertica cannot halt your function, and has to wait for it to finish. If it is running in fenced mode, HP Vertica can eventually kill the side process running it, but not until it has wasted some processing time.

How often your `processPartition` function calls `isCanceled` depends on how much processing it performs on each row of data. Calling `isCanceled` does add some overhead to your function, so you shouldn't call it too often. For transforms that do not perform lengthy processing, you could check for cancelation every 100 or 1000 rows or so. If your `processPartition` performs extensive processing for each row, you may want to check isCanceled every 10 or so rows.

The following code fragment shows how you could have the `StringTokenizer` UDTF example check whether its query has been canceled:

```
// The primary class for the StringTokenizer UDTF.
class StringTokenizer : public TransformFunction {
    // Called for each partition in the table. Recieves the data from
    // The source table and
    virtual void processPartition(ServerInterface &srvInterface,
                                PartitionReader &inputReader,
                                PartitionWriter &outputWriter) {
        try {
            // Loop through the input rows
            int rowCount = 0; // Count the number of rows processed.
            do {
                rowCount++; // Processing a new row of data
                // Check for cancelation every 100 rows.
                if (rowCount % 100 == 0)
                {
                    if (isCanceled()) // See if query has been canceled
                    {
                        // Log cancelation
                        srvInterface.log("Got canceled!");
                        return; // Exit out of UDTF immediately.
                    }
                }
                // Rest of the function here
                .         .        .
```

This example checks for cancelation after processing 100 rows in the partition of data. If the query has been canceled, the example logs a message, then returns to the caller to exit the function.

Note: You need to strike a balance between adding overhead to your functions by calling `isCanceled` and having your functions waste CPU time by running after their query has been canceled (usually, a rare event). For functions such as `StringTokenizer` which have a low overall processing cost, it usually does not make sense to test for cancelation. The cost of adding overhead to all function calls outweigh the amount of resources wasted by having the function run to completion or having its zygote process killed by HP Vertica on the rare occasions that its query is canceled.

Implementing the Cancel Callback Function

Your User Defined Transform Function (UDTF) or Analytic Function (UDAnF) can implement a `cancel` callback function that HP Vertica calls if the query that called the function has been canceled. You usually implement this function to perform an orderly shutdown of any additional processing that your UDF spawned. For example, you can have your `cancel` function shut down threads that your UDF has spawned or signal a third-party library that it needs to stop processing and exit. Your `cancel` function should leave your UDF's function class ready to be destroyed, since HP Vertica calls the UDF's destroy function after the cancel function has exited.

Notes

- If your UDTF or UDAnF does not implement cancel, HP Vertica assumes your UDF does not need to perform any special cancel processing, and calls the function class's `destroy` function to have it free any resources (see UDF Resource Use).

- Your `cancel` function is called from a different thread than the thread running your UDF's `processPartition` function.
- The call to the `cancel` function is not synchronized in any way with your UDF's `processPartition` function. If you need your `processPartition` function to exit before your `cancel` function performs some action (killing threads, for example) you need to have the two function synchronize their actions.
- If your `cancel` function runs for too long, HP Vertica kills the side process running your function, if it is running in Fenced Mode.

UDF Debugging Tips

You must thoroughly debug your UDF before deploying it to a production environment. The following tips can help you get your UDF is ready for deployment.

Use a Single Node For Initial Debugging

You can attach to the HP Vertica process using a debugger such as gdb to debug your UDF code. Doing this in a multi-node environment, however, is very difficult. Therefore, consider setting up a single-node HP Vertica test environment to initially debug your UDF.

Write Messages to the HP Vertica Log

You can write to log files using the `ServerInterface.log` function. Every function in your UDF receives an instance of the ServerInterface object, so you can call the log function from anywhere in your UDF. The function acts similarly to printf, taking a formatted string, and an optional set of values and writing the string to a log file. Where the message is written depends on whether your function runs in fenced mode or unfenced mode:

- Functions running in unfenced mode write their messages into the `vertica.log` file in the catalog directory.
- Functions running in fenced mode write their messages into a log file named `UDxLogs/UDxFencedProcesses.log` in the catalog directory.

To help identify your function's output, HP Vertica adds the SQL function name bound to your UDF function (see Deploying and Using UDSFs for an explanation) to the log message.

The following code fragment shows how you can add a call to `srvInterface.log` in the Add2ints example code's `processBlock` function to log its input values:

```
const vint a = arg_reader.getIntRef(0);
const vint b = arg_reader.getIntRef(1);
srvInterface.log("got a: %d and b: %d", (int) a, (int) b);
```

This code generates an entries in the log file for each row the UDF processes. For example:

```
11-05-06 14:37:20.838 nameless:0x3f3a210 [UserMessage] <UDx> Add2ints - got a: 1 and b: 2
11-05-06 14:37:20.838 nameless:0x3f3a210 [UserMessage] <UDx> Add2ints - got a: 2 and b: 2
11-05-06 14:37:20.838 nameless:0x3f3a210 [UserMessage] <UDx> Add2ints - got a: 3 and b: 2
11-05-06 14:37:20.838 nameless:0x3f3a210 [UserMessage] <UDx> Add2ints - got a: 1 and b: 4
11-05-06 14:37:20.838 nameless:0x3f3a210 [UserMessage] <UDx> Add2ints - got a: 5 and b: 2
```

See Monitoring the Log Files in the Administrator's Guide for details on viewing the HP Vertica log files.

Adding Metadata to C++ Libraries

You can add metadata, such as author name, the version of the library, a description of your library, and so on to your library. This metadata lets you track the version of your function that is deployed on an Vertica Analytics Platform cluster and lets third-party users of your function know who created the function. Your library's metadata appears in the USER_LIBRARIES system table after your library has been loaded into the Vertica Analytics Platform catalog.

You declare the metadata for your library by calling the `RegisterLibrary` function in one of the source files for your UDx. If there is more than one function call in the source files for your UDx, whichever gets interpreted last as Vertica Analytics Platform loads the library is used to determine the library's metadata.

The `RegisterLibrary` function takes eight string parameters:

```
RegisterLibrary(author, library_build_tag, library_version, library_sdk_version,
                source_url, description, licenses_required, signature);
```

- `author` contains whatever name you want associated with the creation of the library (your own name or your company's name for example).
- `library_build_tag` is a string you want to use to represent the specific build of the library (for example, the SVN revision number or a timestamp of when the library was compiled). This is useful for tracking instances of your library as you are developing them.
- `library_version` is the version of your library. You can use whatever numbering or naming scheme you want.
- `library_sdk_version` is the version of the Vertica Analytics Platform SDK Library for which you've compiled the library.

  Note: This field isn't used to determine whether a library is compatible with a version of the Vertica Analytics Platform server. The version of the Vertica Analytics Platform SDK you use to compile your library is embedded in the library when you compile it. It is this information that Vertica Analytics Platform server uses to determine if your library is compatible with it.

- `source_url` is a URL where users of your function can find more information about it. This can be your company's website, the GitHub page hosting your library's source code, or whatever site you like.
- `description` is a concise description of your library.
- `licenses_required` is a placeholder for licensing information. In this release of Vertica Analytics Platform, you must leave this field as an empty string.
- `signature` is a placeholder for a signature that will authenticate your library. In this release of Vertica Analytics Platform, you must leave this field as an empty string.

For example, the following code demonstrates adding metadata to the Add2Ints example (see Subclassing ScalarFunction and Subclassing ScalarFunctionFactory

```
// Include the top-level Vertica SDK file
#include "Vertica.h"
// Using the Vertica namespace means we don't have to prefix all
// class references with Vertica::
using namespace Vertica;
/*
* ScalarFunction implementation for a UDSF that adds
* two numbers together.
*/

class Add2Ints : public ScalarFunction
{
public:
  /*
   * This function does all of the actual processing for the UDF.
   * In this case, it simply reads two integer values and returns
   * their sum.
   *
   * The inputs are retrieved via arg_reader
   * The outputs are returned via arg_writer
   */
    virtual void processBlock(ServerInterface &srvInterface,
                            BlockReader &arg_reader,
                            BlockWriter &res_writer)
    {
    // While we have input to processdo
      {
        // Read the two integer input parameters by calling the
        // BlockReader.getIntRef class functionconst
        vint a = arg_reader.getIntRef(0);
        const vint b = arg_reader.getIntRef(1);
        // Call BlockWriter.setInt to store the output value, which is the
        //  two input values added together
        res_writer.setInt(a+b);
        // Finish writing the row, and advance to the next output row
        res_writer.next();
        // Continue looping until there are no more input rows
      }
    while (arg_reader.next());
  }
};

/*
* This class provides metadata about the ScalarFunction class, and
* also instantiates a member of that class when needed.
*/
class Add2IntsFactory : public ScalarFunctionFactory
{
    // return an instance of Add2Ints to perform the actual addition.
    virtual ScalarFunction *createScalarFunction(ServerInterface &interface)
    {
        // Calls the vt_createFuncObj to create the new Add2Ints class instance.
        return vt_createFuncObj(interface.allocator, Add2Ints);
    }

    // This function returns the description of the input and outputs of the
    // Add2Ints class's processBlock function.  It stores this information in
    // two ColumnTypes objects, one for the input parameters, and one for
    // the return value.
    virtual void getPrototype(ServerInterface &interface,
                              ColumnTypes &argTypes,
                              ColumnTypes &returnType)
    {
        // Takes two ints as inputs, so add ints to the argTypes object
        argTypes.addInt();
        argTypes.addInt();
        // returns a single int, so add a single int to the returnType object.
        // Note that ScalarFunctions *always* return a single value.
        returnType.addInt();
    }
};

// Register the factory with HP Vertica
RegisterFactory(Add2IntsFactory);

// Register the library's metadata.
RegisterLibrary("Whizzo Analytics Ltd.",
                "1234",
                "2.0",
                "7.0.0",
                "http://www.example.com/add2ints",
                "Add 2 Integer Library",
                "",
                "");
```

Loading the library and querying the USER_LIBRARIES system table shows the metadata supplied in the call to `RegisterLibrary`:

```
=> CREATE LIBRARY add2intslib AS '/home/dbadmin/add2ints.so';
CREATE LIBRARY
=> \x
Expanded display is on.
=> SELECT * FROM USER_LIBRARIES WHERE lib_name = 'add2intslib';
-[ RECORD 1 ]-----+----------------------------------------
schema_name       | public
lib_name          | add2intslib
```

```
lib_oid           | 45035996273869808
author            | Whizzo Analytics Ltd.
owner_id          | 45035996273704962
lib_file_name     | public_add2intslib_45035996273869808.so
md5_sum           | 732c9e145d447c8ac6e7304313d3b8a0
sdk_version       | v7.0.0-20131105
revision          | 125200
lib_build_tag     | 1234
lib_version       | 2.0
lib_sdk_version   | 7.0.0
source_url        | http://www.example.com/add2ints
description       | Add 2 Integer Library
licenses_required |
signature         |
```

Developing a User Defined Function in R

HP Vertica supports User Defined Functions written in the R programming language. R is a free, open source programming language used for statistical computing.

The topics in this section guide you through developing a User Defined Function in R.

User Defined Functions in R Notes and Considerations

- You must first install The R Language Pack for HP Vertica before creating R functions inside of HP Vertica.
- User Defined Functions developed in R always run in Fenced Mode in a process outside of the main HP Vertica process.
- You can create Scalar Functions and Transform Functions using the R language. Other UDx types are not supported with the R language.
- NULL values in HP Vertica are translated to R NA values when sent to the R function. R NA values are translated into HP Vertica null values when returned from the R function to HP Vertica.
- R supports different data types than those available in HP Vertica, so data types are mapped between the two systems. This table details the data-type mapping for R:

HP Vertica Data Type	**R Data Type**
Boolean	logical
Date/Time: DATE, DATETIME, SMALLDATETIME, TIME, TIMESTAMP, TIMESTAMPZ, TIMETZ	numeric
Approximate Numeric: DOUBLE PRECISION, FLOAT, REAL	numeric
Exact Numeric: BIGINT, DECIMAL, INT, NUMERIC, NUMBER, MONEY	numeric
BINARY, VARBINARY	character
CHAR, VARCHAR	character

Installing/Upgrading the R Language Pack for HP Vertica

To use R with HP Vertica, install the RPM (or Debian .deb) R language package that matches your server version. The R language pack includes the R runtime and associated libraries for interfacing with HP Vertica.

This topic details:

- HP Vertica R Language Pack Prerequisites
- Installing the HP Vertica R Language Pack
- Upgrading the the HP Vertica R Language Pack

Important: You **must** install the R Language Pack on **each node** in the cluster.

Note: When upgrading from HP Vertica 6.1.0/6.1.1 (which uses R 2.14) to 6.1.2 or later, see the upgrade instructions at the end of this topic if you have installed additional R packages. If you do not follow the upgrade instructions then your additional R packages may not work.

HP Vertica R Language Pack Prerequisites

The R Language Pack RPM requires libgfortran.so.1, which may not be installed by default on your system. Install the RPM that contains libgfortran.so.1. See the table below to determine how to install libgfortran.so.1:

Linux Version	**How to Install libgfortran**
RHEL 6 and CentOS 6	Install the compat-libgfortran-41 RPM with the command: `yum install compat-libgfortran-41`
RHEL 5 and CentOS 5	Install the libgfortran RPM with the command: `yum install libgfortran`
Other supported platforms that use yum, such as Suse11.	You can determine the RPM needed for libgfortran.so.1 with the command: `yum whatprovides /usr/lib64/libgfortran.so.1` Typical packages that include libgfortran.so.1 include: • libgfortran-41-<any_minor_version>.rpm • compat-libgfortran-41-<any_minor_version>.rpm; • gcc41-fortran.rpm

To Install the HP Vertica R Language Pack:

1. Download the R language package by going to the myVertica portal, clicking the downloads tab, and selecting the vertica-R-lang_-*version*.rpm (or .deb) file for your server version. The R language package version must match your server version to three decimal points. For example, if your server version is 6.1.1, then the R Language Pack version must also be 6.1.1.
2. Install the package as root or using sudo:

- RPM: rpm -Uvh vertica-R-lang_-*version*.rpm
- Debian: dpkg -i vertica-R-lang_-*version*.deb

The installer puts the R binary in `/opt/vertica/R`. The installer also adds the file `vertica-udx-R.conf` to `/etc/ld.so.conf.d/`. This file is removed if you uninstall the package.

To Upgrade the HP Vertica R Language Pack:

When upgrading from HP Vertica 6.1.0 or 6.1.1 (which uses R 2.14) to 6.1.2 or later (which uses R 3.0), any R packages that you have manually installed may not work with R 3.0 and may have to be reinstalled. If you don't update your package(s), then R returns an error if the package cannot be used in R 3.0: *Error: package '[package name]' was built before R 3.0.0: please re-install it*. Instructions for upgrading these packages are below.

Note: The R packages provided in the R Language Pack are automatically upgraded and do not need to be reinstalled.

1. You must uninstall the R_lang RPM before upgrading the server RPM. Any additional R packages that you manually installed remain in `/opt/vertica/R` and are not removed when you uninstall the RPM.
2. Upgrade your server RPM as detailed in Upgrading HP Vertica to a New Version.
3. After the server RPM has been updated, install the new R Language Pack on each host as detailed above in *To install the HP Vertica R Language Pack*.
4. If you have installed additional R packages, on each node:
   a. as root run `/opt/vertica/R/bin/R` and issue the command: `update.packages(checkBuilt=TRUE)`
   b. Select a CRAN mirror from the list displayed.
   c. You are prompted to update each package that has an update available for it. Do **NOT** update:
      - Rcpp
      - Rinside

      You can optionally update any other packages installed with the R Language Pack. You must update any packages that you manually installed and are not compatible with R 3.0.

      The packages you selected to be updated are installed. Quit R with the command: `quit()`

Note: HP Vertica UDx functions written in R do not need to be compiled and you do not need to reload your HP Vertica-R libraries and functions after an upgrade.

R Packages

The HP Vertica R Language Pack includes the following R packages in addition to the default packages bundled with R:

- Rcpp
- RInside
- lpSolve
- lpSolveAPI

You can install additional R packages not included in the HP Vertica R Language Pack by using one of two methods. You **must** install the same packages on **all** nodes.

1. By using R Language Pack R binary at the command line and using the install.packages() R command. For example:

   ```
   # /opt/vertica/R/bin/R ...
   > install.packages("Zelig");
   ```

2. By running the following command:

   ```
   /opt/vertica/R/bin/R CMD INSTALL <path-to-package-tgz>
   ```

The install places the packages in: `/opt/vertica/R/library`.

Using the HP Vertica SDK R Examples

HP Vertica provides example R functions in /opt/vertica/sdk/examples/RFunctions/RFunctions.R

You can load the examples into HP Vertica with the command:

```
CREATE LIBRARY rLib AS '/opt/vertica/sdk/examples/RFunctions/RFunctions.R' LANGUAGE 'R';
```

You can then load the example functions included in the library. For example:

```
CREATE FUNCTION mul AS LANGUAGE 'R' NAME 'mulFactory' LIBRARY rLib;
```

Creating R Functions

To create a user defined R function, you define:

- An R Factory Function.
- A main R function.
- Optionally, an *outtypecallback* function to define the type(s) and precision of the values returned to HP Vertica and a parametertype callback function to define parameter names and types.

About the R Factory Function

User Defined Functions in R require a single factory function. The factory function wraps all of the information required by HP Vertica to load the user-defined R function. The factory function allows you to define the following:

Parameter	Description
name	The name of the primary R function in this User Defined Function.
udxtype	The type of User Defined function. Can be either "scalar" or "transform".
intype	The type(s) of arguments accepted by the function. See the supported types below. **Note:** User Defined Scalar Functions (UDSFs) can have a maximum limit of 32 arguments. User Defined Scalar Functions (UDSFs) have no limitation on the number of arguments.
outtype	The type(s) of arguments returned by the function. See the supported types below.
outtypecallback	(optional) The callback function to call before sending the data back to HP Vertica. It defines the types and precision that the main function returns.
parametertypecallback	(optional) The callback function to send parameter types and names to HP Vertica.
volatility	Indicate whether the function returns the same output given the same input, can be one of three types.
strictness	Indicate whether the function always returns null when any of its input arguments is null.

Factory Function - Supported Data Types

The following data types are supported in the factory function:

- boolean
- int

- float
- real
- char
- varchar
- long varchar
- date
- datetime
- smalldatetime
- time
- timestamp
- timestamptz
- timetz
- numeric
- varbinary
- binary
- long varbinary

Example Factory Function

The following example is a factory function for a scalar function written in R named *mul* that takes two floats and returns a float. It also defines an outtypecallback function named *mulReturnType*.

```
mulFactory <- function()
{
       list(name=mul,udxtype=c("scalar"),intype=c("float","float"), outtype=c("float"), outtype
callback=mulReturnType)
}
```

About the Main R Function

The main function (which is defined in the name parameter of the factory function) is the first function called when HP Vertica runs your function. The main function takes as input exactly one data frame and returns a single data frame. The factory function converts the intype arguments into a data frame for the primary function. For example, intype=c("float","float") is converted by the factory function into a two-dimensional matrix. The primary function returns a data frame, and the data frame is converted into the type defined by the outtype parameter of the factory function.

The following main function corresponds to the factory function defined earlier. It takes a data frame as input (converted from two floats) and multiplies the two values in each row of the data frame, then returns a data frame, which is converted to a single column of floats and passed back to HP Vertica.

```
mul<- function(x)
{
       pr <- x[,1] * x[,2]
       pr
}
```

About the Outtypecallback Function

You can optionally create an outtypecallback function to define the type(s), length/precision, and scale of the data being returned to HP Vertica. You return up to a four-column matrix:

1. data type
2. length/precision
3. scale
4. name of the column in the output (optional)

If any of the columns are left blank (or the outtypecallback function is omitted entirely), then HP Vertica uses default values.

When creating the outtypecallback function, you define one row for each value returned, using the same order as the outtypes that were defined in the factory function.

For example, if your function returns a data frame with three columns, containing INT, VARCHAR (24) and VARCHAR(2) data types, you could define the outtype as:

```
ret[1,1] = "int"
ret[1,4] = "product"
ret[2,1] = "varchar"
ret[2,2] = "24"
ret[3,1] = "varchar"
ret[3,2] = "2"
```

Note: When specifying *long varchar* or *long varbinary* data types, include the space between the two words. For example:

```
ret[1,1] = "long varchar"
```

The following outtypecallback example mimics the defaults used by HP Vertica. If the outtypecallback function were omitted, then HP Vertica provides these settings as defaults.

```
mulReturnType <- function(x)
{
        ret = data.frame(datatype = rep(NA,1), length = rep(NA,1), scale = rep(NA,1), name = re
p(NA,1))
        ret[1,1] = "float"
        ret[1,4] = "product"
        ret
}
```

Deploying the Function into HP Vertica

- To deploy a Scalar Function, see Deploying and Using UDSFs.
- To deploy a Transform Function, see Deploying and Using User Defined Transforms.

Also see CREATE LIBRARY and CREATE FUNCTION in the SQL Reference Manual.

Example R Scalar Function

```
##########
# Example: Multiplication
# Filename: mul.R
##########
###
# @brief multiplies col1 and col2 of the input data frame.
###
mul<- function(x)
{
       pr <- x[,1] * x[,2]
       pr
}
mulFactory <- function()
{
       list(name=mul,udxtype=c("scalar"),intype=c("float","float"), outtype=c("float"), outtype
callback=mulReturnType)
}
mulReturnType <- function(x)
{
        ret = data.frame(datatype = rep(NA,1), length = rep(NA,1), scale = rep(NA,1), name = re
p(NA,1))
        ret[1,1] = "float"
        ret[1,4] = "Multiplied"
        ret
}
```

Example Usage:

```
=> select * from twocols;
 x | y
```

```
---+---
 1 | 1
 2 | 2
 3 | 3
 4 | 4
 5 | 5
 6 | 6
 7 | 7
 8 | 8
 9 | 9
(9 rows)
=> CREATE LIBRARY mulLib AS '/home/dbadmin/mul.R' LANGUAGE 'R';
CREATE LIBRARY
=> CREATE FUNCTION mul AS NAME 'mulFactory' LIBRARY mulLib;
=> select mul(x,y) from twocols;
 Multiplied
-----
   1
   4
   9
  16
  25
  36
  49
  64
  81
(9 rows)
```

Setting Null Input and Volatility Behavior for R Functions

Starting in version 6.1, HP Vertica supports defining volatility and null-input settings for functions written in R. Both settings aid in the performance of your R function:

- Volatility settings describe the behavior of the function to the HP Vertica optimizer. For example, if you have identical rows of input data and you know the function is immutable, then you can define the function as IMMUTABLE. This tells the HP Vertica optimizer that it can return a cached value for subsequent identical rows on which the function is called rather than having the function run on each identical row.
- Null input setting determine how to respond to rows that have null input. For example, you can choose to return null if any inputs are null rather than calling the function and having the function deal with a NULL input.

Volatility Settings

To indicate your function's volatility, set the volatility parameter of your R factory function to one of the following values:

Value	Description
VOLATILE	Repeated calls to the function with the same input parameters always result in different values. HP Vertica always calls volatile functions for each invocation.
IMMUTABLE	Calls to the function with the same input parameters always results in the same output.
STABLE	Repeated calls to the function with the same input *within the same statement* returns the same output. For example, a function that returns the current user name would be stable since the user cannot change within a statement, but could change between statements.
DEFAULT_ VOLATILITY	The default volatility. This is the same as VOLATILE.

If you do not define a volatility, then the function is considered to be VOLATILE.

The following example sets the volatility to STABLE in the mulFactory function:

```
mulFactory <- function(){
       list(name=mulwithparams,udxtype=c("scalar"),intype=c("float","float"),
       outtype=c("float"), volatility=c("stable"), parametertypecallback=mulparams  )
}
```

Null Input Behavior

To indicate how your function reacts to NULL input, set the strictness parameter of your R factory function to one of the following values:

Value	Description
CALLED_ON_NULL_INPUT	The function must be called, even if one or more input values are NULL.
RETURN_NULL_ON_ NULL_INPUT	The function always returns a NULL value if any of its inputs are NULL.
STRICT	A synonym for RETURN_NULL_ON_NULL_INPUT
DEFAULT_STRICTNESS	The default strictness setting. This is the same as CALLED_ ON_NULL_INPUT.

If you do not define a null input behavior, then the function is called on every row of data regardless of the presence of NULLS.

The following example sets the null input behavior to STRICT in the mulFactory function:

```
mulFactory <- function()
{
       list(name=mulwithparams,udxtype=c("scalar"),intype=c("float","float"),
       outtype=c("float"), strictness=c("strict"), parametertypecallback=mulparams  )
}
```

Using Parameters in R

Starting in version 6.1, HP Vertica supports using parameters in functions written in R. Parameters are passed to the R function with the **USING PARAMETERS** keyword followed by a series of key-value pairs. The function accesses parameters as the second argument it receives, which is a list of key-value pairs.

You also must specify a field in the factory function called `parametertypecallback`. This field points to the callback function that defines the parameters expected by the function. The callback function simply defines a four-column data frame:

1. data type
2. length/precision
3. scale
4. name of the column in the output (optional)

If any of the columns are left blank (or the outtypecallback function is omitted entirely), then HP Vertica uses default values.

Syntax

```
SELECT <function-name>(<argument> [, ...] [USING PARAMETERS <key>=<value> [, ...]])
       FROM foo;
SELECT <function-name>(<argument> [, ...] [USING PARAMETERS <key>=<value> [, ...]])
       OVER (...) FROM foo;
```

Example Usage

In version 6.0, if you created a kmeans function, you would have to hard code the value of K (the number of clusters in to which to group the data points) into your function. In version 6.1 and later you can, for example, specify a parameter for the value of K and assign a value to K when you call the function from within HP Vertica. For example:

```
SELECT kmeans(geonameid, latitude, longitude, country_code USING PARAMETERS k = 5)
OVER (ORDER BY country_code) FROM geotab_kmeans;
```

Your R function receives everything before 'USING PARAMETERS' as the first dataframe, and the parameters are contained in the second parameter, which is a list.

Complete R Example

```
mykmeansparams <- function(x, y)
{
    # error check and get the number of clusters to be formed
    # The y argument contains the parameters from the USING PARAMETERS clause in
    # your SQL function call.
    if(!is.null(y[['k']]))
        k=as.numeric(y[['k']])
    else
        stop("Expected parameter k")
    # get the number of columns in the input data frame
    cols = ncol(x)
    # run the kmeans algorithm
    cl <- kmeans(x[,2:cols-1], k)
    # get the cluster information from the result of above
    Result <- cl$cluster
    #return result to vertica
    Result <- data.frame(VCol=Result1)
    Result
}

# call back function to return parameter types
kmeansParameters <- function()
{
     params <- data.frame(datatype=rep(NA, 1), length=rep(NA,1), scale=rep(NA,1),
               name=rep(NA,1) )
     param[1,1] = "int"
     param[1,4] = "k"
     param
}


# Function that tells vertica the name of the actual R function, the parameter types and
# the return types
keansFactoryParams <- function()
{
    list(name=mykmeansparams,udxtype=c("transform"), intype=c("int","float","float","floa
t",
         "float","varchar"), outtype=c("int"),
         parametertypecallback=kmeansParameters, volatility=c("stable"),
         strict=c("called_on_null_input"))
}
```

Polymorphic Functions in R

Polymorphism in R functions allow you to easily modify how the function is used on different datasets. Polymorphic functions in R can accept any number and type of argument that the user supplies. HP Vertica does not check the number or types of argument that the user passes to the function - it just passes all of the arguments supplied by the user. It is up to your polymorphic-function's main function to examine the number and types of arguments it received and determine if it can handle them.

Note: : Transform functions written in R can have an unlimited number of arguments. Scalar functions written in R are limited to a maximum number of 32 arguments.

Polymorphic functions are more flexible than using multiple factory classes for your function (see User Defined Function Overloading), since your function can determine at run time if it can process the arguments rather than accepting specific sets of arguments. However, your polymorphic function needs to perform more work to determine whether it can process the arguments that it has been given.

Declare the Function As Polymorphic

Your polymorphic R function declares it accepts any number of arguments in its factory's function by specifying "any" as the argument to the intype parameter and optionally the outtype parameter. If you define "any" argument for intype or outtype, then it is the only type that your function can declare for the respective parameter. You cannot define required arguments and then call "any" to declare the rest of the signature as optional. If your function has requirements for the arguments it accepts, your process function must enforce them.

For example, this is a factory function that declares that the function is polymorphic:

```
RFactory <- function()
{
     list(
               name=RFunction,
               udxtype=c("transform"),
               intype=c("any"),
               outtype=c("any"),
               outtypecallback=ReturnType
        )
}
```

Define the outtypecallback for Polymorphic Functions

The outtypecallback method is used to indicate the argument types and sizes it has been called with, and is expected to indicate the types and sizes that the function returns. The outtypecallback method can also be used to check for unsupported types and/or number of arguments. For example, the function may require only integers, with no more than 10 of them:

```
# This function returns the same number and types as was input
ReturnType <- function(x)
{
     ret <- NULL
     if(nrow(x) > 10)
     {
         stop("The function cannot accept more than 10 arguments")
     }
     for( i in nrow(x))
     {
          if(x[i,2] != "int")
               {
                    stop("The function only accepts integers")
               }
         rbind(ret,x{i,]) -> ret
     }
     ret
}
```

Complete Example

The examples below uses the popular Iris Flower Dataset to demonstrate how the R k-means algorithm clusters the data and how you can use the polymorphic properties of the function to run kmeans against different columns.

```
#The k-means ploymorphic algorithm

# Input: A dataframe with 5 columns
# Output: A dataframe with one column that tells the cluster to which each data
#          point belongs

mykmeansPoly <- function(x,y)
{
         # get the number of clusters to be formed
        if(!is.null(y[['k']]))
            k=as.numeric(y[['k']])
        else
           stop("Expected parameter k")

        # get the number of columns in the input data frame
    cols = ncol(x)
        # run the kmeans algorithm
        cl <- kmeans(x[,2:cols-1], k)
    # get the cluster information from the result of above
        Result <- cl$cluster
    #return result to vertica
        Result <- data.frame(VCol=Result)
        Result
}

# Function that tells vertica the name of the actual R function, and the
# polymorphic parameters
kmeansFactoryPoly <- function()
{
    list(name=mykmeansPoly,udxtype=c("transform"), intype=c("any"), outtype=c("int"),
         parametertypecallback=kmeansParameters)
}

# call back function to return parameter types
kmeansParameters <- function()
{
     params <- data.frame(datatype=rep(NA, 1), length=rep(NA,1), scale=rep(NA,1),
                         name=rep(NA,1) )
     params[1,1] = "int"
     params[1,4] = "k"
     params
}
```

To Use the Example

1. Create a table to hold the Iris data:

   ```
   create table iris (sl FLOAT, sw FLOAT, pl FLOAT, pw FLOAT, spec VARCHAR(32));
   ```

2. Create a CSV version of the Iris Flower Dataset and copy it into HP Vertica:

   ```
   copy iris from '/home/dbadmin/iris.csv' DELIMITER ','
   EXCEPTIONS '/home/dbadmin/iris_exceptions.txt';
   ```

3. Copy and paste the example into a file named kmean.r in the dbadmin home directory. Using vsql, create the library:

   ```
   create library rlib2 as '/home/dbadmin/kmean.r' language 'R';
   ```

4. Create the function:

   ```
   create transform function mykmeansPoly as language 'R' name 'kmeansFactoryPoly'
    library rlib2;
   ```

5. Run the function:

   ```
   select spec, mykmeansPoly(sl,sw,pl,pw,spec USING PARAMETERS k = 3) over(partition by
   spec) as kmean from iris;
   ```

6. Run the function with different number of arguments:

   ```
   select spec, mykmeansPoly(sl,pl,spec USING PARAMETERS k = 3) over(partition by spec)
   as kmean from iris;
   ```

7. To clean up the example library, function, and table:

   ```
   drop transform function mykmeansPoly();drop library rlib2;
   drop table iris;
   ```

Adding Metadata to R Libraries

You can add metadata, such as author name, the version of the library, a description of your library, and so on to your library. This metadata lets you track the version of your function that is deployed on an Vertica Analytics Platform cluster and lets third-party users of your function know who created the function. Your library's metadata appears in the USER_LIBRARIES system table after your library has been loaded into the Vertica Analytics Platform catalog.

You declare the metadata for your library by calling the `RegisterLibrary` function in one of the source files for your UDx. If there is more than one function call in the source files for your UDx, whichever gets interpreted last as Vertica Analytics Platform loads the library is used to determine the library's metadata.

The `RegisterLibrary` function takes eight string parameters:

```
RegisterLibrary(author, library_build_tag, library_version, library_sdk_version,
                source_url, description, licenses_required, signature);
```

- `author` contains whatever name you want associated with the creation of the library (your own name or your company's name for example).
- `library_build_tag` is a string you want to use to represent the specific build of the library (for example, the SVN revision number or a timestamp of when the library was compiled). This is useful for tracking instances of your library as you are developing them.
- `library_version` is the version of your library. You can use whatever numbering or naming scheme you want.
- `library_sdk_version` is the version of the Vertica Analytics Platform SDK Library for which you've compiled the library.

  Note: This field isn't used to determine whether a library is compatible with a version of the Vertica Analytics Platform server. The version of the Vertica Analytics Platform SDK you use to compile your library is embedded in the library when you compile it. It is this information that Vertica Analytics Platform server uses to determine if your library is compatible with it.

- `source_url` is a URL where users of your function can find more information about it. This can be your company's website, the GitHub page hosting your library's source code, or whatever site you like.
- `description` is a concise description of your library.
- `licenses_required` is a placeholder for licensing information. In this release of Vertica Analytics Platform, you must leave this field as an empty string.
- `signature` is a placeholder for a signature that will authenticate your library. In this release of Vertica Analytics Platform, you must leave this field as an empty string.

For example, the following code demonstrates adding metadata to the R mul example (Example R Scalar Function) Subclassing ScalarFunctionFactory

```
########### Example: Multiplication
# Filename: mul.R
##########
###
# @brief multiplies col1 and col2 of the input data frame.
###
mul<- function(x)
{
       pr <- x[,1] * x[,2]
       pr
}
mulFactory <- function()
{
       list(name=mul,udxtype=c("scalar"),intype=c("float","float"), outtype=c("float"), outtype
callback=mulReturnType)
}
mulReturnType <- function(x)
{
        ret = data.frame(datatype = rep(NA,1), length = rep(NA,1), scale = rep(NA,1), name = re
p(NA,1))
        ret[1,1] = "float"
        ret[1,4] = "Multiplied"
        ret
}

# Register the library's metadata.
RegisterLibrary("Whizzo Analytics Ltd.",
                "1234",
                "1.0",
                "7.0.0",
                "http://www.example.com/mul.R",
                "Multiplier R Library",
                "",
                "");
```

Loading the library and querying the USER_LIBRARIES system table shows the metadata supplied in the call to `RegisterLibrary`:

```
dbadmin=> CREATE LIBRARY rLib AS '/home/dbadmin/mul.R' LANGUAGE 'R';
CREATE LIBRARY


dbadmin=> select * from user_libraries where lib_name = 'rLib';
-[ RECORD 1 ]-----+--------------------------------
schema_name       | public
lib_name          | rLib
lib_oid           | 45035996453516356
author            | Whizzo Analytics Ltd.
owner_id          | 45035996273704962
lib_file_name     | public_rLib_45035996453516356.R
md5_sum           | 72548af5510fc43db9c5e187931a1835
sdk_version       |
revision          |
lib_build_tag     | 1234
lib_version       | 1.0
lib_sdk_version   | 7.0.0
source_url        | http://www.example.com/mul.R
description       | Multiplier R Library
licenses_required |
signature         |
```

Developing User Defined Functions in Java

User-Defined Functions (UDFs) are functions contained in external shared libraries that you develop in supported programming languages and load into HP Vertica using the CREATE LIBRARY and CREATE FUNCTION statements. They are best suited for analytic operations that are difficult to perform in SQL, and need to be performed frequently enough that their speed is a major concern.

You can develop UDFs in the Java programming language using the HP Vertica Java SDK.

In order to create a UDF in Java, you must create two subclasses:

- A function class that carries out the processing you want your UDF to perform.
- A factory class that provides metadata about the function class, and creates an instance of it to handle function calls.

UDFs written in Java always run in Fenced Mode, since the Java Virtual Machine that executes Java programs cannot run directly within the HP Vertica process.

The following sections explain how to develop a UDF using Java.

Supported Features

The HP Vertica Java SDK supports the following features:

- INTEGER, FLOAT, DATE, CHAR, VARCHAR, BINARY, VARBINARY, NUMERIC, LONG VARCHAR, LONG VARBINARY, and TIMESTAMP data types (see Java and HP Vertica Data Type Conversions).
- User Defined Scalar Functions (UDSFs) , User Defined Transform Functions (UDTFs), and User Defined Load (UDLs) (see Supported Java SDK Function Types).
- Overloaded and Polymorphic User Defined Functions which are capable of handing different sets of input values (see Accepting Different Numbers and Types of Arguments).
- UDx parameters (see

Supported Java SDK Function Types

There are several different types of UDFs which each have their own purpose. They are used in different types of SQL statements and process different sets of input and output values. The HP Vertica Java SDK defines two types of UDFs which are described below.

User Defined Scalar Functions (UDSFs)

UDSFs are the simplest form of UDF: they take between zero and thirty-two arguments (which is treated as a row of data) and return a single value. They must return a value for each row of data read in. They can be used in most places that HP Vertica's own built-in functions can be used. For example, one of the sample UDSFs, add2ints, accepts two INTEGER values as arguments and returns their sum. You can use it within a query like this:

```
=> SELECT * from T;
 a | b
---+---
 3 | 4
 5 | 6
 1 | 2
(3 rows)
=> SELECT a, b, add2ints(a,b) FROM T ORDER BY a ASC;
 a | b | add2ints
---+---+----------
 1 | 2 |        3
 3 | 4 |        7
 5 | 6 |       11
(3 rows)
```

See Developing a User Defined Scalar Function in Java for details of creating a UDSF using the Java SDK.

User Defined Transform Functions (UDTF)

UDTFs read zero or more arguments and optionally return one or more values. They let you create functions that transform a table of data into another table. The row of data it outputs does not have to correspond in any way to the row of data it reads in, and it can return as many or as few rows as it wants. You can only use UDTFs in a SELECT statement, which can only contain the UDTF function call and a required OVER clause. The SELECT statement can contain any subset of the PARTITION BY clause used in the OVER statement. For example, the sample tokenize UDTF takes a single VARCHAR string, breaks it into individual words, and returns each word in its own row. The following example demonstrates calling tokenize:

```
=> CREATE TABLE T (url varchar(30), description varchar(2000));
CREATE TABLE
=> INSERT INTO T VALUES ('www.amazon.com',
-> 'Online retail merchant and provider of cloud services');
 OUTPUT
--------
      1
(1 row)
=> INSERT INTO T VALUES ('www.hp.com',
-> 'Leading provider of computer hardware and imaging solutions');
 OUTPUT
--------
      1
(1 row)
=> INSERT INTO T VALUES ('www.vertica.com','World''s fastest analytic database');
 OUTPUT
--------
      1
(1 row)
dbadmin=> COMMIT;
COMMIT
```

```
=> SELECT url, tokenize(description) OVER (partition by url) FROM T;
       url       |  Tokens
-----------------+-----------
 www.hp.com      | Leading
 www.hp.com      | provider
 www.hp.com      | of
 www.hp.com      | computer
 www.hp.com      | hardware
 www.hp.com      | and
 www.hp.com      | imaging
 www.hp.com      | solutions
 www.vertica.com | World's
 www.vertica.com | fastest
 www.vertica.com | analytic
 www.vertica.com | database
 www.amazon.com  | Online
 www.amazon.com  | retail
 www.amazon.com  | merchant
 www.amazon.com  | and
 www.amazon.com  | provider
 www.amazon.com  | of
 www.amazon.com  | cloud
 www.amazon.com  | services
(20 rows)
```

See Developing a User Defined Transform Function in Java for details on developing a UDSF using the HP Vertica Java SDK.

The Vertica Analytics Platform Java SDK also supports developing User Defined Load (UDL) functions. See Developing UDLs in Java.

Java UDF Resource Management

Java Virtual Machines (JVMs) allocate a set amount of memory when they start. This set memory allocation complicates memory management for Java UDFs, since (unlike UDFs developed in C++) memory cannot be dynamically allocated and freed by the UDF as it is processing data.

To control the amount of memory consumed by Java UDFs, HP Vertica has a memory pool named jvm that HP Vertica uses to allocate memory for Java UDF JVMs. If this memory pool is exhausted, queries that call Java UDFs block until enough memory in the pool becomes free to start a new JVM.

By default, the jvm pool has:

- no memory of its own assigned to it, so it borrows memory from the GENERAL pool.
- its MAXMEMORYSIZE set to either 10% of system memory or 2GB, whichever is smaller.
- its PLANNEDCONCURRENCY set to AUTO, so that it inherits the GENERAL pool's PLANNEDCONCURRENCY setting.

When a SQL statement calls a Java UDF, HP Vertica checks the jvm memory pool to determine if there is enough memory in it to start a new JVM instance to handle the function call. HP Vertica starts each new JVM with its heap memory size set to approximately the jvm pool's MAXMEMORYSIZE parameter divided by its PLANNEDCONCURRENCY parameter.

If your Java UDF attempts to consume more memory than has been allocated to the JVM's heap size, it exits with a memory error. You can attempt to resolve this issue by:

- increasing the jvm pool's MAXMEMORYSIZE parameter.
- decreasing the jvm pool's PLANNEDCONCURRENCY parameter.
- changing your Java UDF's code to consume less memory.

See Managing Workloads in the Administrator's Guide for details on how to tune the parameters of the jvm and other resource pools.

Notes

- The jvm resource pool is only used to allocate memory for the Java UDF functionc alls in a statement. The rest of the resources required by the SQL statement come from other memory pools.
- The first time a Java UDF is called, HP Vertica starts a JVM to execute some Java methods to get metadata about the UDF during the query planning phase. The memory for this JVM is also taken from the jvm memory pool.

Installing Java on HP Vertica Hosts

You must install a Java Virtual Machine (JVM) on every host in your cluster in order for HP Vertica to be able to execute your Java UDFs.

Installing Java on your HP Vertica cluster is a two-step process:

1. Download and install the Java installation package on all of the hosts in your cluster.
2. Set the JavaBinaryForUDx configuration parameter to tell HP Vertica the location of the Java executable.

Downloading and Installing the Java Installation Package

For Java-based features, HP Vertica requires a 64-bit Java Standard Edition 6 (Java version 1.6) or later runtime from Oracle. The OpenJDK environment is not supported. You can choose to install either the Java Runtime Environment (JRE) or Java Development Kit (JDK), since the JDK also includes the JRE. See the Java Standard Edition (SE) Download Page to download an installation package for your Linux platform. You usually run the installation package as root in order to install it. See the download page for instructions.

Once you have installed a JVM on each host, ensure that the `java` command is in the search path and calls the correct JVM by running the command:

```
$ java -version
```

This command should print something similar to:

```
java version "1.6.0_37"Java(TM) SE Runtime Environment (build 1.6.0_37-b06)
Java HotSpot(TM) 64-Bit Server VM (build 20.12-b01, mixed mode)
```

Note: Any previously installed Java VM (such as OpenJDK or an earlier unsupported version of Java) on your hosts may interfere with a newly installed Java runtime. See your Linux distribution's documentation for instructions on configuring which JVM is the default. Unless absolutely required, you should uninstall any incompatible version of Java before installing the Oracle Java 6 or Java 7 runtime.

Setting the JavaBinaryForUDx Configuration Parameter

The JavaBinaryForUDx configuration parameter tells HP Vertica where to look for the JRE to execute Java UDxs. After you have installed the JRE on all of the nodes in your cluster, set this parameter to the absolute path of the Java executable. You can use the symbolic link that some Java installers create (for example `/usr/bin/java`). If the Java executable is in your shell search path, you can get the path of the Java executable by running the following command from the Linux command line shell:

```
$ which java
/usr/bin/java
```

If the `java` command is not in the shell search path, use the path to the Java executable in the directory where you installed the JRE. Suppose you installed the JRE in `/usr/java/default` (which is where the installation package supplied by Oracle installs the Java 1.6 JRE). In this case the Java executable is `/usr/java/default/bin/java`.

You set the configuration parameter by executing the following statement as a **database superuser**:

```
=> SELECT SET_CONFIG_PARAMETER('JavaBinaryForUDx','/usr/bin/java');
```

See SET_CONFIG_PARAMETER in the SQL Reference Manual for more information on setting configuration parameters.

To view the current setting of the configuration parameter, query the CONFIGURATION_PARAMETERS system table:

```
=> \x
Expanded display is on.
=> SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'JavaBinaryForUDx';
-[ RECORD 1 ]-----------------+----------------------------------------------------------
node_name                     | ALL
```

```
parameter_name                 | JavaBinaryForUDx
current_value                  | /usr/bin/java
default_value                  |
change_under_support_guidance  | f
change_requires_restart        | f
description                    | Path to the java binary for executing UDx written in Java
```

Once you have set the configuration parameter, HP Vertica can find the Java executable on each node in your cluster.

Note: Since the location of the Java executable is set by a single configuration parameter for the entire cluster, you must ensure that the Java executable is installed in the same path on all of the hosts in the cluster.

Configuring Your Java Development Environment

Before you start developing your UDF in Java, you need to configure your development environment. You can choose to develop your Java UDF on a node in a development HP Vertica database (not in a production environment) or on a desktop system.

Install the Java Development Kit (JDK) version on your development system that matches the Java version you have installed on your database hosts (see Installing Java on HP Vertica Hosts).

You also need two files from the Java support package:

- `/opt/vertica/bin/VerticaSDK.jar` contains the HP Vertica Java SDK and other supporting files.
- `/opt/vertica/sdk/com/vertica/sdk/BuildInfo.java` contains version information about the SDK. You must compile this file and include it within your Java UDF JAR files.

If you are not using a node in a development database as a development system, you can copy these files from one of the database nodes to your development system.

Compiling BuildInfo.java

You need to compile the `BuildInfo.java` into a class file, so you can include it in your Java UDF JAR library. If you are using an HP Vertica node as a development system, you can either copy the `BuildInfo.java` file to another location on your host, or compile it in place if you have root privileges. Only root has privileges to write files to the `/opt/vertica/sdk` directory.

To compile the file into a class file, use the command:

```
$ javac -classpath /opt/vertica/bin/VerticaSDK.jar \
      /path/BuildInfo.java
```

replacing *path* with the path to the file. If you want to compile it in place, use the command:

```
$ sudo javac -classpath /opt/vertica/bin/VerticaSDK.jar \
        /opt/vertica/sdk/com/vertica/sdk/BuildInfo.java
```

If you want to develop your Java UDFs on a system other than one of your database hosts, you must copy BuildInfo.java to your development system and compile it.

Note: If you use an IDE such as Eclipse, you can include the `BuildInfo.java` file in your project, and add the `VerticaSDK.jar` file to the project's build path. See your IDE's documentation for details on how to include files and libraries in your projects.

The HP Vertica Java SDK Documentation

You can find detailed documentation of all of the classes in the HP Vertica Java SDK in the SDK documentation. This documentation is included in the same package containing this document.

Java and HP Vertica Data Type Conversions

The HP Vertica Java SDK converts HP Vertica's native data types into the appropriate Java data type. The following table lists the HP Vertica data types and their corresponding Java data types.

HP Vertica Data Type	Java Data Type
INTEGER	long
FLOAT	double
NUMERIC	com.vertica.sdk.VNumeric
DATE	java.sql.Date
CHAR, VARCHAR, LONG VARCHAR	com.vertica.sdk.VString
BINARY, VARBINARY, LONG VARBINARY	com.vertica.sdk.VString
TIMESTAMP	java.sql.Timestamp

Note: Some Vertica Analytics Platformdata types are not supported.

Testing for Null Values

You can test whether a value being read from HP Vertica is NULL by using data-type-specific methods (such as `isLongNull`, `isDoubleNull`, and `isBooleanNull`) on the `BlockReader` or `PartitionReader` object that your UDF uses to read data from HP Vertica :

```
// See if the Long value in column 0 is a NULL
if (inputReader.isLongNull(0)) {
    // value is null
    . . .
```

Developing a User Defined Scalar Function in Java

A UDSF function returns a single value for each row of data it reads. It can be used anywhere a built-in HP Vertica function can be used. You usually develop a UDF to perform data manipulations that are too complex or too slow to perform using SQL statements and functions. UDFs also let you use analytic functions provided by third-party libraries within HP Vertica while still maintaining high performance.

To create your UDSF, you create subclasses of two classes defined by the HP Vertica Java SDK:

- The `ScalarFunction` class, which carries out whatever processing you want your UDSF to perform.
- The `ScalarFunctionFactory` class which defines the metadata about your UDSF such as its arguments and return type, and creates an instance of your `ScalarFunction` subclass.

Developing a UDSF in Java can be broken down into three steps:

- Subclassing the ScalarFunction Class to define your UDSF's data processing.
- Defining the Arguments and Return Type for Your UDSF by overriding the `ScalarFunctionFactory.getPrototype` method.
- Overriding createScalarFunction to create an instance of your `ScalarFunction` subclass.

The topics in this section explain these steps.

Java UDSF Requirements

Your UDSF must meet several requirements:

- It must always return an output value for each input row. Not doing so can result in incorrect query results or other issues.
- It can expect between zero and thirty-two arguments.
- If it depends on third party libraries, you must either include them in the JAR file along with your UDSF's classes or you must install them on each host in your cluster in a directory that's in your CLASSPATH environment variable.

Subclassing the ScalarFunction Class

Your subclass of `ScalarFunction` is where you define your UDSF's data processing. You have two choices of where to define your subclass of `ScalarFunction`:

- Define it as an inner class of your `ScalarFunctionFactory` subclass (which means placing its source code within the code of your `ScalarFunctionFactory` class). This is the simplest method, since you do not need to manage an additional source file to contain your `ScalarFunction` class. Java allows just one top-level public class per source file, so `ScalarFunction` must be an inner class of `ScalarFunctionFactory` if you want them to share a source file.

  Defining your `ScalarFunction` subclass as an inner class can become cumbersome if you have a lot of code or you have broken the processing performed by your UDSF into multiple subclasses.

- Define it in its own source file. This requires a bit more management, but is the best solution of your UDSF code is complex. You must define it in its own source file if you want to use your `ScalarFunction` class with multiple `ScalarFunctionFactory` classes to accept multiple function signatures (see Accepting Different Numbers and Types of Arguments).

Your subclass of `ScalarFunction` must at least override the `processBlock` method, which performs the actual processing for your UDSF. It reads a row of arguments, performs some operation on those arguments, and then outputs a value. It repeats this process until it has read every row of input.

The parameters passed to this method are:

- An instance of the `ServerInterface` class which provides some utility methods to interact with the HP Vertica server. See Communicating with HP Vertica Using ServerInterface for more information.
- An instance of the `BlockReader` class that your method uses to read data.
- An instance of the `BlockWriter` class that your method uses to write its output.

Your `processBlock` method reads its input arguments from the `BlockReader` instance by using data-type-specific getters, such as `getLong` and `getString`. It then performs whatever processing is required on the data to get the result. When your `processBlock` method has finished processing the input, it writes its return value by calling a data type specific method on the `BlockWriter` object, then advances to the next row of output by calling `BlockWriter.next`.

Once it has finished processing the first row of data, your `processBlock` method must call `BlockReader.next` to advance to the next row of input until it returns false, indicating that there are no more input rows to be processed.

The following example code is for a `ScalarFunction` subclass named `Add2ints` that reads two integer values, adds them together, and returns their sum.

```
    public class Add2ints extends ScalarFunction
    {
        @Override
        public void processBlock(ServerInterface srvInterface,
                                 BlockReader argReader,
                                 BlockWriter resWriter)
```

```
                      throws UdfException, DestroyInvocation
        {
            do {
                // The input and output objects have already loaded
                // the first row, so you can start reading and writing
                // values immediately.

                // Get the two integer arguments from the BlockReader
                long a = argReader.getLong(0);
                long b = argReader.getLong(1);

                // Process the arguments and come up with a result. For this
                // example, just add the two arguments together.
                long result = a+b;

                // Write the integer output value.
                resWriter.setLong(result);

                // Advance the output BlocKWriter to the next row.
                resWriter.next();

                // Continue processing input rows until there are no more.
            } while (argReader.next());
        }
    }
```

Notes

- Your implementation of `processBlock` cannot assume it is called from the same thread that instantiated the `ScalarFunction` object.
- The same instance of your `ScalarFunction` subclass can be called on to process multiple blocks of data.
- The rows of input sent to your `processBlock` method are not guaranteed to be any particular order.

Defining the Arguments and Return Type for Your UDSF

You define the inputs and outputs used in your User Defined Scalar Function (UDSF) in your subclass of the `ScalarFunctionFactory` class. You must override one or two members of this class, depending on the data types of the arguments and the return type:

- `getPrototype` defines the data types of the input columns and output value. You must always override this method in your `ScalarFunctionFactory` subclass.
- `getReturnType` defines the width of the output value if its data type is variable-width, or the precision of output value if its data type requires precision. It can also optionally name the output column. You are only required to override this method if your UDSF returns a data type that requires precision or has a variable width. If it does not, you may still choose to override this method in order to name your output column.

Overriding getPrototype

HP Vertica calls the `ScalarFunctionFactory.getPrototype` method to get the number and data types of its arguments, and the data type of its return value. This method gets three arguments: an instance of the `ServerInterface` class (see Communicating with HP Vertica Using ServerInterface), and two instances of the `ColumnTypes` class. You use the first instance to set the arguments your function expects and the second to set the data type of your function's return value.

The `ColumnTypes` class has a set of methods that start with the word "add" (such as `addInt` and `addFloat`) that set the data type for an argument. On the first instance of `ColumnTypes` passed to your `getPrototype` method, you call one of these methods to add each argument that your UDSF expects. Since UDSFs only return one value, you call a single method on the second instance of `ColumnTypes` to set the output data type.

The following example code demonstrates creating a subclass of `ScalarFunctionFactory` named `Add2intsFactory` which overrides the `getPrototype` method so the UDSF accepts two integer values as arguments and returns an integer value.

```
// You will need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.example;
// Import all of the Vertica SDK
import com.vertica.sdk.*;
public class Add2intsFactory extends ScalarFunctionFactory
{
    @Override
    public void getPrototype(ServerInterface srvInterface,
                             ColumnTypes argTypes,
                             ColumnTypes returnType)
    {
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }
```

Setting Precision, Width, and Name of the Output Value in getReturnType

The `getReturnType` method lets you set additional information about your UDSF's output argument. You must override this method if your UDSF returns a value that is variable-width (VARCHAR or BINARY, for example) or requires precision.

The `getReturnType` method gets three parameters: an instance of `ServerInterface` (see Communicating with HP Vertica Using ServerInterface), and two instances of `SizedColumnTypes`. These instances describes the width and precision of columns. The first instance is pre-populated with the width and precision information of the arguments to your function. You can use this object to get the precision, width, or column name of any of the arguments. Your override of `getReturnType` calls a method on the second `SizedColumnTypes` instance to set the precision, width, and optionally a column name of your UDSF's return value.

Note: When developing your `getReturnType` method, you must ensure that the column width or precision you set in the `SizedColumnTypes` object match the output value type you defined in the `getProtype` method. For example, setting width on a fixed-width value can generate errors.

The following example code is from a UDSF that takes two VARCHAR arguments as input, and returns the shorter of the two strings as an output value. Since the return value is a VARCHAR, it has to override the `getReturnType` method to report the width of the value it will output. In this case, it will return the shorter of the two strings, so it sets the output value's width to the size of the shorter input string.

```
// Factory class for UDSF that returns the shorter of two strings.
public class ShorterStringFactory extends ScalarFunctionFactory
{
    @Override
       public void getPrototype(ServerInterface srvInterface,
                             ColumnTypes argTypes,
                             ColumnTypes returnType)
    {
        // Accepts two VARCHAR arguments and returns two.
        argTypes.addVarchar();
        argTypes.addVarchar();
        returnType.addVarchar();
    }
    @Override
    // Report the width of the output value. This is equal to
    // the length of the shorter of the arguments. This method can
    // actually handle any number of input arguments. It compares the
    // length of the first argument string to the subsequent arguments,
    // and sets the output length to the shortest string found.
    public void getReturnType(ServerInterface srvInterface,
                              SizedColumnTypes argTypes,
                              SizedColumnTypes returnType)
    {
        // Get length of first argument string
        int len = argTypes.getColumnType(0).getStringLength();
        // Loop through remaining arguments, and see if any of their
        // lengths are shorter.
        for (int i = 1; i < argTypes.getColumnCount(); ++i) {
            int arg_len = argTypes.getColumnType(i).getStringLength();
            if (len > arg_len) len = arg_len;
        }
        // Set the width of the output to the width of the shortest string
        // found.
        returnType.addVarchar(len);
    }
    . . .
```

Overriding createScalarFunction

The last piece of your `ScalarFunctionFactory` class is to override the `createScalarFunction` method. HP Vertica calls this method to create an instance of your `ScalarFunction` subclass to process a block of data. All this method needs to do is return an instance of your `ScalarFunction` class. The following example demonstrates overriding the `createScalarFunction` to instantiate a member of the `Add2ints` class defined in Subclassing the ScalarFunction Class.

```
@Override
public ScalarFunction createScalarFunction(ServerInterface srvInterface)
{
    return new Add2ints();
}
```

Complete Java UDSF Example

The following code example is the full source of the Add2ints example UDSF described in Subclassing the ScalarFunction Class, Defining the Arguments and Return Type for Your UDSF, and Overriding createScalarFunction. To simplify handling the source code, the `ScalarFunction` class is defined as an inner class of the `ScalarFunctionFactory` class.

```
// You will need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.example;
// Import all of the Vertica SDK
import com.vertica.sdk.*;
public class Add2intsFactory extends ScalarFunctionFactory
{
    @Override
    public void getPrototype(ServerInterface srvInterface,
                             ColumnTypes argTypes,
                             ColumnTypes returnType)
    {
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }

   // This ScalarFunction is defined as an inner class of
   // its ScalarFunctionFactory class. This gets around having
   // to have a separate source file for this public class.

    public class Add2ints extends ScalarFunction
    {
        @Override
        public void processBlock(ServerInterface srvInterface,
                                 BlockReader argReader,
                                 BlockWriter resWriter)
                    throws UdfException, DestroyInvocation
        {
            do {
                // The input and output objects have already loaded
                // the first row, so you can start reading and writing
                // values immediately.
                // Get the two integer arguments from the BlockReader
                long a = argReader.getLong(0);
                long b = argReader.getLong(1);

                // Process the arguments and come up with a result. For this
                // example, just add the two arguments together.
                long result = a+b;

                // Write the integer output value.
                resWriter.setLong(result);

                // Advance the output BlocKWriter to the next row.
                resWriter.next();

                // Continue processing input rows until there are no more.
            } while (argReader.next());
        }
    }


    @Override
    public ScalarFunction createScalarFunction(ServerInterface srvInterface)
    {
        return new Add2ints();
    }

}
```

Deploying and Using Your Java UDSF

Once you have finished developing the code for your Java UDSF, you need to deploy it:

1. Compile your code and package it into a JAR file. See Compiling and Packaging a Java UDF.
2. Copy your UDSF JAR file to a host in your database. You only need to copy it to a single host—HP Vertica automatically distributes the JAR file to the rest of the hosts in your database when you add the library to the database catalog.
3. Connect to the host as the database administrator.
4. Connect to the database using **vsql**.
5. Add your library to the database catalog using the CREATE LIBRARY statement.

```
=> CREATE LIBRARY libname AS '/path_to_jar_file/filename.jar'
-> LANGUAGE 'Java';
```

Where *libname* is the name you want to use to reference the library and *path_to_jar_file/filename*.jar is the fully-qualified pathname to the JAR file you copied to the host.

For example, if you created a JAR file named Add2intsLib.jar and copied it to the dbadmin account's home directory, you would use this command to load the library:

```
=> CREATE LIBRARY add2intslib AS '/home/dbadmin/Add2intsLib.jar'
-> LANGUAGE 'Java';
```

6. Define your UDSF in the catalog using the CREATE FUNCTION statement.

```
=> CREATE FUNCTION functionName AS LANGUAGE 'Java' NAME
-> 'namespace.factoryClassName' LIBRARY libname;
```

The *functionName* is the name you want to give your function. The *namespace.factoryClassName* is the fully-qualified name (namespace and class name) of your UDSF's factory class. The *libname* is the name you gave your Java UDF library in step 5.

For example, to define the Add2ints function shown in Complete Java UDSF Example, you would use the command:

```
=> CREATE FUNCTION add2ints AS LANGUAGE 'Java' NAME
-> 'com.mycompany.example.Add2intsFactory' LIBRARY add2intslib;
```

7. You can now call your Java UDSF the same way you call any other function:

```
=> select add2ints(123, 567);
 add2ints
----------
      690
(1 row)
=> SELECT * from T;
 a | b
---+---
 3 | 4
 5 | 6
 1 | 2
(3 rows)
=> SELECT a, b, add2ints(a,b) FROM T ORDER BY a ASC;
 a | b | add2ints
---+---+----------
 1 | 2 |        3
 3 | 4 |        7
 5 | 6 |       11
(3 rows)
```

Developing a User Defined Transform Function in Java

A User Defined Transform Function (UDTF) reads one or more arguments (treated as a row of data), and returns zero or more rows of data consisting of one or more columns. The schema of the output table does not need to correspond to the schema of the input table—they can be totally different. The UDTF can return any number of output rows for each row of input.

UDTFs can only be used in the SELECT list that contains just the UDTF call and a required OVER clause.

Unlike other types of User Defined Functions, UDTFs do not have a limit on the number of arguments that they can accept. Most other types of UDFs have a maximum of 32 arguments..

Note: Your UDTF must set a value for each column in any row of data it writes. Columns do not have any sort of default value. Writing a row of data with missing values can result in errors.

To create your UDTF, you create subclasses of two classes defined by the HP Vertica Java SDK:

- The `TransformFunction` class, which carries out the processing you want your UDTF to perform.
- The `TransformFunctionFactory` class, which provides HP Vertica with metadata about your UDTF such as its arguments and output columns, and also creates an instance of your `TransformFunction` subclass.

The topics in this section explain how to create your UDTF by subclassing these classes.

Subclassing the TransformFunction Class

Your subclass of `TransformFunction` is where you define your UDTF's data processing. You can define this subclass in one of two places:

- Define it as an inner class of your `TransformFunctionFactory` subclass (which means placing its source code within the code of your `TransformFunctionFactory` class). This is the simpler method, since you do not need to manage an additional source file to contain your `TransformFunction` class. Java only allows one top-level public class per source file, so you cannot define your `TransformFunction` in the same source file as your `TransformFunctionFactory` class unless it is an inner class. Defining your `TransformFunction` subclass as an inner class can become cumbersome if you have a lot of code, or have broken the processing performed by your UDTF into multiple subclasses.
- Define it in its own source file. This requires a bit more management, but is the best solution of your UDTF code is complex. You must define it in its own source file if you want to use your `TransformFunction` class with multiple `TransformFunctionFactory` classes to accept multiple function signatures (see Accepting Different Numbers and Types of Arguments).

Your subclass of `TransformFunction` must override the `processPartition` method, which performs the actual processing for your UDTF. HP Vertica sends this method a partition of data to be processed. The `processPartition` methods reads rows of data by calling methods on an instance of `PartitionReader`, performs some operation on the data, and optionally outputs one or more rows of output by calling methods on an instance of `PartitionWriter`.

The following example code is for a class that breaks the contents in a single VARCHAR column into individual words (substrings separated by one or more spaces). This class corresponds to the factory class described in Defining Your Java UDTF's Input and Output Table Columns.

```
public class TokenizeString extends TransformFunction
{
    @Override
    public void processPartition(ServerInterface srvInterface,
                                 PartitionReader inputReader,
                                 PartitionWriter outputWriter)
                throws UdfException, DestroyInvocation
    {
        // Loop over all rows passed in in this partition.
        do {
            // First test if the input string is NULL. If so, return NULL
            if (inputReader.isStringNull(0)) {
                outputWriter.setStringNull(0);
            } else {
                // Get the string value in column zero and break it into a
                // it into words. Output each word as its own
                // value.
                String[] tokens = inputReader.getString(0).split("\\s+");
                // Output each word on a separate row.
                for (int i = 0; i < tokens.length; i++)  {
                    // Output a string value in column 0 of the output
                    outputWriter.getStringWriter(0).copy(tokens[i]);
                    // Move to the next row of output
                    outputWriter.next();
                }
            }
        // Loop until there are no more input rows in partition.
        } while (inputReader.next());
    }
}
```

Defining Your Java UDTF's Input and Output Table Columns

You define the input and output of your User Defined Transform Function (UDTF) in your subclass of the `TransformFunctionFactory` class. You need to override one or two methods in this class, depending on data types of its input and output values:

- `getPrototype` defines the data types of the input and output columns. You must always override this method in your `TransformFunctionFactory` subclass.

- `getReturnType` defines the width of any variable-width output columns, the precision of data types that require precision, and optionally names the output columns. You are only required to override this method if your UDTF returns one or more columns that have a data type that requires precision or a variable width. If none of your columns are variable-width or require precision, you may still choose to override this method in order to name your output columns.

Note: Java UDx does not currently support data types that require precision.

Overriding getPrototype

The `TransformFunctionFactory.getPrototype` method gets three parameters as input: an instance of `ServerInterface` (see Communicating with HP Vertica Using ServerInterface), and two instances of `ColumnTypes`, one representing the columns in the input rows, and the other the columns in the output rows.

On both of these objects, you need to call data-type-specific methods (such as `addInt` and `addVarchar`) to define your UDTF's the input and output columns. The order in which you call these methods defines input and output table schemas. The following example code is from a UDTF named TokenFactory that parses a single VARCHAR column into individual words. it defines a single VARCHAR input and output column:

```
// Break a single string input into individual words (substrings delimited by
// one or more spaces).
public class TokenFactory extends TransformFunctionFactory
{
    // Set the number and data types of the columns in the input and output rows.
    @Override
    public void getPrototype(ServerInterface srvInterface,
                             ColumnTypes argTypes, ColumnTypes returnType)
    {
        // One column in the input row: a Varchar
        argTypes.addVarchar();
        // One column in the output row: a Varchar
        returnType.addVarchar();
    }
```

Overriding getReturnType

The `getReturnType` method gets three parameters: an instance of `ServerInterface` (see Communicating with HP Vertica Using ServerInterface), and two instances of `SizedColumnTypes`. These instances describes the width and precision of columns. The first instance is pre-populated with the width and precision information of the input rows. You can use this object to get the precision, width, or column name of any of the columns in the input row. Your override of `getReturnType` call methods on the second `SizedColumnTypes` instance to set the precision, width, and column name of your UDTF's output columns.

Note: When developing your `getReturnType` method, you must ensure that the column widths and precision you set in the `SizedColumnTypes` object match the columns you defined in the `getProtype` method. For example, setting width on a fixed-width column can generate errors.

Since the `getProtoype` method in the previous example defined a variable-width column (a VARCHAR), the `TokenFactory` class must override the `getReturnType` method to set the width of the output column. The following code example demonstrates setting the width of the single output column to the width of the input column, since the longest string the UDTF could return is the full input string (in the case where there are no spaces in the input string).

```
// Set the width of any variable-width output columns, and also name
// the columns.
@Override
public void getReturnType(ServerInterface srvInterface, SizedColumnTypes
                            inputTypes, SizedColumnTypes outputTypes)
{
    // Set the maximum width of the return column to the width
    // of the input column and name the output column "Tokens"
    outputTypes.addVarchar(
        inputTypes.getColumnType(0).getStringLength(), "Tokens");
}
```

Overriding the createTransformFunction Method

The last step in creating your `TransformFunctionFactory` class is to override the `createTransformFunction` method. HP Vertica calls this method to create an instance of your `TransformFunction` subclass to process a partition of data. All this method needs to do is return an instance of your `TransformFunction` class. The following example demonstrates overriding the `createTransformFunction` to instantiate a member of the `TokenizeString` class defined in Subclassing the TransformFunction Class.

```
@Override
public TransformFunction createTransformFunction(ServerInterface srvInterface)
{ return new TokenizeString(); }
```

Complete Java UDTF Example

The example code below is the complete code of the example developed in the topics Defining Your Java UDTF's Input and Output Table Columns, Subclassing the TransformFunction Class, and Overriding the createTransformFunction Method. To make code management simpler, the `TransformFunction` class is defined as an inner class of the `TransformFactoryClass`.

```
// You will need to specify the full package when creating functions based on // the clas
ses in your library.
package com.mycompany.example;
// Import the entire Vertica SDK
import com.vertica.sdk.*;

  // Break a single string input into individual words (substrings delimited by
  // one or more spaces).
  public class TokenFactory extends TransformFunctionFactory
```

```
{
    // Set the number and data types of the columns in the input and output rows.
    @Override
    public void getPrototype(ServerInterface srvInterface,
                             ColumnTypes argTypes, ColumnTypes returnType)
    {
        // One column in the input row: a Varchar
        argTypes.addVarchar();
        // One column in the output row: a Varchar
        returnType.addVarchar();
    }


    // Set the width of any variable-width output columns, and also name
    // the columns.
    @Override
    public void getReturnType(ServerInterface srvInterface, SizedColumnTypes
                              inputTypes, SizedColumnTypes outputTypes)
    {
        // Set the maximum width of the return column to the width
        // of the input column and name the output column "Tokens"
        outputTypes.addVarchar(
            inputTypes.getColumnType(0).getStringLength(), "Tokens");
    }


    public class TokenizeString extends TransformFunction
    {
        @Override
        public void processPartition(ServerInterface srvInterface,
                                     PartitionReader inputReader,
                                     PartitionWriter outputWriter)
                    throws UdfException, DestroyInvocation
        {
            // Loop over all rows passed in in this partition.
            do {
                // First test if the input string is NULL. If so, return NULL
                if (inputReader.isStringNull(0)) {
                    outputWriter.setStringNull(0);
                } else {
                    // Get the string value in column zero and break it into a
                    // it into words. Output each word as its own
                    // value.
                    String[] tokens = inputReader.getString(0).split("\\s+");
                    // Output each word on a separate row.
                    for (int i = 0; i < tokens.length; i++)  {
                        // Output a string value in column 0 of the output
                        outputWriter.getStringWriter(0).copy(tokens[i]);
                        // Move to the next row of output
                        outputWriter.next();
                    }
                }
            // Loop until there are no more input rows in partition.
            } while (inputReader.next());
        }
    }


    @Override
    public TransformFunction createTransformFunction(ServerInterface srvInterface)
    { return new TokenizeString(); }

}
```

```
// Break a single string input into individual words (substrings delimited by
// one or more spaces).
public class TokenFactory extends TransformFunctionFactory
{
    // Set the number and data types of the columns in the input and output rows.
    @Override
    public void getPrototype(ServerInterface srvInterface,
                             ColumnTypes argTypes, ColumnTypes returnType)
    {
        // One column in the input row: a Varchar
        argTypes.addVarchar();
        // One column in the output row: a Varchar
        returnType.addVarchar();
    }
```

```
    // Set the width of any variable-width output columns, and also name
    // the columns.
    @Override
    public void getReturnType(ServerInterface srvInterface, SizedColumnTypes
                               inputTypes, SizedColumnTypes outputTypes)
    {
        // Set the maximum width of the return column to the width
        // of the input column and name the output column "Tokens"
        outputTypes.addVarchar(
            inputTypes.getColumnType(0).getStringLength(), "Tokens");
    }
```

```
    public class TokenizeString extends TransformFunction
    {
        @Override
        public void processPartition(ServerInterface srvInterface,
                                     PartitionReader inputReader,
                                     PartitionWriter outputWriter)
                    throws UdfException, DestroyInvocation
        {
            // Loop over all rows passed in in this partition.
            do {
                // First test if the input string is NULL. If so, return NULL
                if (inputReader.isStringNull(0)) {
                    outputWriter.setStringNull(0);
                } else {
                    // Get the string value in column zero and break it into a
                    // it into words. Output each word as its own
                    // value.
                    String[] tokens = inputReader.getString(0).split("\\s+");
                    // Output each word on a separate row.
                    for (int i = 0; i < tokens.length; i++)  {
                        // Output a string value in column 0 of the output
                        outputWriter.getStringWriter(0).copy(tokens[i]);
                        // Move to the next row of output
                        outputWriter.next();
                    }
                }
            // Loop until there are no more input rows in partition.
            } while (inputReader.next());
        }
    }
```

```
    @Override
    public TransformFunction createTransformFunction(ServerInterface srvInterface)
    { return new TokenizeString(); }
```

```
}
```

Deploying and Using Your Java UDTF

Once you have finished developing the code for your Java UDTF, you need to deploy it:

1. Compile your code and package it into a JAR file. See Compiling and Packaging a Java UDF.
2. Copy your UDTF JAR file to a host in your database. You only need to copy it to a single host—HP Vertica automatically distributes the JAR file to the rest of the hosts in your database when you add the library to the database catalog.
3. Connect to the host as the database administrator.
4. Connect to the database using **vsql**.
5. Add your library to the database catalog using the CREATE LIBRARY statement.

   ```
   => CREATE LIBRARY libname AS '/path_to_jar_file/filename.jar'
   -> LANGUAGE 'Java';
   ```

   The *libname* is the name you want to use to reference the library, *path_to_jar_file/filename*.jar is the fully-qualified pathname to the JAR file you copied to the host.

   For example, if you created a JAR file named TokenizeStringLib.jar and copied it to the dbadmin account's home directory, you would use this command to load the library:

```
=> CREATE LIBRARY tokenizelib AS '/home/dbadmin/TokenizeStringLib.jar'-> LANGUAGE 'Ja
va';
```

6. Define your UDTF in the catalog using the CREATE TRANSFORM FUNCTION statement:

   ```
   => CREATE TRANSFORM FUNCTION functionName AS LANGUAGE 'Java' NAME
   S-> 'namespace.factoryClassName' LIBRARY libname;
   ```

   The *functionName* is the name want to give your function. The *namespace.factoryClassName* is the fully-qualified name (namespace and class name) of your UDTF's factory class. The *libname* is the name you gave your Java UDF library in step 5.

   For example, to define the tokenize function whose code is shown in Complete Java UDTF Example, you would use the command:

   ```
   => CREATE TRANSFORM FUNCTION tokenize AS LANGUAGE 'Java' NAME
   -> 'com.mycompany.example.TokenFactory' LIBRARY tokenizelib;
   ```

7. You can now call your UDTF. For example:

   ```
   => CREATE TABLE T (url varchar(30), description varchar(2000));
   CREATE TABLE
   => INSERT INTO T VALUES ('www.amazon.com','Online retail merchant and provider of clo
   ud services');
    OUTPUT
   --------
         1
   (1 row)
   => INSERT INTO T VALUES ('www.hp.com','Leading provider of computer hardware and imag
   ing solutions');
    OUTPUT
   --------
         1
   (1 row)
   => INSERT INTO T VALUES ('www.vertica.com','World''s fastest analytic database');
    OUTPUT
   --------
         1
   (1 row)
   => COMMIT;
   COMMIT
   => SELECT url, tokenize(description) OVER (partition by url) FROM T;
          url       |  Tokens
   -----------------+-----------
    www.hp.com      | Leading
    www.hp.com      | provider
    www.hp.com      | of
    www.hp.com      | computer
    www.hp.com      | hardware
    www.hp.com      | and
   ```

```
  www.hp.com       | imaging
  www.hp.com       | solutions
  www.vertica.com  | World's
  www.vertica.com  | fastest
  www.vertica.com  | analytic
  www.vertica.com  | database
  www.amazon.com   | Online
  www.amazon.com   | retail
  www.amazon.com   | merchant
  www.amazon.com   | and
  www.amazon.com   | provider
  www.amazon.com   | of
  www.amazon.com   | cloud
  www.amazon.com   | services
(20 rows)
```

Compiling and Packaging a Java UDF

Before you can use your Java UDF, you need to compile it and package it into a JAR file.

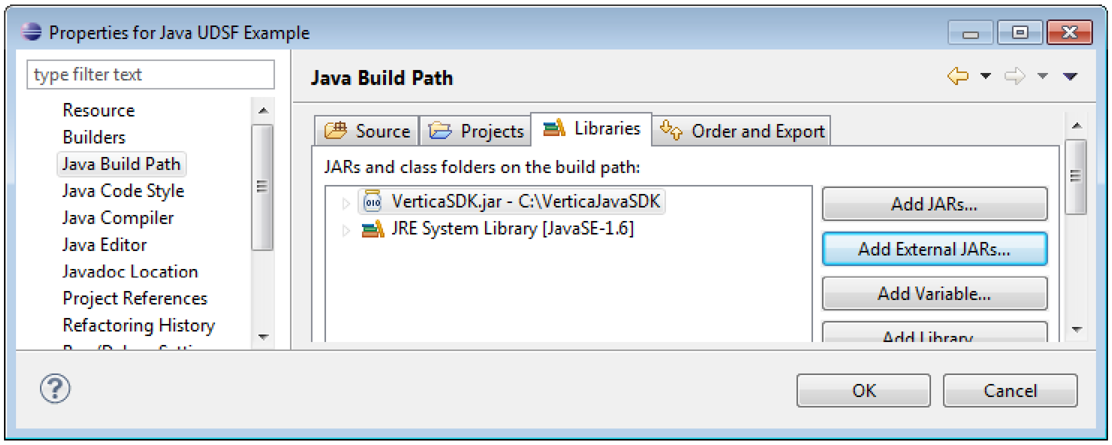
Compiling Your Java UDF

You need to include the HP Vertica Java SDK JAR file in the classpath when you compile your Java UDF source files into classes, so the Java compiler can resolve the HP Vertica Java SDK API calls. If you are using the command-line Java compiler on a host in your database cluster, you would use the command:

```
$ javac -classpath /opt/vertica/bin/VerticaSDK.jar factorySource.java \
      [functionSource.java...]
```

If you use multiple source files in your UDF, you must compile all of them into class files. The easiest method is to use the wildcard `*.java` to compile all of the Java files in the directory.

If you are using an IDE, make sure that a copy of the `VerticaSDK.jar` file is in the build path. For example, if you are using Eclipse on a Windows system to develop and compile your UDF, you will need to copy `VerticaSDK.jar` to it and then include it in your UDF's project.



JARs and class folders on the build path:
VerticaSDK.jar - C:\VerticaJavaSDK
JRE System Library [JavaSE-1.6]

Packaging Your UDF into a JAR File

Once you have compiled your UDF, you must package its class files and the `BuildInfo.class` file into a JAR file (see Configuring Your Java Development Environment).

> **Note:** You can package as many UDFs as you want into the same JAR file. Bundling your UDFs together saves you from having to load multiple libraries.

If you are using the jar command packaged as part of the JDK, you must have your UDF class files organized into a directory structure that match your class's package. For example, if your UDF's factory class has a fully-qualified name of `com.mycompany.udfs.Add2ints`, your class files must must be in the directory hierarchy `com/mycompany/udfs` relative to the your project's base directory. In addition, you must have a copy of the `BuildInfo.class` in the path `com/vertica/sdk` so it can be included in the JAR file. This class must be present in your JAR file to indicate the SDK version that was used to compile your Java UDF.

> **Note:** The `BuildInfo.class` and `VerticaSDK.jar` that you used to compile your class files must be from the same SDK version, and that both must match the version of the SDK files on your HP Vertica hosts. Versioning is only an issue if you are not compiling your UDFs on an HP Vertica host. If you are compiling on a separate development system, always refresh your copies of these files and recompile your UDFs just before deploying them.

For example, the Add2ints UDSF example explained in Developing a User Defined Scalar Function in Java could have the following directory structure after it has been compiled:

```
com/vertica/sdk/BuildInfo.class
com/mycompany/example/Add2intsFactory.class
com/mycompany/example/Add2intsFactory$Add2ints.class
```

To create a JAR file from the command line:

1. Change to the root directory of your project.

2. Use the jar command to package the `BuildInfo.class` file and all of the classes in your UDF:

   ```
   # jar -cvf libname.jar com/vertica/sdk/BuildInfo.class \
          packagePath/*.class
   ```

   where *libname* is the filename you have chosen for your JAR file (choose whatever name you like), and *packagePath* is the path to the directory containing your UDF's class files.

   For example, to package the files from the Add2ints example, you use the command:

   ```
   # jar -cvf Add2intsLib.jar com/vertica/sdk/BuildInfo.class \
         com/mycompany/example/*.class
   ```

Note: You must include all of the class files that make up your UDF in your JAR file. Your UDF always consists of at least two classes (the factory class and the function class). Even if you defined your function class as an inner class of your factory class, Java generates a separate class file for the inner class.

Once you have packaged your UDF into a JAR file, you are ready to deploy it to your HP Vertica database. See Deploying and Using Your Java UDSF and Deploying and Using Your Java UDTF for details.

Handling Dependencies

There are several methods you can use to handle any JARs that your UDF relies on:

- Install the JAR files on each host in your database and add the directory containing them to the host's CLASSPATH environment variable. This method has several drawbacks, since you need to remember to copy the JARs over to any newly-deployed nodes and you need to ensure that the same version of the library is installed on each node.
- Bundle the JARs into your UDF JAR file using a tool such as One-JAR or Eclipse's Runnable JAR Export Wizard to package dependencies into the JAR file.
- Try unpacking the JAR file and repacking its contents in your UDF's JAR file.

Handling Errors

If your UDF encounters an unrecoverable error, it should instantiate and throw a `UdfException`. The exception causes the transaction containing the function call to be rolled back.

The `UdfException` constructor takes a numeric code (which can be anything you want since it is just reported in the error message) and an error message string. If you want to report additional diagnostic information about the error, you can write messages to a log file before throwing the exception (see Writing Messages to the Log File).

The following code fragment demonstrates adding error checking to the Add2ints UDSF example (shown in Complete Java UDSF Example). If either of the arguments are NULL, the `processBlock` method throws an exception.

```
@Override
public void processBlock(ServerInterface srvInterface,
                         BlockReader argReader,
                         BlockWriter resWriter)
            throws UdfException, DestroyInvocation
{
    do {
        // Test for NULL value. Throw exception if one occurs.
        if (argReader.isLongNull(0) || argReader.isLongNull(1) ) {
            // No nulls allowed. Throw exception
            throw new UdfException(1234, "Cannot add a NULL value");
```

```
                }
```

Note: This example isn't realistic, since you would likely just replace the NULL value with a zero or return a NULL value. Your UDF should only throw an exception if there is no way to compensate for the error.

When your UDF throws an exception, the side process running your UDF reports the error back to HP Vertica and exits. HP Vertica displays the error message contained in the exception and a stack trace to the user:

```
=> SELECT add2ints(2, NULL);
ERROR 3399:  Failure in UDx RPC call InvokeProcessBlock(): Error in User Defined Object [
add2ints], error code: 1234
com.vertica.sdk.UdfException: Cannot add a NULL value
        at com.mycompany.example.Add2intsFactory$Add2ints.processBlock(Add2intsFactory.ja
va:37)
        at com.vertica.udxfence.UDxExecContext.processBlock(UDxExecContext.java:700)
        at com.vertica.udxfence.UDxExecContext.run(UDxExecContext.java:173)
        at java.lang.Thread.run(Thread.java:662)
```

Handling Cancel Requests

The query that calls your UDF can be canceled (usually, by the user pressing CTRL+C in **vsql**). When the calling query is canceled, HP Vertica begins a process of shutting down your UDF. Since UDTFs can perform lengthy and costly processing, the HP Vertica Java SDK defines several ways that HP Vertica attempts to signal UDTFs to terminate before it takes the step of killing the fenced-mode JVM process that is executing the UDTF. These attempts to signal the UDTF can help reduce the amount of CPU and memory that is wasted by having the UDF process continue processing after its results are no longer required.

When the user cancels a UDF, HP Vertica takes the following steps:

1. It sets the `isCanceled` property on UDTFs to true. Your `processPartition` methods can test this property to see if the function call has been canceled.
2. It calls UDTF's `TransformFunction.cancel` method. You should override this method to perform any shutdown tasks (such as killing threads).
3. It calls all types of UDF's `destroy` method. You should implement this method to free any resources your UDF has allocated.
4. It kills the JVM process running your UDF.

The topics in this section explain how your UDTF can use the cancel API.

Exiting When the Calling Query Has Been Canceled

Since User Defined Transform Functions (UDTFs) often perform lengthy and CPU-intensive processing, it makes sense for them to terminate if the query that called them has been canceled. Exiting when the query has been canceled helps prevent wasting CPU cycles and memory on continued processing.

The `TransformFunction` class has a getter named `.isCanceled` that returns true if the calling query has been canceled. Your `processPartition` method can periodically check the value of this getter to determine if the query has been canceled, and exit if it has.

How often your `processPartition` function calls `isCanceled` depends on how much processing it performs on each row of data. Calling `isCanceled` does add overhead to your function, so you shouldn't call it too often. For transforms that do not perform lengthy processing, you could check for cancelation every 100 or 1000 rows. If your `processPartition` performs extensive processing for each row, you may want to check `isCanceled` every 10 or so rows.

The following code fragment shows how you could have the `StringTokenizer` UDTF example check whether its query has been canceled:

```
public class CancelableTokenizeString extends TransformFunction
{
    @Override
    public void processPartition(ServerInterface srvInterface,
                                  PartitionReader inputReader,
                                  PartitionWriter outputWriter)
                throws UdfException, DestroyInvocation
    {
        // Loop over all rows passed in in this partition.

        int rowcount = 0; // maintain count of rows processed
        do {
            rowcount++; // Processing new row

            // Check for cancelation every 100 rows
            if (rowcount % 100 == 0) {
                // Check to see if Vertica marked this class as canceled
                if (this.isCanceled()) {
                    srvInterface.log("Got canceled! Exiting...");
                    return;
                }
            }
            // Rest of the function here
            .         .        .
```

This example checks for cancelation after processing 100 rows in the partition of data. If the query has been canceled, the example logs a message, then returns to the caller to exit the function.

Note: You need to strike a balance between adding overhead to your functions by calling `isCanceled` and having your functions waste CPU time by running after their query has been canceled (a rare event). For functions such as `StringTokenizer` which have a low overall processing cost, it usually does not make sense to test for cancelation. The cost of adding overhead to all function calls outweigh the amount of resources wasted by having the function run to completion or having its JVM process killed by HP Vertica on the rare occasions that its query is canceled.

Overriding the Cancel Method

Your User Defined Transform Function (UDTF) can override the `TransformFunction.cancel` method that HP Vertica calls if the query that called the function has been canceled. You should override this method to perform an orderly shutdown of any additional processing that your UDF spawned. For example, you can have your `cancel` method shut down threads that your UDTF has spawned or signal a third-party library that it needs to stop processing and exit. Your `cancel` method must leave your UDTF's function class ready to be destroyed, since HP Vertica calls the UDF's `destroy` method after the `cancel` method has exited.

Notes

- If your UDTF does not override `cancel`, HP Vertica assumes your UDTF does not need to perform any special cancel processing and calls the function class's `destroy` method to have it free any resources.

- Your `cancel` method is called from a different thread than the thread running your UDF's `processPartition` function.
- The call to the `cancel` method is not synchronized in any way with your UDTF's `processPartition` method. If you need your `processPartition` function to exit before your `cancel` method performs some action (killing threads, for example) you need to have the two methods synchronize their actions.
- If your `cancel` method runs for too long, HP Vertica kills the JVM side process your UDF.

Communicating with HP Vertica Using ServerInterface

Every method in the HP Vertica SDK that you override to create your UDF receives an instance of the `ServerInterface` class object. This class is used to query information from and pass information back to the HP Vertica server.

There are two methods in this class that you can use in your UDFs:

- `log` writes a message to a log file stored in the `UDxLogs` directory of the database's catalog directory. See Writing Messages to the Log File for more information.
- `getLocale` gets the current session's locale.

Writing Messages to the Log File

Writing messages to a log is useful when you are debugging your Java UDFs, or you want to output additional information about an error condition. Your UDFs write messages to a log file by calling the `ServerInterface.log` method, passing it a printf-style `String` value along with any variables referenced in the string (see the java.util.Formatter class documentation for details of formatting this string value). An instance of the `ServerInterface` class is passed to every method you can override in the Java SDK (see Communicating with HP Vertica Using ServerInterface for more information).

The following code fragment demonstrates how you could log the values passed into the Add2ints UDSF example (see Complete Java UDSF Example for the full code).

```
@Override
public void processBlock(ServerInterface srvInterface,
                         BlockReader argReader,
                         BlockWriter resWriter)
            throws UdfException, DestroyInvocation
{
    do {
        // Get the two integer arguments from the BlockReader
        long a = argReader.getLong(0);
        long b = argReader.getLong(1);
        // Log the input values
        srvInterface.log("Got values a=%d and b=%d", a, b);
```

The messages are written to a log file stored in the catalog directory's `UDxlog` subdirectory named `UDxFencedProcessesJava.log`:

```
$ tail VMart/v_vmart_node0001_catalog/UDxLogs/UDxFencedProcesses.log
2012-12-12 10:23:47.649 [Java-2164] 0x01 UDx side process (Java) started
2012-12-12 10:23:47.871 [Java-2164] 0x0b [UserMessage] add2ints - Got
values a=5 and b=6
2012-12-12 10:23:48.598 [Java-2164] 0x0c Exiting UDx side process
```

The SQL name of the UDF is added to the log message, along with the string `[UserMessage]` to mark the entry as a message added by a call to the `log` method. These additions make it easier for you to filter the log to find the messages generated by your UDF.

Accepting Different Numbers and Types of Arguments

Usually, your UDFs accept a set number of arguments that are a specific data type (called its signature). You can create UDFs that handle multiple signatures, or even accept all arguments supplied to them by the user, using either of these techniques:

- Overloading your UDF by assigning the same SQL function name to multiple factory classes, each of which defines a unique function signature. When a user uses the function name in a query, HP Vertica tries to match the signature of the function call to the signatures declared by the factory's `getPrototype` method. This is the best technique to use if your UDF needs to accept a few different signatures (for example, accepting two required and one optional argument).

- Creating a polymorphic UDF by using the special "Any" argument type that tells Vertica to send all arguments that the user supplies to your function. Your UDF decides whether it can handle the arguments or not.

The following topics explain each of these techniques.

Overloading Your Java UDFs

You may want your UDF to accept several different signatures (sets of arguments). For example, you might want your UDF to accept:

- One or more optional arguments.

- One or more argument that can be one of several data types.

- Completely distinct signatures (either all INTEGER or all VARCHAR, for example).

You can create a function with this behavior by creating several factory classes each of which accept a different signature (the number and data types of arguments), and associate a single SQL function name with all of them. You can use the same SQL function name to refer to multiple factory classes as long as the signature defined by each factory is unique. When a user calls your UDF, HP Vertica matches the number and types of arguments supplied by the user to the arguments accepted by each of your function's factory classes. If one matches, HP Vertica uses it to instantiate a function class to process the data.

Multiple factory classes can instantiate the same function class, so you can re-use one function class that is able to process multiple sets of arguments and then create factory classes for each of the function signatures. You can also create multiple function classes if you want.

The following example code demonstrates creating a User Defined Scalar Function (UDSF) that adds two or three integers together. The Add2or3ints class is prepared to handle two or three arguments. It checks the number of arguments that have been passed to it, and adds all two or three of them together. The `processBlock` method checks whether it has been called with less than 2 or more than 3 arguments. In theory, this should never happen, since HP Vertica only calls the UDSF if the user's function call matches a signature on one of the factory classes you create for your function. In practice, it is a good idea to perform this sanity checking, in case your (or someone else's) factory class reports that your function class accepts a set of arguments that it actually does not.

```
// You need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.multiparamexample;
// Import the entire Vertica SDK
import com.vertica.sdk.*;
// This ScalarFunction accepts two or three integer arguments. It tests
// the number of input columns to determine whether to read two or three
// arguments as input.
public class Add2or3ints extends ScalarFunction
{
    @Override
    public void processBlock(ServerInterface srvInterface,
                             BlockReader argReader,
                             BlockWriter resWriter)
                throws UdfException, DestroyInvocation
    {
        // See how many arguments were passed in
        int numCols = argReader.getNumCols();

        // Return an error if less than two or more than 3 aerguments
        // were given. This error only occurs if a Factory class that
        // accepts the wrong number of arguments instantiates this
        // class.
        if (numCols < 2 || numCols > 3) {
            throw new UdfException(0,
                "Must supply 2 or 3 integer arguments");
        }

        // Process all of the rows of input.
        do {
            // Get the first two integer arguments from the BlockReader
            long a = argReader.getLong(0);
            long b = argReader.getLong(1);

            // Assume no third argument.
            long c = 0;

            // Get third argument value if it exists
            if (numCols == 3) {
                c = argReader.getLong(2);
            }

            // Process the arguments and come up with a result. For this
            // example, just add the three arguments together.
            long result = a+b+c;

            // Write the integer output value.
            resWriter.setLong(result);

            // Advance the output BlocKWriter to the next row.
            resWriter.next();

            // Continue processing input rows until there are no more.
        } while (argReader.next());
    }
}
```

The main difference between the `Add2ints` class and the `Add2or3ints` class is the inclusion of a section that gets the number of arguments by calling `BlockReader.getNumCols`. This class also tests the number of columns it received from HP Vertica to ensure it is in the range it is prepared to handle. This test will only fail if you create a `ScalarFunctionFactory` whose `getPrototype` method defines a signature that accepts less than two or more than three arguments. This is not really necessary in this simple example, but for a more complicated class it is a good idea to test the number of columns and data types that HP Vertica passed your function class.

Within the `do` loop, `Add2or3ints` uses a default value of zero if HP Vertica sent it two input columns. Otherwise, it retrieves the third value and adds that to the other two. Your own class needs to use default values for missing input columns or alter its processing in some other way to handle the variable columns.

You must define your function class in its own source file, rather than as an inner class of one of your factory classes since Java does not allow the instantiation of an inner class from outside its containing class. You factory class has to be available for instantiation by multiple factory classes.

Once you have created a function class or classes, you create a factory class for each signature you want your function class to handle. These factory classes can call individual function classes, or they can all call the same class that is prepared to accept multiple sets of arguments.

The following example `ScalarFunctionFactory` class is almost identical to the `Add2intsFactory` example explained in Defining the Arguments and Return Type for Your UDSF. The only difference is that its `createScalarFunction` method instantiates a member of the `Add2or3ints` class, rather than a member of `Add2ints`.

```
// You will need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.multiparamexample;
// Import the entire Vertica SDK
import com.vertica.sdk.*;
public class Add2intsFactory extends ScalarFunctionFactory
{
    @Override
       public void getPrototype(ServerInterface srvInterface,
                                ColumnTypes argTypes,
                                ColumnTypes returnType)
    {
        // Accept two integers as input
        argTypes.addInt();
        argTypes.addInt();
        // writes one integer as output
        returnType.addInt();
    }
    @Override
       public ScalarFunction createScalarFunction(ServerInterface srvInterface)
    {
        // Instantiate the class that can handle either 2 or 3 integers.
        return new Add2or3ints();
    }
}
```

The following `ScalarFunctionFactory` subclass accepts three integers as input. It, too, instantiates a member of the `Add2or3ints` class to process the function call:

```
// You will need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.multiparamexample;
// Import the entire Vertica SDK
import com.vertica.sdk.*;
public class Add3intsFactory extends ScalarFunctionFactory
{
    @Override
       public void getPrototype(ServerInterface srvInterface,
                                ColumnTypes argTypes,
                                ColumnTypes returnType)
    {
        // Accepts three integers as input
        argTypes.addInt();
        argTypes.addInt();
        argTypes.addInt();
        // Returns a single integer
        returnType.addInt();
    }
    @Override
       public ScalarFunction createScalarFunction(ServerInterface srvInterface)
    {
        // Instantiates the Add2or3ints ScalarFunction class, which is able to
        // handle eitehr 2 or 3 integers as arguments.
        return new Add2or3ints();
    }
}
```

The factory classes and the function class or classes they call must be packaged into the same JAR file (see Compiling and Packaging a Java UDF for details). If a host in the database cluster has the JDK installed on it, you could use the following commands to compile and package the example:

```
$ cd pathToJavaProject$ javac -classpath /opt/vertica/bin/VerticaSDK.jar \
> com/mycompany/multiparamexample/*.java
$ jar -cvf Add2or3intslib.jar com/vertica/sdk/BuildInfo.class \
> com/mycompany/multiparamexample/*.class
added manifest
adding: com/vertica/sdk/BuildInfo.class(in = 1202) (out= 689)(deflated 42%)
adding: com/mycompany/multiparamexample/Add2intsFactory.class(in = 677) (out= 366)(deflat
ed 45%)
adding: com/mycompany/multiparamexample/Add2or3ints.class(in = 919) (out= 601)(deflated 3
4%)
adding: com/mycompany/multiparamexample/Add3intsFactory.class(in = 685) (out= 369)(deflat
ed 46%)
```

Once you have packaged your overloaded UDF, you deploy it the same way as you do a regular UDF (see Deploying and Using Your Java UDSF and Deploying and Using Your Java UDTF), except you use multiple CREATE FUNCTION statements to define the function, once for each factory class.

```
=> CREATE LIBRARY add2or3intslib as '/home/dbadmin/Add2or3intslib.jar'
```

```
-> language 'Java';
CREATE LIBRARY
=> CREATE FUNCTION add2or3ints as LANGUAGE 'Java' NAME 'com.mycompany.multiparamexample.A
dd2intsFactory' LIBRARY add2or3intslib;
CREATE FUNCTION
=> CREATE FUNCTION add2or3ints as LANGUAGE 'Java' NAME 'com.mycompany.multiparamexample.A
dd3intsFactory' LIBRARY add2or3intslib;
CREATE FUNCTION
```

You call the overloaded function the same way you call any other function.

```
=> SELECT add2or3ints(2,3);
 add2or3ints
-------------
           5
(1 row)
=> SELECT add2or3ints(2,3,4);
 add2or3ints
-------------
           9
(1 row)
=> SELECT add2or3ints(2,3,4,5);
ERROR 3457:  Function add2or3ints(int, int, int, int) does not exist, or permission is de
nied for add2or3ints(int, int, int, int)
HINT:  No function matches the given name and argument types. You may need to add explici
t type casts
```

The last error was generated by HP Vertica, not the UDF code. It returns an error if it cannot find a factory class whose signature matches the function call's signature.

Creating an overloaded UDF is useful if you want your function to accept a limited set of potential arguments. If you want to create a more flexible function, you can create a polymorphic function (see Creating a Polymorphic Java UDF).

Creating a Polymorphic Java UDF

Polymorphic UDFs accept any number and type of argument that the user supplies. HP Vertica does not check the number or types of argument that the user passes to the UDF—it just passes the UDF all of the arguments supplied by the user. It is up to your polymorphic UDF's main processing method (for example, `processBlock` in User Defined Scalar Functions) to examine the number and types of arguments it received and determine if it can handle them.

Note: User Defined Transform Functions (UDTFs) can have an unlimited number of arguments. All other UDFs except UDTFs are limited to a maximum number of 32 arguments.

Polymorphic UDFs are more flexible than using multiple factory classes for your function (see Overloading Your Java UDFs), since you function can determine at run time if it can process the arguments rather than accepting specific sets of arguments. However, your polymorphic function needs to perform more work to determine whether it can process the arguments that it has been given.

Your polymorphic UDF declares it accepts any number of arguments in its factory's `getPrototype` method by calling the `addAny` method on the `ColumnTypes` object that defines its input arguments. This "any argument" argument type is the only one that your function can declare. You cannot define required arguments and then call `addAny` to declare the rest of the signature as optional. If your function has requirements for the arguments it accepts, its process method must enforce them.

The following example shows an implementation of a `ScalarFunctionFactory` class with an inner `ScalarFunction` class that adds together two or more integers.

```
// You will need to specify the full package when creating functions based on
// the classes in your library.
package com.mycompany.multiparamexample;
// Import the entire Vertica SDK
import com.vertica.sdk.*;
// Factory class to create polymorphic UDSF that adds all of the integer
// arguments it recieves and returns a sum.
public class AddManyIntsFactory extends ScalarFunctionFactory
{
    @Override
       public void getPrototype(ServerInterface srvInterface,
                            ColumnTypes argTypes,
                            ColumnTypes returnType)
    {
        // Accepts any number and type or arguments. The ScalarFunction
        // class handles parsing the arguments.
        argTypes.addAny();
        // writes one integer as output
        returnType.addInt();
    }
    // This polymorphic ScalarFunction adds all of the integer arguments passed
    // to it. Returns an error if there are less than two arguments, or if one
    // argument is not an integer.
    public class AddManyInts extends ScalarFunction
    {
        @Override
        public void processBlock(ServerInterface srvInterface,
                                 BlockReader argReader,
                                 BlockWriter resWriter)
                    throws UdfException, DestroyInvocation
        {
            // See how many arguments were passed in
            int numCols = argReader.getNumCols();

            // Return an error if less than two arguments were given.
            if (numCols < 2) {
                throw new UdfException(0,
                    "Must supply at least 2 integer arguments");
            }

            // Make sure all input columns are integer.
            SizedColumnTypes inTypes = argReader.getTypeMetaData();
            for (int param = 0; param < numCols; param++) {
                VerticaType paramType = inTypes.getColumnType(param);
                if (!paramType.isInt()) {
                    throw new UdfException(0, "Error: Argument " + (param+1) +
                    " was not an integer. All arguments must be integer.");
                }
            }

            // Process all of the rows of input.
            do {
                long total = 0; // Hold the running total of arguments

                // Get all of the arguments and add them up
                for (int x = 0; x < numCols; x++) {
                    total += argReader.getLong(x);
                }

                // Write the integer output value.
                resWriter.setLong(total);

                // Advance the output BlocKWriter to the next row.
                resWriter.next();

                // Continue processing input rows until there are no more.
            } while (argReader.next());
        }
    }

    @Override
       public ScalarFunction createScalarFunction(ServerInterface srvInterface)
    {
        // Instantiate the polymorphic UDF class.
        return new AddManyInts();
    }
}
```

The `ScalarFunctionFactory.getPrototype` method calls the `addAny` method to declare that the UDSF is polymorphic.

Most of the work in the example is done by the `ScalarFunction.processBlock` method. It performs two checks on the arguments that have been passed in through the `BlockReader` object:

- There are at least two arguments.
- The data type of all arguments are integers.

It is up to your polymorphic UDF to determine that all of the input passed to it is valid.

Once the `processBlock` validates its arguments, it loops over the them, adding them together.

You assign a SQL name to your polymorphic UDF using the same statement you use to assign one to a non-polymorphic UDF. The following demonstration shows how you load and call the polymorphic function from the example.

```
=> CREATE LIBRARY addmanyintslib AS '/home/dbadmin/AddManyIntsLib.jar'
-> LANGUAGE 'Java';
CREATE LIBRARY
=> CREATE FUNCTION addmanyints AS LANGUAGE 'Java' NAME
-> 'com.mycompany.multiparamexample.AddManyIntsFactory' LIBRARY addmanyintslib;
CREATE FUNCTION
```

```
=> SELECT addmanyints(1,2,3,4,5,6,7,8,9,10);
 addmanyints
-------------
          55
(1 row)
=> SELECT addmanyints(1); --Too few parameters
ERROR 3399:  Failure in UDx RPC call InvokeProcessBlock(): Error in User
Defined Object [addmanyints], error code: 0
com.vertica.sdk.UdfException: Must supply at least  2 integer arguments
        at
com.mycompany.multiparamexample.AddManyIntsFactory$AddManyInts.processBlock
(AddManyIntsFactory.java:39)
        at com.vertica.udxfence.UDxExecContext.processBlock(UDxExecContext.java:700)
        at com.vertica.udxfence.UDxExecContext.run(UDxExecContext.java:173)
        at java.lang.Thread.run(Thread.java:662)
=> SELECT addmanyints(1,2,3.14159); --Non-integer parameter
ERROR 3399:  Failure in UDx RPC call InvokeProcessBlock(): Error in User
Defined Object [addmanyints], error code: 0
com.vertica.sdk.UdfException: Error: Argument 3 was not an integer. All
arguments must be integer.
        at
com.mycompany.multiparamexample.AddManyIntsFactory$AddManyInts.processBlock(AddManyIntsFa
ctory.java:48)
        at com.vertica.udxfence.UDxExecContext.processBlock(UDxExecContext.java:700)
        at com.vertica.udxfence.UDxExecContext.run(UDxExecContext.java:173)
        at java.lang.Thread.run(Thread.java:662)
```

Polymorphic UDFs and Schema Search Paths

If a user does not supply a schema name as part of a function call, HP Vertica searches each schema in the schema search path for a function whose name and signature match the function call. See Setting Schema Search Paths in the Administrator's Guide for more information about schema search paths.

Since polymorphic functions do not have a specific signature associated with them, HP Vertica initially skips them when searching for a function to handle the function call. If none of the schemas in the search path contain a function whose name and signature match the function call, HP Vertica searches the schema search path again for a polymorphic function whose name matches the function name in the function call.

This behavior gives precedence to functions whose signature exactly matches the function call. It allows you to create a "catch all" polymorphic function that is called only if none of the non-polymorphic functions with the same name have matching signatures.

This behavior may cause confusion if your users expect the the first polymorphic function in the schema search path to handle a function call. To avoid confusion, you should:

- Avoid using the same name for different functions. You should always uniquely name functions unless you intend to create an overloaded function with multiple signatures.
- When you cannot avoid having functions with the same name in different schemas, always supply the schema name as part of the function call. Using the schema name prevents ambiguity and ensures that HP Vertica uses the correct function to process your function calls.

UDF Parameters

Parameters let you define arguments for your UDFs that remain constant across all of the rows processed by the SQL statement that calls you UDF. Typically, your UDFs accept arguments that come from columns in a SQL statement. For example, in the following SQL statement, the arguments a and b to the add2ints UDSF change value for each row processed by the SELECT statement:

```
=> SELECT a, b, add2ints(a,b) AS 'sum' FROM example;
 a | b  | sum
---+----+-----
 1 |  2 |   3
 3 |  4 |   7
 5 |  6 |  11
 7 |  8 |  15
 9 | 10 |  19
(5 rows)
```

Parameters remain constant for all the rows your UDF processes. You can also make parameters optional so that if the user does not supply it, your UDF uses a default value. For example, the following example demonstrates calling a UDSF named add2intsWithConstant that has a single parameter value named constant whose value is added to each the arguments supplied in each row of input:

```
=> SELECT a, b, add2intsWithConstant(a, b USING PARAMETERS constant=42)
-> AS 'a+b+42' from example;
 a | b  | a+b+42
---+----+--------
 1 |  2 |     45
 3 |  4 |     49
 5 |  6 |     53
 7 |  8 |     57
 9 | 10 |     61
(5 rows)
```

Note: When calling a UDF with parameters, there is no comma between the last argument and the USING PARAMETERS clause.

The topics in this section explain how develop UDFs that accept parameters.

Defining the Parameters Your Java UDF Accepts

You define the parameters that your UDF accepts in its factory class (`ScalarFunctionFactory`, `TransformFunctionFactory`, etc.) by implementing the `getParameterType` method. This method is similar to the `getReturnType` method: you call data-type-specific methods on a `SizedColumnTypes` object that is passed in as an argument. Each of these method calls sets the name, data type, and width or precision (if the data type requires it) of the parameter.

The following code fragment demonstrates adding a single parameter to the add2ints UDSF example (see Developing a User Defined Scalar Function in Java). The `getParameterType` method defines a single integer parameter that is named constant.

```
package com.mycompany.example;
import com.vertica.sdk.*;
public class Add2intsWithConstantFactory extends ScalarFunctionFactory
{
    @Override
    public void getPrototype(ServerInterface srvInterface,
                             ColumnTypes argTypes,
                             ColumnTypes returnType)
    {
        argTypes.addInt();
        argTypes.addInt();
        returnType.addInt();
    }

    @Override
    public void getReturnType(ServerInterface srvInterface,
                              SizedColumnTypes argTypes,
                              SizedColumnTypes returnType)
    {
        returnType.addInt("sum");
    }

    // Defines the parameters for this UDSF. Works similarly to defining
    // arguments and return types.
    public void getParameterType(ServerInterface srvInterface,
                              SizedColumnTypes parameterTypes)
    {
        // One INTEGER parameter named constant
        parameterTypes.addInt("constant");
    }

    @Override
    public ScalarFunction createScalarFunction(ServerInterface srvInterface)
    {
        return new Add2intsWithConstant();
    }
}
```

See the HP Vertica Java SDK entry for `SizedColumnTypes` for a full list of the data-type-specific methods you can call to define parameters.

Accessing Parameter Values

Your UDF uses the parameter values it declared in its factory class (see Defining the Parameters Your Java UDF Accepts) in its function class's process method (for example, `processBlock` or `processPartition`). It reads parameter values from a `ParamReader` object, which is available from the `ServerInterface` object that is passed to your process method. Reading parameters from this object is similar to reading argument values from `BlockReader` or `PartitionReader` objects: you call a data-type-specific method with the name of the parameter whose value you want to read. For example:

```
// Get the parameter reader from the ServerInterface to see if
// there are supplied parameters
ParamReader paramReader = srvInterface.getParamReader();
// Get the value of an integer parameter named constant
long constant = paramReader.getLong("constant");
```

Note: String data values do not have any of their escape characters processed before they are passed to your function. Therefore, your function may need to process the escape sequences itself if it needs to operate on unescaped character values.

Testing Whether the User Supplied Parameter Values

Unlike arguments, HP Vertica does not immediately return an error if a user's UDF function call does not include a value for a parameter defined by your UDF's factory class. This means that your function can attempt to read a parameter value that the user did not supply. If it does so, HP Vertica returns a non-existent parameter error to the user, and the query containing the function call is canceled. This behavior is fine if you want a parameter to be required by your UDF—just attempt to access its value. If the user didn't supply a value, HP Vertica reports the resulting error about a missing parameter to the user.

If you want your parameter to be optional, you can test whether the user supplied a value for the parameter before attempting to access its value. Your function determines if a value exists for a particular parameter by calling the `ParamReader.containsParameter` method with the parameter's name. If this function returns true, your function can safely retrieve the value. If this function returns false, your UDF can use a default value or change its processing in some other way to compensate for not having the parameter value. As long as your UDF does not try to access the non-existent parameter value, HP Vertica does not generate an error or warning about missing parameters.

Note: If the user passes your UDF a parameter that it has not defined, HP Vertica issues a warning that the parameter is not used. It still executes the SQL statement, ignoring the parameter.

The following code fragment demonstrates using the parameter value that was defined in the example shown in Defining the Parameters Your Java UDF Accepts . The `Add2intsWithConstant` class defines a UDF that adds two integer values. If the user supplies it, the function also adds the value of the optional integer parameter named constant.

```
    // Actual function class, declared here as a subclass of the factory to
    // keep things simple.
    public class Add2intsWithConstant extends ScalarFunction
    {
        @Override
        public void processBlock(ServerInterface srvInterface,
                                 BlockReader arg_reader,
                                 BlockWriter res_writer)
                    throws UdfException,DestroyInvocation
        {
            long constant = 0;
            // Get the parameter reader from the ServerInterface to see if
            // there are supplied parameters
            ParamReader paramReader = srvInterface.getParamReader();
            // See if the user supplied the constant parameter
            if (paramReader.containsParameter("constant"))
                // There is a parameter, so get its value.
                constant = paramReader.getLong("constant");


            do {
                long a = arg_reader.getLong(0);
                long b = arg_reader.getLong(1);
                // srvInterface.log("a = %d, b = %d", a, b);
                res_writer.setLong(a+b+constant);
                // srvInterface.log("writing result = %d", a+b);
                res_writer.next();
            } while (arg_reader.next());
        }
    }
```

Using Parameters in the Factory Class

In addition to using parameters in your UDF function class, you can also access the parameters in the factory class. You may want to access the parameters to let the user control the input or output values of your function in some way. For example, your UDF can have a parameter that lets the user choose to have your UDF return a single or double-precision value. The process of accessing parameters in the factory class is the same as accessing it in the function class: get a `ParamReader` object from the `ServerInterface.getParamReader` function, them read the parameter values.

Calling UDFs with Parameters

You pass parameters to a UDF by adding a USING PARAMETERS clause in the function call after the last argument. There is no comma between the last argument and the USING PARAMETERS clause. After the USING PARAMETERS clause you add one or more parameter definitions which contains the parameter name, followed by an equal sign, then the parameter's value. Multiple parameter definitions are separated by commas.

Note: Parameter values can be a constant expression (for example `1234 + SQRT(5678)`). You cannot use volatile functions (such as RANDOM) in the expression, since they do not return a constant value. If you do supply a volatile expression as a parameter value, HP Vertica returns an incorrect parameter type warning, and tries to run the UDF without the parameter value. If the UDF requires the parameter, it returns its own error which cancels the query.

The following example demonstrates calling the add2intsWithConstant UDSF example from Defining the Parameters Your UDF Accepts and Getting Parameter Values in UDFs:

```
=> SELECT a, b, add2intsWithConstant(a, b USING PARAMETERS constant=42)
```

```
-> AS 'a+b+42' from example;
 a | b  | a+b+42
---+----+--------
 1 |  2 |     45
 3 |  4 |     49
 5 |  6 |     53
 7 |  8 |     57
 9 | 10 |     61
(5 rows)
```

Multiple parameters are separated by commas. The following example calls a version of the tokenize UDTF that has parameters to limit the shortest allowed word and force the words to be output in uppercase.

```
=> SELECT url, tokenize(description USING PARAMETERS
-> minLength=4, uppercase=true) OVER (partition by url) FROM T;
       url       |   words
-----------------+-----------
 www.amazon.com  | ONLINE
 www.amazon.com  | RETAIL
 www.amazon.com  | MERCHANT
 www.amazon.com  | PROVIDER
 www.amazon.com  | CLOUD
 www.amazon.com  | SERVICES
 www.hp.com      | LEADING
 www.hp.com      | PROVIDER
 www.hp.com      | COMPUTER
 www.hp.com      | HARDWARE
 www.hp.com      | IMAGING
 www.hp.com      | SOLUTIONS
 www.vertica.com | WORLD'S
 www.vertica.com | FASTEST
 www.vertica.com | ANALYTIC
 www.vertica.com | DATABASE
(16 rows)
```

The add2intsWithConstant UDSF's constant parameter is optional; calling it without the parameter does not return an error or warning:

```
=> SELECT a,b,add2intsWithConstant(a, b) AS 'sum' FROM example;
 a | b  | sum
---+----+-----
 1 |  2 |   3
 3 |  4 |   7
 5 |  6 |  11
 7 |  8 |  15
 9 | 10 |  19
(5 rows)
```

Calling a UDF with incorrect parameters does generate a warning, but the query still runs:

```
=> SELECT a, b,  add2intsWithConstant(a, b USING PARAMETERS wrongparam=42)
-> AS 'result' from example;
```

```
WARNING 4332:  Parameter wrongparam was not registered by the function and cannot
be coerced to a definite data type
 a | b  | result
---+----+--------
 1 |  2 |      3
 3 |  4 |      7
 5 |  6 |     11
 7 |  8 |     15
 9 | 10 |     19
(5 rows)
```

Adding Metadata to Java UDx Libraries

You can add metadata, such as author name, the version of the library, a description of your library, and so on to your library. This metadata lets you track the version of your function that is deployed on an Vertica Analytics Platform cluster and lets third-party users of your function know who created the function. Your library's metadata appears in the USER_LIBRARIES system table after your library has been loaded into the Vertica Analytics Platform catalog.

To add metadata to your Java UDx library, you create a subclass of the `UDXLibrary` class that contains your library's metadata. You then include this class within your JAR file. When you load your class into the Vertica Analytics Platform catalog using the CREATE LIBRARY statement, looks for a subclass of `UDXLibrary` for the library's metadata.

In your subclass of `UDXLibrary`, you need to implement eight getters that return String values containing the library's metadata. The getters in this class are:

- `getAuthor()` returns the name you want associated with the creation of the library (your own name or your company's name for example).
- `getLibraryBuildTag()` returns whatever String you want to use to represent the specific build of the library (for example, the SVN revision number or a timestamp of when the library was compiled). This is useful for tracking instances of your library as you are developing them.
- `getLibraryVersion()` returns the version of your library. You can use whatever numbering or naming scheme you want.
- `getLibrarySDKVersion()` returns the version of the Vertica Analytics Platform SDK Library for which you've compiled the library.

  Note: This field isn't used to determine whether a library is compatible with a version of the Vertica Analytics Platform server. The version of the Vertica Analytics Platform SDK you use to compile your library is embedded in the library when you compile it. It is this information that Vertica Analytics Platform server uses to determine if your library is compatible with it.

- `getSourceUrl()` returns a URL where users of your function can find more information about it. This can be your company's website, the GitHub page hosting your library's source code, or whatever site you like.
- `getDescription()` returns a concise description of your library.
- `getLicensesRequired()` returns a placeholder for licensing information. In this release of Vertica Analytics Platform, you must leave this field as an empty string.
- `getSignature()` returns a placeholder for a signature that will authenticate your library. In this release of Vertica Analytics Platform, you must leave this field as an empty string.

For example, the following code demonstrates creating a UDXLibrary subclass to be included in the Add2Ints UDSF example JAR file (see Complete Java UDSF Example).

```
// Import the UDXLibrary class to hold the metadata
import com.vertica.sdk.UDXLibrary;

public class Add2IntsLibrary extends UDXLibrary
{
       // Return values for the metadata about this library.

       @Override public String getAuthor() {return "Whizzo Analytics Ltd.";}
       @Override public String getLibraryBuildTag() {return "1234";}
       @Override public String getLibraryVersion() {return "1.0";}
       @Override public String getLibrarySDKVersion() {return "7.0.0";}
       @Override public String getSourceUrl() {
               return "http://example.com/add2ints";
       }
       @Override public String getDescription() {
               return "My Awesome Add 2 Ints Library";
       }
       @Override public String getLicensesRequired() {return "";}
       @Override public String getSignature() {return "";}
}
```

When the library containing the Add2IntsLibrary class loaded, the metadata appears in the USER_LIBRARIES system table:

```
=> CREATE LIBRARY JavaAdd2IntsLib AS :libfile LANGUAGE 'JAVA';
CREATE LIBRARY
>=> CREATE FUNCTION JavaAdd2Ints as LANGUAGE 'JAVA'  name 'com.mycompany.example.Add2Ints
Factory' library JavaAdd2IntsLib;
CREATE FUNCTION
>=> \x
Expanded display is on.
>=> SELECT * FROM USER_LIBRARIES WHERE lib_name = 'JavaAdd2IntsLib';
-[ RECORD 1 ]-----+--------------------------------------------
schema_name       | public
lib_name          | JavaAdd2IntsLib
lib_oid           | 45035996273869844
author            | Whizzo Analytics Ltd.
owner_id          | 45035996273704962
```

```
lib_file_name     | public_JavaAdd2IntsLib_45035996273869844.jar
md5_sum           | f3bfc76791daee95e4e2c0f8a8d2737f
sdk_version       | v7.0.0-20131105
revision          | 125200
lib_build_tag     | 1234
lib_version       | 1.0
lib_sdk_version   | 7.0.0
source_url        | http://example.com/add2ints
description       | My Awesome Add 2 Ints Library
licenses_required |
signature         |
```

Developing User Defined Load (UDL) Functions

The COPY statement is the primary way to load data into HP Vertica. This statement performs several steps while loading data:

- It reads data from a file or input stream.
- It optionally filters or converts it in some manner,such as decompressing the data using GZIP.
- It parses the data into tuples (for example, by breaking comma-separated data into individual columns).

After the final step, COPY inserts the data into a table (or rejects it, if it is not in the correct format).

In some cases, you may want to change how the COPY statement performs one or more of these steps. The User Defined Load (UDL) feature lets you develop one or more functions that change how the COPY statement operates. To align with the three major steps required to load data, you can implement three types of UDLs:

- User Defined Source: Controls how the COPY statement obtains the data it loads into the database. For example, by fetching it via HTTP or through cURL.
- User Defined Filter: Filters the data. For example, unzipping a file or converting UTF-16 to UTF-8, or by doing both in sequence. You can chain multiple User Defined Filters together to transform data in several ways.
- User Defined Parser: Parses the data into tuples that are ready to be inserted into a table. For example, extracting data from an XML-like format. You can optionally define a User Defined Chunker (UDChunker), to have the parser perform parallel parsing.

UDL Requirements

User Defined Load Functions:

- Are written using the HP Vertica C++ or Jave SDK and compiled into a shared library.
- Can have up to one source process. This single process can obtain data from multiple sources.
- Can have zero or more Filters.
- Can have up to one Parser.

Note: You can define a `UDChunker` for parsers written in C++, but UDL parsers written in Java do not support the `UDChunker`.

Deploying User Defined Load Functions

You use the CREATE LIBRARY statement to load your compiled UDL library into HP Vertica. For each function, you must use the appropriate CREATE statement to load the function into HP Vertica. There is a CREATE statement for each type of UDL:

- CREATE SOURCE
- CREATE FILTER
- CREATE PARSER

Important: Installing an untrusted UDL function can compromise the security of the server. UDx's can contain arbitrary code. In particular, UD Source functions can read data from any arbitrary location. It is up to the developer of the function to enforce proper security limitations. Superusers must not grant access to UDx's to untrusted users.

Conversely, you remove UDL libraries with DROP LIBRARY and remove UDL functions with the following:

- DROP SOURCE
- DROP FILTER
- DROP PARSER

Note: You cannot ALTER UDL functions.

Developing UDLs in C++

C++ is one of the supported development languages for UDLs. See Setting up a C++ UDF Development Environment for more information.

Requirements for C++ UDLs

C++ UDLs:

- Can run in Fenced Mode starting in version 7.0. Fenced mode is enabled by default when you create the filter, parser, or source function in HP Vertica unless you explicitly state otherwise. UDL code created before version 7.0 does not need to be modified to work in fenced mode.

- Must not permit an exception to be passed back to HP Vertica. Doing so could lead to issues such as memory leaks (caused by the memory allocated by the exception never being freed). Your UDL should always contain a top-level try-catch block to catch any stray exceptions caused by your code or libraries your code calls.

- Must properly free any resources that the UDL function allocates. Even a single byte of allocated memory that is not freed can become an issue in a UDL that is called over millions of rows. Instead of allocating memory directly, your function should use the memory allocation macros in the HP Vertica SDK. See Allocating Resources for UDFs for details.

The header files that define the majority of classes and methods are `VerticaUDx.h` and `VerticaUDl.h`. These header files, along with the main `Vertica.h` header file, are available in `/opt/vertica/sdk/include`.

The SDK documentation is available in the SDK itself at /opt/vertica/sdk/doc and online: HP Vertica SDK documentation.

UDL Source

Developing Source Functions for User Defined Load

UDL Source functions allow you to process a source of data using a method that is not built into HP Vertica. For example, accessing the data from an HTTP source using cURL. Only a single User Defined Source can be defined in a COPY statement, but that source function can pull data from multiple sources.

A Source can optionally be used with UDFilters and a UDParser. The source can be obtained using a UDSource function, passed through one or more UDFilters, and finally parsed by a UDParser before being loaded.

You must implement a UDSource class and a SourceFactory class for sources.

The HP Vertica SDK provides example source functions in `/opt/vertica/sdk/examples/SourceFunctions`.

Subclassing SourceFactory

About the Source Factory Class

The SourceFactory class performs initial validation and planning of the query and instantiates objects to perform further initialization on each node once the query has been distributed.

SourceFactory Methods:

You implement the following methods in your SourceFactory class:

- plan() - The plan method is used to check parameters, populate the plan data, and assign the work to one or more nodes.

  When developing your plan() method you should check the parameters that have been passed from the function call and provide a helpful error message if the arguments do not conform, but it is not required. You can also optionally populate the NodeSpecifyingPlanContext object with any information that must be passed to the other nodes doing the work from the initiator node. Finally, you must specify which nodes the source(s) are obtained from. You can split up the work so that one or multiple specific nodes load data, or specify that any node load the data.

- prepareUDSources() - This method directly instantiates all provided sources and returns a vector of the sources.

- getParameterType() - The getParameterType() method allows you to define the name and types of parameters that the function uses. HP Vertica uses this information to warn function callers that certain parameters that they provide have no effect, or that certain parameters are not being set and are reverting to default values. You should, as a best practice, define the types and parameters for your function, but using this method is optional.

The name of the factory class is the value used for the NAME modifier in the in the CREATE SOURCE statement.

After creating your SourceFactory, you must register it with RegisterFactory();

Example SourceFactory

The following example is provided is part of: `/opt/vertica/sdk/examples/SourceFunctions/cURL.cpp`. It defines the factory function for the curl source function.

```
class CurlSourceFactory : public SourceFactory {public:
    virtual void plan(ServerInterface &srvInterface,
            NodeSpecifyingPlanContext &planCtxt) {
        std::vector<std::string> args = srvInterface.getParamReader().getParamNames();
       /* Check parameters */
        if (args.size() != 1 || find(args.begin(), args.end(), "url") == args.end()) {
            vt_report_error(0, "You must provide a single URL.");
        }
        /* Populate planData */
        planCtxt.getWriter().getStringRef("url").copy(
                                    srvInterface.getParamReader().getStringRef("url"));

        /* Assign Nodes */
        std::vector<std::string> executionNodes = planCtxt.getClusterNodes();
        while (executionNodes.size() > 1) executionNodes.pop_back();
        // Only run on the first node in the list.
        planCtxt.setTargetNodes(executionNodes);
    }
    virtual std::vector<UDSource*> prepareUDSources(ServerInterface &srvInterface,
            NodeSpecifyingPlanContext &planCtxt) {
        std::vector<UDSource*> retVal;
        retVal.push_back(vt_createFuncObj(srvInterface.allocator, CurlSource,
                planCtxt.getReader().getStringRef("url").str()));
        return retVal;
    }
    virtual void getParameterType(ServerInterface &srvInterface,
                                  SizedColumnTypes &parameterTypes) {
        parameterTypes.addVarchar(65000, "url");
    }
};
RegisterFactory(CurlSourceFactory);
```

Subclassing UDSource

About the UDSource Class

The UDSource class is responsible for acquiring the data from an external source and producing that data in a streaming manner. A wrapper is also provided for UDSource called ContinuousUDSource. ContinuousUDSource provides an abstraction that allows you to treat the input data as a continuous stream of data. This allows you to write the data from the source "at will" instead of having to create an iterator to use the base UDSource method. ContinuousUDSource is available in `/opt/vertica/sdk/examples/HelperLibraries/ContinuousUDSource.h`.

UDSource Methods:

- setup() - Invoked before the first time that process() is called. Use this method to do things such as open file handles.
- destroy() - Invoked after the last time that process() is called. Use this method to do things such as close file handles.
- process() - Invoked repeatedly until it returns DONE or the query is canceled by the function caller. On each invocation, process() acquires more data and writes the data to the DataBuffer specified by 'output'.

  Returns OUTPUT_NEEDED if this source has more data to produce or DONE if it has no more data to produce.
- getSize() - Returns the estimates number of bytes that process() will return. This value is an estimate only and is used to indicate the file size in the LOAD_STREAMS table. getSize() can be called before setup is called. See the SDK documentation for additional important details about the getSize() method.

ContinuousUDSource Functions:

The ContinuousUDSource wrapper allows you to write and process the data "at will" instead of having to iterate through the data. An example of using ContinuousUDSource is provided in `/opt/vertica/sdk/examples/SourceFunctions/MultiFileCurlSource.cpp`.

- initialize() - Invoked before run(). You can optionally override this function to perform setup and initialization.
- run() - Processes the data. Use write() on the ContinuousWriter to write the data from the source.
- deinitialize() - Invoked after run() has returned. You can optionally override this function to perform tear-down and destruction.

Functions that are already implemented that you use in your code:

- yield() - use to yield control back to the server during idle or busy loops so the server can check for status changes or query cancelations.
- cw - A ContinuousWriter which is defined in `/opt/vertica/sdk/examples/HelperLibraries/CoroutineHelpers.h`. Used to write the data to the output data buffer.

Example UDSource

The following example loads the source with the url_fread method in the helper library available in `/opt/vertica/sdk/examples/HelperLibraries/`. It allows you to use cURL to open and read in a file over HTTP. The data is loaded in chunks. If End Of File is received then the process() method returns DONE, otherwise it returns OUTPUT_NEEDED and process() processes another chunk of data. The functions included in the helper library (`url_fread()`, `url_fopen`, etc.) are based on examples that come with the libcurl library. For example, see http://curl.haxx.se/libcurl/c/fopen.html.

For setup, the handle to the file is opened, again using a function from the help library.

For destroy, the handle to the file is closed using a function from the helper library.

```
class CurlSource : public UDSource {private:
    URL_FILE *handle;
    std::string url;
    virtual StreamState process(ServerInterface &srvInterface, DataBuffer &output) {
        output.offset = url_fread(output.buf, 1, output.size, handle);
        return url_feof(handle) ? DONE : OUTPUT_NEEDED;
    }
public:
    CurlSource(std::string url) : url(url) {}
    void setup(ServerInterface &srvInterface) {
        handle = url_fopen(url.c_str(),"r");
    }
    void destroy(ServerInterface &srvInterface) {
        url_fclose(handle);
    }
};
```

UDL Filter

Developing Filter Functions for User Defined Load

UDL Filter functions allow you to manipulate data obtained from a source in various ways. For example, you could process a compressed file in a compression format not natively supported by vertica, or take UTF-16 encoded data and transcode it to UTF-8 encoding, or even perform search and replace operations on data before it is loaded into HP Vertica.

You can also pass data through multiple filters before it is loaded into HP Vertica. For instance, you could unzip a file compressed with 7Zip, convert the content from UTF-16 to UTF-8, and finally search and replace various text strings before loading the data.

Filters can optionally be used with UDSources and UDParsers. The source can be obtained using a UDSource function, passed through one or more UDFilters, and finally parsed by a UDParser before being loaded.

You must implement a UDFilter class and a FilterFactory class for your filter.

The HP Vertica SDK provides example filter functions in `/opt/vertica/sdk/examples/FilterFunctions`.

Subclassing FilterFactory

About the Filter Factory Class:

The Filter Factory class performs initial validation and planning of the query and instantiates objects to perform further initialization on each node once the query has been distributed.

FilterFactory Methods:

You implement the following methods in your FilterFactory class:

- plan() - Like the UDSource and UDParser plan() methods, the UDFilter plan() method is used to check parameters and populate the plan data. However, you cannot specify the nodes on which the work is done. HP Vertica automatically selects the best nodes to complete the work based on available resources.

  When developing your plan() method you should check the parameters that have been passed from the function call and provide a helpful error message if the arguments do not conform, but it is not required. You can also optionally populate the NodeSpecifyingPlanContext object with any information that must be passed to the other nodes doing the work from the initiator node. Finally, you must specify which nodes the source(s) are obtained from. You can split up the work so that one or multiple specific nodes load data, or specify that any node load the data.

- prepare() - This method is called on each node prior to Load operator execution. It creates the function object using the vt_createFuncObj method.

- getParameterType() - The getParameterType() method allows you to define the name and types of parameters that the function uses. HP Vertica uses this information to warn function callers that certain parameters that they provide have no effect, or that certain parameters are not being set and are reverting to default values. You should, as a best practice, define the types and parameters for your function, but using this method is optional.

The name of the factory class is the value used for the NAME modifier in the in the CREATE FILTER statement.

After creating your FilterFactory, you must register it with RegisterFactory();

Example FilterFactory

The following example is provided as part of `/opt/vertica/sdk/examples/SourceFunctions/Iconverter.cpp`. It defines the factory class for the IConverter filter function.

```
class IconverterFactory : public FilterFactory{
public:
    virtual void plan(ServerInterface &srvInterface,
            PlanContext &planCtxt) {
        std::vector<std::string> args = srvInterface.getParamReader().getParamNames();
        /* Check parameters */
        if (!(args.size() == 0 ||
                (args.size() == 1 && find(args.begin(), args.end(), "from_encoding")
                        != args.end()) || (args.size() == 2
                        && find(args.begin(), args.end(), "from_encoding") != args.end()
                        && find(args.begin(), args.end(), "to_encoding") != args.end())))
{
            vt_report_error(0, "Invalid arguments.  Must specify either no arguments,  or
"
                               "'from_encoding' alone, or 'from_encoding' and 'to_encodin
g'.");
        }
        /* Populate planData */
        // By default, we do UTF16->UTF8, and x->UTF8
        VString from_encoding = planCtxt.getWriter().getStringRef("from_encoding");
        VString to_encoding = planCtxt.getWriter().getStringRef("to_encoding");
        from_encoding.copy("UTF-16");
        to_encoding.copy("UTF-8");
        if (args.size() == 2)
        {
            from_encoding.copy(srvInterface.getParamReader().getStringRef("from_encodin
g"));
            to_encoding.copy(srvInterface.getParamReader().getStringRef("to_encoding"));
        }
        else if (args.size() == 1)
        {
            from_encoding.copy(srvInterface.getParamReader().getStringRef("from_encodin
g"));
        }
        if (!from_encoding.length()) {
            vt_report_error(0, "The empty string is not a valid from_encoding value");
        }
        if (!to_encoding.length()) {
            vt_report_error(0, "The empty string is not a valid to_encoding value");
        }
    }
    virtual UDFilter* prepare(ServerInterface &srvInterface,
            PlanContext &planCtxt) {
        return vt_createFuncObj(srvInterface.allocator, Iconverter,
                planCtxt.getReader().getStringRef("from_encoding").str(),
                planCtxt.getReader().getStringRef("to_encoding").str());
    }
    virtual void getParameterType(ServerInterface &srvInterface,
                                  SizedColumnTypes &parameterTypes) {
        parameterTypes.addVarchar(32, "from_encoding");
        parameterTypes.addVarchar(32, "to_encoding");
    }
};
RegisterFactory(IconverterFactory);
```

Subclassing UDFilter

About the UDFilter Class

The UDFilter class is responsible for reading raw input data from a source and preparing it to be loaded into HP Vertica or processed by a parser. This preparation may involve decompression, re-encoding, or any other sort of binary manipulation. A wrapper is also provided for UDFilter called ContinuousUDFilter. ContinuousUDFilter provides an abstraction that allows you to treat the input data as a continuous stream of data. This allows you to write the filtered data and process it "at will" instead of having to create an iterator to use the base UDFilter method. ContinuousUDFilter is available in `/opt/vertica/sdk/examples/HelperLibraries/ContinuousUDFilter.h`.

UDFilter Methods:

- setup() - Invoked before the first time that process() is called.

  Note: UDFilters must be restartable. If loading large numbers of files, a given UDFilter may be re-used for multiple files. HP Vertica follows the worker-pool design pattern: At the start of COPY execution, several Parsers and several Filters are instantiated per node by calling the corresponding prepare() method multiple times. Each Filter/Parser pair is then internally assigned to an initial Source (UDSource or internal). At that point, setup() is called; then process () is called until it is finished; then destroy() is called. If there are still sources in the pool waiting to be processed, then the UDFilter/UDSource pair will be given a second Source; setup() will be called a second time, then process() until it is finished, then destroy(). This repeats until all sources have been read.

- destroy() - Invoked after the last time that process() is called.

- process() - Invoked repeatedly until it returns DONE or the query is canceled by the function caller. On each invocation, process() acquires more data and writes the data to the DataBuffer specified by 'output'.

Returns:

- OUTPUT_NEEDED if this source has more data to produce.
- INPUT_NEEDED if it requires more data to continue working.
- DONE if it has no more data to produce.
- KEEP_GOING if it cannot proceed for an extended period of time. It will be called again. Do not block indefinitely. If you do, then you prevent the user from canceling the query.

Process() must set `input.offset` to the number of bytes that were successfully read from the `input` buffer, and that will not need to be re-consumed by a subsequent invocation of process(). If 'input_state' == END_OF_FILE, then the last byte in 'input' is the last byte in the input stream and returning INPUT_NEEDED does not result in any new input appearing. process() should return DONE in this case as soon as this operator has finished producing all output that it is going to produce.

process() must set `output.offset` to the number of bytes that were written to the `output` buffer. This may not be larger than `output.size`. If it is set to 0, this indicates that process() requires a larger output buffer.

ContinuousUDFilter Functions:

The ContinuousUDFilter wrapper allows you to write and process the data "at will" instead of having to iterate through the data. An example of using ContinuousUDFilter is provided in `/opt/vertica/sdk/examples/FilterFunctions/SearchAndReplaceFilter.cpp`.

- initialize() - Invoked before run(). You can optionally override this function to perform setup and initialization.
- run() - Processes the data. Use reserve() and seek(), or read() of the ContinuousReader to read (), and reserve() on the ContinuousWriter and memcopy to write the data to the output buffer.
- deinitialize() - Invoked after run() has returned. You can optionally override this function to perform tear-down and destruction.

Functions that are already implemented that you use in your code:

- yield() - use to yield control back to the server during idle or busy loops so the server can check for status changes or query cancelations.
- cr - A ContinuousReader which is defined in `/opt/vertica/sdk/examples/HelperLibraries/CoroutineHelpers.h`. Used to read from the data stream.
- cw - A ContinuousWriter which is defined in `/opt/vertica/sdk/examples/HelperLibraries/CoroutineHelpers.h`. Used to write the filtered data to the output data buffer.

Example UDFilter

The following example shows how to convert encoding for a file from one type to another. The example converts UTF-16 encoded data to UTF-8 encoded data. This example is available in the SDK at `/opt/vertica/sdk/examples/FilterFunctions/IConverter.cpp`.

```
class Iconverter : public UDFilter{
private:
    std::string fromEncoding, toEncoding;
    iconv_t cd; // the conversion descriptor opened
    uint converted; // how many characters have been converted
protected:
    virtual StreamState process(ServerInterface &srvInterface, DataBuffer &input,
                                InputState input_state, DataBuffer &output)
    {
        char *input_buf = (char *)input.buf + input.offset;
        char *output_buf = (char *)output.buf + output.offset;
        size_t inBytesLeft = input.size - input.offset, outBytesLeft = output.size - outp
ut.offset;
        // end of input
        if (input_state == END_OF_FILE && inBytesLeft == 0)
        {
            // Gnu libc iconv doc says, it is good practice to finalize the
            // outbuffer for stateful encodings (by calling with null inbuffer).
            //
            // http://www.gnu.org/software/libc/manual/html_node/Generic-Conversion-Inter
face.html
            iconv(cd, NULL, NULL, &output_buf, &outBytesLeft);
            // output buffer can be updated by this operation
            output.offset = output.size - outBytesLeft;
            return DONE;
        }
        size_t ret = iconv(cd, &input_buf, &inBytesLeft, &output_buf, &outBytesLeft);
        // if conversion is successful, we ask for more input, as input has not reached E
OF.
        StreamState retStatus = INPUT_NEEDED;
        if (ret == (size_t)(-1))
        {
            // seen an error
            switch (errno)
            {
            case E2BIG:
                // input size too big, not a problem, ask for more output.
                retStatus = OUTPUT_NEEDED;
                break;
            case EINVAL:
                // input stops in the middle of a byte sequence, not a problem, ask for m
ore input
                retStatus = input_state == END_OF_FILE ? DONE : INPUT_NEEDED;
                break;
            case EILSEQ:
                // invalid sequence seen, throw
                // TODO: reporting the wrong byte position
                vt_report_error(1, "Invalid byte sequence when doing %u-th conversion", c
onverted);
```

```
            case EBADF:
                // something wrong with descriptor, throw
                vt_report_error(0, "Invalid descriptor");
            default:
                vt_report_error(0, "Uncommon Error");
                break;
            }
        }
        else converted += ret;
        // move position pointer
        input.offset = input.size - inBytesLeft;
        output.offset = output.size - outBytesLeft;
        return retStatus;
    }
public:
    Iconverter(const std::string &from, const std::string &to)
    : fromEncoding(from), toEncoding(to), converted(0)
    {
        // note "to encoding" is first argument to iconv...
        cd = iconv_open(to.c_str(), from.c_str());
        if (cd == (iconv_t)(-1))
        {
            // error when creating converters.
            vt_report_error(0, "Error initializing iconv: %m");
        }
    }
    ~Iconverter()
    {
        // free iconv resources;
        iconv_close(cd);
    }
};
```

UDL Parser

Developing Parser Functions for User Defined Load

Parsers take a stream of bytes and pass a corresponding sequence of tuples to the HP Vertica load process. UDL Parser functions can be used to parse data in formats not understood by the HP Vertica built-in parser, or for data that require more specific control than the built-in parser supplies. For example, you could load a CSV file using a specific CSV library. Two CSV examples are provided with the HP Vertica SDK.

COPY supports a single UDL Parser that can be used in conjunction with a `UDSource` and zero or more `UDFilters`.

You must implement a `UDParser` class and a `ParserFactory` class for your parser.

You can optionally implement a `UDChunker` to organize data for parallel parsing during data load operations for delimited or fixed width data. If you do not implement the `UDChunker` class, parsing continues in single-threaded mode on one core of the node.

The HP Vertica SDK provides the following example parsers in `/opt/vertica/sdk/examples/ParserFunctions`:

Parser Name	Purpose
`BasicIntegerParser_continuous.cpp`	Parses a continuous string of integer values separated by non-numeric characters.
`BasicIntegerParser_raw.cpp`	Parses a string of integer values separated by non-numeric characters.
`ExampleDelimitedParser.cpp`	Delimited parser with the `UDChunker` class, as defined in `ExampleDelimitedChunker.cpp`.
`Rfc4180CsvParser.cpp`	RFC 4180 CSV parser
`TraditionalCsvParser.cpp`	Traditional CSV parser

The traditional CSV parser uses the `boost::tokenizer` library to read the CSV output from common programs such as Microsoft Excel. The RFC 4180 parser parses CSV files written to the RFC 4180 standard and uses libcsv.

Subclassing ParserFactory

About the ParserFactory Class

The `ParserFactory` class performs initial validation and planning for the query and instantiates objects to perform further initialization on each node once the query has been distributed. Subclasses of `ParserFactory` should be stateless,with no fields containing data, only methods.

The name of the factory class is the value used for the `NAME` modifier in the in the CREATE PARSER statement.

After creating your `ParserFactory`, you must register it with `RegisterFactory()`.

ParserFactory Methods:

You implement the following methods in your `ParserFactory` Class:

- `plan()` – Instantiates a `UDParser` instance. Like the UDSource and UDFilter `plan()` methods, this `plan()` method checks parameters and populates the plan data. You cannot specify the nodes on which the work is done. HP Vertica automatically selects the best nodes to complete the work based on available resources. The `plan()` method must not modify any global variables or state, only variables supplied as arguments.

- `prepare()` – This method is called on each node prior to Load operator execution. It creates the function object using the `vt_createFuncObj` method. The `prepare()` method must not modify any global variables or state, only variables supplied as arguments.

- `prepareChunker()` – Optional method to support parallel parsing.

- `getParserReturnType()` – This method defines the return types (and length/precision if necessary) for this UDX.

  By default, HP Vertica uses the same output column types as the destination table. This requires that the `UDParser` validate the expected output column types and emit appropriate tuples. Users can use `COPY` expressions to perform typecasting and conversion if necessary.

  Define the output types as follows:

  - For `CHAR/VARCHAR` types, specify the max length.
  - For `NUMERIC` types, specify the precision and scale.
  - For Time/Timestamp types (with or without time zone), specify the precision, where -1 means unspecified.
  - For IntervalYM/IntervalDS types, specify the precision and range.
  - For all other types, no length/precision specification is required.

- `getParameterType()` – The `getParameterType()` method lets you define the name and types of parameters that the function uses. HP Vertica uses this information to warn function callers that certain parameters that they provide have no effect, or that certain parameters are not being set and are reverting to default values. As a best practice, you should define the types and parameters for your function, but using this method is optional.

UDChunker Methods

Implement this method to have the parser participate in cooperative parsing:

- `UDChunker::prepareChunker()` – Invoked after `UDParser::prepare()` to set up the `UDChunker`.

  Returns a `UDChunker` object if the parser supports consumer/producer parallelism, or `NULL` if `UDChunker` is not implemented. :

ParserFactory Class Example

The following example is provided as part of `/opt/vertica/sdk/examples/ParserFunctions/ExampleDelimitedParser.cpp`. It defines the parser factory for the `GenericDelimitedParserFrameworkExampleFactory` parser, and includes `prepareChunker()`.

```
class GenericDelimitedParserFrameworkExampleFactory : public ParserFactory
{
public:
    virtual void plan(ServerInterface &srvInterface,
            PerColumnParamReader &perColumnParamReader,
            PlanContext &planCtxt) {
        /* Check parameters */
        // TODO: Figure out what parameters I should have; then make sure I have them

        /* Populate planData */
        // Nothing to do here
    }

    // todo: return an appropriate udchunker
    virtual UDChunker* prepareChunker(ServerInterface &srvInterface,
                                      PerColumnParamReader &perColumnParamReader,
                                      PlanContext &planCtxt,
                                      const SizedColumnTypes &returnType)
   {
        // Defaults.
        std::string delimiter(","), record_terminator("\n");
        std::vector<std::string> formatStrings;

        //return NULL;
        return vt_createFuncObject<ExampleDelimitedUDChunker>
               (srvInterface.allocator,
                delimiter[0],
                record_terminator[0],
                formatStrings
            );
    }

    virtual UDParser* prepare(ServerInterface &srvInterface,
            PerColumnParamReader &perColumnParamReader,
            PlanContext &planCtxt,
            const SizedColumnTypes &returnType)
    {
        ParamReader args(srvInterface.getParamReader());

        // Defaults.
        std::string delimiter(","), record_terminator("\n");
        std::vector<std::string> formatStrings;

        // Args.
        if (args.containsParameter("delimiter"))
            delimiter = args.getStringRef("delimiter").str();
        if (args.containsParameter("record_terminator"))
            record_terminator = args.getStringRef("record_terminator").str();

        // Validate.
        if (delimiter.size()!=1) {
            vt_report_error(0, "Invalid delimiter \"%s\": single character required",
                            delimiter.c_str());
        }
        if (record_terminator.size()!=1) {
            vt_report_error(1, "Invalid record_terminator \"%s\": single character requir
ed",
                            record_terminator.c_str());
        }

        // Extract the "format" argument.
        // Default to the global setting, but let any per-column settings override for th
at column.
        if (args.containsParameter("format"))
            formatStrings.resize(returnType.getColumnCount(), args.getStringRef("format")
.str());
        else
            formatStrings.resize(returnType.getColumnCount(), "");

        for (size_t i = 0; i < returnType.getColumnCount(); i++) {
            const std::string &cname(returnType.getColumnName(i));
            if (perColumnParamReader.containsColumn(cname)) {
                ParamReader &colArgs = perColumnParamReader.getColumnParamReader(cname);
                if (colArgs.containsParameter("format")) {
                    formatStrings[i] = colArgs.getStringRef("format").str();
                }
            }
        }

        return vt_createFuncObject<DelimitedParserFrameworkExample<StringParsersImpl> >
               (srvInterface.allocator,
                delimiter[0],
                record_terminator[0],
                formatStrings
            );
    }

        virtual void getParserReturnType(ServerInterface &srvInterface,
            PerColumnParamReader &perColumnParamReader,
            PlanContext &planCtxt,
            const SizedColumnTypes &argTypes,
            SizedColumnTypes &returnType)
    {
        returnType = argTypes;
    }

   virtual void getParameterType(ServerInterface &srvInterface,
                                 SizedColumnTypes &parameterTypes) {
        parameterTypes.addVarchar(1, "delimiter");
        parameterTypes.addVarchar(1, "record_terminator");
        parameterTypes.addVarchar(256, "format");
    }
};

typedef GenericDelimitedParserFrameworkExampleFactory<StringParsers> DelimitedParserFrame
workExampleFactory;
RegisterFactory(DelimitedParserFrameworkExampleFactory);
```

Subclassing UDParser

About the UDParser Class

The `UDParser` Class is responsible for parsing an input stream into tuples/rows for insertion into an HP Vertica table. A wrapper is also provided for `UDParser` called `ContinuousUDParser`. ContinuousUDParser provides an abstraction that allows you to treat the input data as a continuous stream of data. This allows you to read from the data and process it *at will*, instead of having to create an iterator to use the base `UDParser` method. `ContinuousUDParser` is available in `/opt/vertica/sdk/examples/HelperLibraries/ContinuousUDParser.h`.

UDParser Methods:

- `setup()` – Invoked before the first time that `process()` is called.

  UDParsers must be restartable. If loading large numbers of files, a given UDParser may be re-used for multiple files. HP Vertica follows the worker-pool design pattern: At the start of COPY execution, several Parsers and several Filters are instantiated per node by calling the corresponding `prepare()` method multiple times. Each Filter/Parser pair is then internally assigned to an initial Source (UDSource or internal). At that point,`setup()` is called; then `process()` is called until it is finished, after which `destroy()` is called. If sources to be processed in the pool are still waiting, then the `UDFilter/UDSource` pair will be given a second chance. The `UDSourcesetup()` is called a second time, then `process()` until it is finished, then `destroy()`. This process repeats until all sources have been read.

- `process()` – Invoked repeatedly during query execution until it returns `DONE` or until the query is canceled by the user.

  On each invocation, `process()` is given an input buffer. The method reads data from that buffer, converting it to fields and tuples. The tuples are written through a *writer*.

  Once `process()` has consumed as much data as is reasonable (for example, the last complete row in the input buffer), process() returns oneshould return `INPUT_NEEDED` to indicate that it requires more data, or `DONE` to indicate that it has completed parsing this input stream and will not be reading more bytes from it.

  If `input_state == END_OF_FILE`, then the last byte in `input` is the last byte in the input stream. Returning `INPUT_NEEDED` does not result in any new input appearing. The `process()` should return `DONE` in this case as soon as this operator has finished producing all output that it is going to produce.

Note that `input` can contain null bytes, if the source file contains them. Note also that `input` is NOT automatically null-terminated.

Returns:

- `INPUT_NEEDED` if this UDParser has more data to produce.
- `DONE` if it has no more data to produce.
- `REJECT` to reject a row (see "Row Rejection" below)

- `destroy()` – Invoked after the last time that `process()` is called.
- `getRejectedRecord()` – returns information about the rejected data.
- `writer` – A member variable of the class, type `StreamWriter`, and used to write parsed tuples to. This is the same API as `PartitionWriter` used in the User Defined Transforms framework.

Row Rejection

To reject some data, there parser needs to do two things:

- Create a `getRejectedRecord()` method on your Parser class that returns an object of type `Vertica::RejectedRecord`, which contains the data that you want to reject, a string describing the reason for rejection, the size of the data, and the terminator string. See `RejectedRecord` in VerticaUDl.h for details or view the SDK Documentation.
- When the parser encounters a rejected data, have process() return `REJECT`. After returning `REJECT`, HP Vertica calls `getRejectedRecord()` to process the rejected record before the next call to `process()`.

One simple way of fulfilling this is to include code in your parser class such as:

```
Vertica::RejectedRecord myRejRec;
Vertica::RejectedRecord getRejectedRecord() {
   return myRejRec;
}
```

In your process() method, add code such as:

```
    (...)         if (some rejection condition) {
        RejectedRecord rr("Bad Record!", "foo data", 8, "\n");
        myRejRec = rr;
        return Vertica::REJECT;
    }
    (...)
```

That is just one simple approach. The only requirement is that there exist a getRejectedRecord() function that can (and always will) be called after process() returns REJECT, and it returns the data that HP Vertica needs to process the rejection.

ContinuousUDParser Functions:

The `ContinuousUDParser` wrapper allows you to read and process the data stream *at will* instead of having to iterate through the data.

- `initialize()` – Invoked before run(). You can optionally override this function to perform setup and initialization.
- `run()` – Processes the data. Use reserve() and seek(), or read() to read the data and the `writer` StreamWriter object to write data to HP Vertica.
- `deinitialize()` – Invoked after run() has returned. You can optionally override this function to perform tear-down and destruction.

Functions that are already implemented that you use in your code:

- `yield()` – Used to yield control back to the server during idle or busy loops so the server can check for status changes or query cancellations.
- `cr` – A `ContinuousReader` used to read from the data stream and defined in `/opt/vertica/sdk/examples/HelperLibraries/CoroutineHelpers.h`.
- `crej` – A `ContinuousReader`used to manage rejected rows, and defined in `/opt/vertica/sdk/examples/HelperLibraries/CoroutineHelpers.h`.

UDParser Class Example:

The following example parses a single column of integers using the `ContinuousUDParser` and writes them to tuples using the `writer` object. This example uses the `ContinuousUDParser` wrapper.

```
class BasicIntegerParser : public ContinuousUDParser {private:
    // campaign for the conservation of keystrokes
    char *ptr(size_t pos = 0) { return ((char*)cr.getDataPtr()) + pos; }
    vint strToInt(const string &str) {
        vint retVal;
        stringstream ss;
        ss << str;
        ss >> retVal;
        return retVal;
    }
public:
    virtual void run() {
        // WARNING: This implementation is not trying for efficiency.
        // It is trying to exercise ContinuousUDParser,
        // and to be quick to implement.
        // This parser assumes a single-column input, and
        // a stream of ASCII integers split by non-numeric characters.
        size_t pos = 0;
        size_t reserved = cr.reserve(pos+1);
        while (!cr.isEof() || reserved == pos+1) {
            while (reserved == pos+1 && (*ptr(pos) >= '0' && *ptr(pos) <= '9')) {
                pos++;
                reserved = cr.reserve(pos+1);
            }
            string st(ptr(), pos);
            writer->setInt(0, strToInt(st));
            writer->next();
            while (reserved == pos+1 && !(*ptr(pos) >= '0' && *ptr(pos) <= '9')) {
                pos++;
                reserved = cr.reserve(pos+1);
            }
            cr.seek(pos);
            pos = 0;
            reserved = cr.reserve(pos+1);
        }
    }
};
```

Subclassing UDChunker

About the UDChunker Class

The `UDChunker` Class is responsible for separating parsing record boundaries.

UDChunker Methods:

- `UDChunker::setup()` – Invoked before the first time that `process()` is called.
- `UDChunker::process()` – Invoked repeatedly during query execution until it returns `DONE` or until the query is canceled by the user.
- `UDChunker::destroy()` – Invoked after the last time that `process()` is called. You can override this method to perform tear-down and destructive tasks. Recall that `UDChunkers` must be restartable.

On each invocation, `process()` is given an input buffer. The method should read data from that buffer, find record boundaries, and align the `input.offset` with the end of the last record in the buffer. Once `process()` has consumed as much data as is reasonable (for example, the last complete row in the input buffer), `process()` should return one of these values:

- `OUTPUT_NEEDED` to indicate that it requires more buffer size
- `INPUT_NEEDED` to indicate more data is required.
- `DONE` to indicate that it has completed parsing this input stream and will not be reading more bytes from it.

If `input_state == END_OF_FILE`, then the last byte in `input` is the last byte in the input stream. Returning `INPUT_NEEDED` does not result in any new input appearing. The `process()` should return `DONE` in this case as soon as this operator has finished producing all output that it is going to produce.

Note: The `input` data can contain null bytes, if the source file contains them. Note also that `input` is NOT automatically null-terminated.

The `UDChunker::process()` method must not block indefinitely. If it is cannot proceed for an extended period of time, it should return `KEEP_GOING`, after which it will be called again shortly. Failing to return `KEEP_GOING` has several consequences, including preventing the user from being able to cancel the query.

UDChunker Class Example:

The following example illustrates an implemented `UDChunker::process()` method. The source code is available at:

`/opt/vertica/sdk/examples/ParserFunctions/ExampleDelimitedChunker.cpp`

```
ExampleDelimitedUDChunker::ExampleDelimitedUDChunker(char delimiter = ',',
                                                     char recordTerminator = '\n',
                                                     std::vector<std::string> formatStrin
gs =
                                                     std::vector<std::string>()) : delimi
ter(delimiter),
                                                     recordTerminator(recordTerminator),
                                                     formatStrings(formatStrings) {}

StreamState ExampleDelimitedUDChunker::process(ServerInterface &srvInterface,
                                               DataBuffer &input,
                                               InputState input_state)
{
    size_t termLen = 1;
    char* terminator = &recordTerminator;

    size_t ret = input.offset, term_index = 0;
    for (size_t index = input.offset; index < input.size; ++index) {
        char c = input.buf[index];
        if (c == terminator[term_index])
        {
            ++term_index;
            if (term_index == termLen)
            {
                ret = index + 1;
                term_index = 0;
            }
            continue;
        }
        else if (term_index > 0)
            index -= term_index;

        term_index = 0;
    }
    // if we were able to find some rows, move the offset to point at the start of the ne
xt (potential) row, or end of block
    if (ret > input.offset) {
        input.offset = ret;
        return OUTPUT_NEEDED;
    }

    if (input_state == END_OF_FILE) {
        input.offset = input.size;
        return DONE;
    }

    return INPUT_NEEDED;
}
```

Developing UDLs in Java

The HP Vertica Java SDK support developing UDLs. If you have not already done so, you need to configure your database hosts to run Java User Defined Extensions (UDxs). For instructions see:

- Installing Java on HP Vertica Hosts
- Configuring Your Java Development Environment

Developing User Defined Source Functions

You create UDL Source functions (referred to as UDSource or UDS) to process a source of data that is not natively supported by the COPY statement. For example, you can create a UDL that accesses data from a web server using a RESTful web API. You can use a single UDSource function in a COPY statement, but that source function can pull data from multiple sources (reading files from multiple URLs, for example).

You can use a UDSource function in a COPY statement with UDFilter functions, a UDParser function, or the built-in filtering and parsing feature of the COPY statement.

To create a UDSource function, you must subclass both the UDSource and SourceFactory classes.

The HP Vertica Java SDK provides an example source function in `/opt/vertica/sdk/examples/JavaUDx/UDLFuctions`. The example explained in this section is based on the example code provided with the SDK.

The following sections demonstrate how to create a simple UDSource function that loads files from the host's filesystem, similar to how the COPY statement natively loads files.

UDSource Example Overview

The example shown in the following sections is a simple UDL Source function named `FileSource` that loads data from files stored on the host's filesystem (similar to the standard COPY statement). To call it, you must supply a parameter named `file` that contains the absolute path to one or more files on the host filesystem. You can specify multiple files as a comma-separated list.

The `FileSource` function also accepts an optional parameter named `nodes` that indicates which nodes should load the files. If you do not supply this parameter, the function defaults to loading data on the initiator host only. Since this is a simple example, the nodes only load the files off of their own file system. Any files in the file parameter must exist on all of the hosts in the nodes parameter. The `FileSource` UDSource attempts to load all of the files in the `file` parameter on all of the hosts in the `nodes` parameter.

You can use the following Python script to generate files and distribute them to hosts in your HP Vertica cluster to experiment with the example UDSource function. You run it using the database administrator account on one of your database hosts, as it requires passwordless-SSH logins in order to copy the files to the other hosts.

```
#!/usr/bin/python
# Save this file as UDLDataGen.py
import string
import random
import sys
import os

# Read in the dictionary file to provide random words. Assumes the words
# file is located in /usr/share/dict/words
wordFile = open("/usr/share/dict/words")
wordDict = []
for line in wordFile:
    if len(line) > 6:
        wordDict.append(line.strip())

MAXSTR = 4 # Maximum number of words to concatentate
NUMROWS = 1000 # Number of rows of data to generate
#FILEPATH = '/tmp/UDLdata.txt' # Final filename to use for UDL source
TMPFILE = '/tmp/UDLtemp.txt'  # Temporary filename.

# Generate a random string by concatenating several words together. Max
# number of words set by MAXSTR
def randomWords():
    words = [random.choice(wordDict) for n in xrange(random.randint(1, MAXSTR))]
    sentence = " ".join(words)
    return sentence

# Create a temporary data file that will be moved to a node. Number of
# rows for the file is set by NUMROWS. Adds the name of the node which will
# get the file, to show which node loaded the data.
def generateFile(node):
    outFile = open(TMPFILE, 'w')
    for line in xrange(NUMROWS):
        outFile.write('{0}|{1}|{2}\n'.format(line,randomWords(),node))
    outFile.close()
```

```
# Copy the temporary file to a node. Only works if passwordless SSH login
# is enabled, which it is for the database administrator account on
# HP Vertica hosts.
def copyFile(fileName,node):
    os.system('scp "%s" "%s:%s"' % (TMPFILE, node, fileName) )

# Loop through the comma-separated list of nodes given in the first
# parameter, creating and copying data files whose full comma-separated
# paths are passed in the second parameter
for node in [x.strip() for x in sys.argv[1].split(',')]:
    for fileName in [y.strip() for y in sys.argv[2].split(',')]:
        print "generating file", fileName, "for", node
        generateFile(node)
        print "Copying file to",node
        copyFile(fileName,node)
```

You call this script by giving it a comma-separated list of hosts to receive the files, and a comma-separated list of absolute paths of files to generate. For example:

```
python UDLDataGen.py node01,node02,node03 /tmp/UDLdata01.txt,/tmp/UDLdata02.txt,\
UDLdata03.txt
```

This script generates files that contain a thousand rows of pipe character (|) delimited columns: an index value, a set of random words, and the node for which the file was generated. The output files look like this:

```
0|megabits embanks|node01
1|unneatly|node01
2|self-precipitation|node01
3|antihistamine scalados Vatter|node01
```

The following example demonstrates loading and using the `FileSource` UDSource:

```
=> --Load library and create the source function
=> CREATE LIBRARY JavaLib AS '/home/dbadmin/JavaUDlLib.jar'
-> LANGUAGE 'JAVA';
CREATE LIBRARY
=> CREATE SOURCE File as LANGUAGE 'JAVA' NAME
-> 'com.mycompany.UDL.FileSourceFactory' LIBRARY JavaLib;
CREATE SOURCE FUNCTION
=> --Create a table to hold the data loaded from files
=> CREATE TABLE t (i integer, text VARCHAR, node VARCHAR);
CREATE TABLE
=> -- Copy a single file from the currently host using the FileSource
=> COPY t SOURCE File(file='/tmp/UDLdata01.txt');
 Rows Loaded
-------------
        1000
(1 row)

=> --See some of what got loaded.
=> SELECT * FROM t WHERE i < 5 ORDER BY i;
```

```
 i |             text              |  node
---+-------------------------------+--------
 0 | megabits embanks              | node01
 1 | unneatly                      | node01
 2 | self-precipitation            | node01
 3 | antihistamine scalados Vatter | node01
 4 | fate-menaced toilworn         | node01
(5 rows)



=> TRUNCATE TABLE t;
TRUNCATE TABLE
=> -- Now load a file from three hosts. All of these hosts must have a file
=> -- named /tmp/UDLdata01.txt, each with different data
=> COPY t SOURCE File(file='/tmp/UDLdata01.txt',
-> nodes='v_vmart_node0001,v_vmart_node0002,v_vmart_node0003');
 Rows Loaded
-------------
        3000
(1 row)

=> --Now see what has been loaded
=> SELECT * FROM t WHERE i < 5 ORDER BY i,node ;
 i |                      text                       |  node
---+-------------------------------------------------+--------
 0 | megabits embanks                                | node01
 0 | nimble-eyed undupability frowsier               | node02
 0 | Circean nonrepellence nonnasality               | node03
 1 | unneatly                                        | node01
 1 | floatmaker trabacolos hit-in                    | node02
 1 | revelrous treatableness Halleck                 | node03
 2 | self-precipitation                              | node01
 2 | whipcords archipelagic protodonatan copycutter  | node02
 2 | Paganalian geochemistry short-shucks            | node03
 3 | antihistamine scalados Vatter                   | node01
 3 | swordweed touristical subcommanders desalinized | node02
 3 | batboys                                         | node03
 4 | fate-menaced toilworn                           | node01
 4 | twice-wanted cirrocumulous                      | node02
 4 | doon-head-clock                                 | node03
(15 rows)

=> TRUNCATE TABLE t;
TRUNCATE TABLE
=> --Now copy from several files on several hosts
=> COPY t SOURCE File(file='/tmp/UDLdata01.txt,/tmp/UDLdata02.txt,/tmp/UDLdata03.txt'
-> ,nodes='v_vmart_node0001,v_vmart_node0002,v_vmart_node0003');
 Rows Loaded
-------------
        9000
(1 row)

=> SELECT * FROM t WHERE i = 0 ORDER BY node ;
 i |                    text                     |  node
---+---------------------------------------------+--------
 0 | Awolowo Mirabilis D'Amboise                 | node01
 0 | sortieing Divisionism selfhypnotization     | node01
 0 | megabits embanks                          | node01
 0 | nimble-eyed undupability frowsier         | node02
 0 | thiaminase hieroglypher derogated soilborne | node02
 0 | aurigraphy crocket stenocranial           | node02
 0 | Khulna pelmets                            | node03
 0 | Circean nonrepellence nonnasality         | node03
 0 | matterate protarsal                       | node03
(9 rows)
```

The following sections explain how to create the `FileSource` UDSource fuinction.

Subclassing SourceFactory in Java

Your subclass of the HP Vertica SDK's `SourceFactory` class to is responsible for:

- performing the initial validation of the parameters in the function call your `UDSource` function.
- setting up any data structures your `UDSource` subclass instances will need to perform their work. This information can include recording which nodes will read which data source.
- creating one instance of your `UDSource` subclass for each data source your function will read from on each host.

SourceFactory Methods

The `SourceFactory` class defines the following methods your subclass can override. You class must override `prepareUDSources()`.

- `plan()` - HP Vertica calls this method once on the initiator node. It should perform the following tasks:
  - Check the parameters the user supplied to the function call in the COPY statement and provide a helpful error message if there are any issues. It reads the parameters by getting a `ParamReader` object from the instance of `ServerInterface` passed into the `plan()` method.
  - Decide which hosts in the cluster will read the data source. How you divide up the work depends on the source your function is reading. Some sources can easily be split across many hosts (such as reading data from many URLs, or from a RESTful API which allows you to segment data). Others, such an individual local file on a host's filesystem, can only be read by a single specific host.

    You store the list of hosts to read the data source by calling the `setTargetNodes()` method on the `NodeSpecifyingPlanContext` object passed into your `plan()` method.
  - Store any information that the individual hosts need in order to process the data sources in the `NodeSpecifyingPlanContext` instance passed in the `planCtxt` parameter. For example, you could store assignments that tell each host which data sources to process. This object is the only means of communication between the `plan()` method (which only runs on the

initiator node) and the `prepareUDSources()` method (which runs on each host reading from a data source).

You store data in the `NodeSpecifyingPlanContext` by getting a `ParamWriter` object from the `getWriter()` method. You then write parameters by calling methods on the `ParamWriter` such as `setString()`.

> **Note:** `ParamWriter` only offers the ability to store simple data types. For complex types, you will need to serialize the data in some manner and store it as a string or long string.

- `prepareUDSources()` - HP Vertica calls this method on all hosts that were chosen to load data by the `plan()` method. It instantiates one or more of your subclass of the `UDSource` class (one for each of the sources that the host has been assigned to process), returning it in an `ArrayList`.
- `getParameterType()` - defines the name and types of parameters that your function uses. HP Vertica uses this information to warn function callers that any unknown parameters that they provide will have no effect, or that parameters they did not provide will use default values. You should define the types and parameters for your function, but overriding this method is optional.

Users will supply the name of your subclass of `SourceFactory` to the CREATE SOURCE statement when defining your UDSource function in the HP Vertica catalog, so you should choose a logical name for it.

Example SourceFactory Subclass

The following example a modified version of the example Java UDsource function provided in the Java UDx support package, located at `/opt/vertica/sdk/examples/JavaUDx/UDLFuctions/com/vertica/JavaLibs/FileSourceFactory.java`. Its override of the `plan()` method verifies that the user supplied the required `file` parameter. If the user also supplied the optional nodes parameter, this method ensures that the nodes exist in the Vertica Analytics Platform cluster. If there is a problem with either parameter, the method throws an exception to return an error to the user. If there are no issues with the parameters, the `plan()` method stores their values in the plan context object.

```
package com.mycompany.UDL;

import java.util.ArrayList;
import java.util.Vector;
import com.vertica.sdk.NodeSpecifyingPlanContext;
import com.vertica.sdk.ParamReader;
import com.vertica.sdk.ParamWriter;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.SizedColumnTypes;
import com.vertica.sdk.SourceFactory;
import com.vertica.sdk.UDSource;
import com.vertica.sdk.UdfException;
```

```
public class FileSourceFactory extends SourceFactory {

    // Called once on the initiator host to do initial setup. Checks
    // parameters and chooses which nodes will do the work.
    @Override
    public void plan(ServerInterface srvInterface,
            NodeSpecifyingPlanContext planCtxt) throws UdfException {

        String nodes; // stores the list of nodes that will load data

        // Get  copy of the parameters the user supplied to the UDSource
        // function call.
        ParamReader args =  srvInterface.getParamReader();

        // A list of nodes that will perform work. This gets saved as part
        // of the plan context.
        ArrayList<String> executionNodes = new ArrayList<String>();

        // First, ensure the user supplied the file parameter
        if (!args.containsParameter("file")) {
            // Withut a file parameter, we cannot continue. Throw an
            // exception that will be caught by the Java UDx framework.
            throw new UdfException(0, "You must supply a file parameter");
        }

        // If the user specified nodes to read the file, parse the
        // comma-separated list and save. Otherwise, assume just the
        // Initiator node has the file to read.
        if (args.containsParameter("nodes")) {
            nodes = args.getString("nodes");

            // Get list of nodes in cluster, to ensure that the node the
            // user specified actually exists. The list of nodes is available
            // from the planCTxt (plan context) object,
            ArrayList<String> clusterNodes = planCtxt.getClusterNodes();

            // Parse the string parameter "nodes" which
            // is a comma-separated list of node names.
            String[] nodeNames = nodes.split(",");

            for (int i = 0; i < nodeNames.length; i++){
                // See if the node the user gave us actually exists
                if(clusterNodes.contains(nodeNames[i]))
                    // Node exists. Add it to list of nodes.
                    executionNodes.add(nodeNames[i]);
                else{
                    // User supplied node that doesn't exist. Throw an
                    // exception so the user is notified.
                    String msg = String.format("Specified node '%s' but no" +
                        " node by that name is available.  Available nodes "
                        + "are \"%s\".",
                        nodeNames[i], clusterNodes.toString());
                    throw new UdfException(0, msg);
                }
            }
        } else {
            // User did not supply a list of node names. Assume the initiator
            // is the only host that will read the file. The srvInterface
            // instance passed to this method has a getter for the current
            // node.
            executionNodes.add(srvInterface.getCurrentNodeName());
        }

        // Set the target node(s) in the plan context
        planCtxt.setTargetNodes(executionNodes);

        // Set parameters for each node reading data that tells it which
        // files it will read. In this simple example, just tell it to
        // read all of the files the user passed in the file parameter
        String files = args.getString("file");

        // Get object to write parameters into the plan context object.
        ParamWriter nodeParams = planCtxt.getWriter();

        // Loop through list of execution nodes, and add a parameter to plan
        // context named for each node performing the work, which tells it the
        // list of files it will process. Each node will look for a
        // parameter named something like "filesForv_vmart_node0002" in its
        // prepareUDSources() method.
        for (int i = 0; i < executionNodes.size(); i++) {
            nodeParams.setString("filesFor" + executionNodes.get(i), files);
        }
    }

    // Called on each host that is reading data from a source. This method
    // returns an array of UDSource objects that process each source.
    @Override
    public ArrayList<UDSource> prepareUDSources(ServerInterface srvInterface,
            NodeSpecifyingPlanContext planCtxt) throws UdfException {

        // An array to hold the UDSource subclasses that we instaniate
        ArrayList<UDSource> retVal = new ArrayList<UDSource>();

        // Get the list of files this node is supposed to process. This was
        // saved by the plan() method in the plancontext
        String myName = srvInterface.getCurrentNodeName();
        ParamReader params = planCtxt.getReader();
        String fileNames = params.getString("filesFor" + myName);

        // Note that you can also be lazy and directly grab the parameters
        // the user passed to the UDSource functon in the COPY statement directly
        // by getting parameters from the ServerInterface object. I.e.:

        //String fileNames = srvInterface.getParamReader().getString("file");

        // Split comma-separated list into a single list.
        String[] fileList = fileNames.split(",");
        for (int i = 0; i < fileList.length; i++){
            // Instantiate a FileSource object (which is a subclass of UDSource)
            // to read each file. The constructor for FileSource takes the
            // file name of the
            retVal.add(new FileSource(fileList[i]));
        }

        // Return the collection of FileSource objects. They will be called,
        // in turn, to read each of the files.
        return retVal;
    }

    // Declares which parameters that this factory accepts.
    @Override
    public void getParameterType(ServerInterface srvInterface,
                                SizedColumnTypes parameterTypes) {
        parameterTypes.addVarchar(65000, "file");
        parameterTypes.addVarchar(65000, "nodes");
    }
}
```

Subclassing UDSource in Java

Your subclass of the `UDSource` class is responsible for reading data from a single external source and producing a data stream that the next stage in the data load process consumes. Your `SourceFactory.prepareUDSources()` method instantiates a member of this subclass for each data source that a host has been requested to read. Each instance of your `UDSource` subclass reads from a single data source. Examples of a single data source are a single file, or the results of a single function call to a RESTful web application.

UDSource Methods

The `UDSource` class has the following methods your subclass can override. It must override the `process()` method.

- `setup()` - performs any necessary setup steps to access the data source. This method establishes network connections, opens files, and other similar initial tasks that need to be performed before the `UDSource` instance can read data from the data source.

- `process()` - reads data from the source and place it into the `buf` field on an instance of the `DataBuffer` it recievd as a parameter. If it runs out of input or fills the output buffer, it must return the value `StreamState.OUTPUT_NEEDED`.When HP Vertica gets this return value, it will call the method again after the output buffer has been processed by the next stage in the data load process. When `process()` reads the last chunk of data from the data source, it returns `StreamState.DONE` to indicate that all of the data has been read from the source.

- `destroy()` - performs whatever cleanup operations are necessary after the UDSource has finished reading the data source or when the query has been canceled. It frees resources such as file handles or network connections that the `setup()` method allocated.

- `getSize()` - estimates the amount of in bytes data that the UDSource will read from the data source. Vertica Analytics Platform may call this method before it calls `setup()`. Therefore `getSize()` must not rely on any resources that `setup()` allocates. See `UDSource` in the SDK documentation for additional important details about the `getSize()` method. Overriding this method is optional.

Caution: This method should not leave any resources open. For example, do not save any file handles opened by `getSize()` for use by the `process()` method. Doing so can lead to exhausting the available resources, since HP Vertica calls `getSize()` on all instances of your `UDSource` subclass before any data is loaded. If many data sources are being opened, these open file handles could use up the system's supply of file handles, leaving none free to perform the actual data load.

- `getUri()` - returns the URI of the data source being read by this `UDSource`. This value is used to update status information to indicate which resources are currently being loaded. Overriding this method is optional.

Example UDSource

The following example shows the source of the `FileSource` class that reads a file from the host filesystem. The constructor for the class (which is called by `FileSourceFactory.prepareUDSources`, see Subclassing SourceFactory in Java) gets the absolute path for the file containing the data to be read. The `setup()` method opens the file, and the `destroy()` method closes it. The `process()` method reads from the file into a buffer provided by the instance of the `DataBuffer` class passed to it as a parameter. If the read operation filled the output buffer, it returns `OUTPUT_NEEDED`. This value tells Vertica Analytics Platform to call it again after the next stage of the load has processed the output buffer. If the read did not fill the output buffer, then `process()` returns DONE to indicate it has finished processing the data source.

```
package com.mycompany.UDL;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.RandomAccessFile;

import com.vertica.sdk.DataBuffer;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.State.StreamState;
import com.vertica.sdk.UDSource;
import com.vertica.sdk.UdfException;

public class FileSource extends UDSource {

    private String filename;  // The file for this UDSource to read
    private RandomAccessFile reader;   // handle to read from file


    // The constructor just stores the absolute filename of the file it will
    // read.
    public FileSource(String filename) {
        super();
        this.filename = filename;
    }
    // Called before HP Vertica starts requesting data from the data source.
    // In this case, setup needs to open the file and save to the reader
    // property.
    @Override
    public void setup(ServerInterface srvInterface ) throws UdfException{
        try {
            reader = new RandomAccessFile(new File(filename), "r");
        } catch (FileNotFoundException e) {
            // In case of any error, throw a UDfException. This will terminate
            // the data load.
             String msg = e.getMessage();
             throw new UdfException(0, msg);
        }
    }

    // Called after data has been loaded. In this case, close the file handle.
    @Override
    public void destroy(ServerInterface srvInterface ) throws UdfException {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                String msg = e.getMessage();
                 throw new UdfException(0, msg);
            }
        }
    }

    @Override
    public StreamState process(ServerInterface srvInterface, DataBuffer output)
                                throws UdfException {

        // Read up to the size of the buffer provided in the DataBuffer.buf
        // property. Here we read directly from the file handle into the
        // buffer.
        long offset;
        try {
            offset = reader.read(output.buf,output.offset,
                                 output.buf.length-output.offset);

        } catch (IOException e) {
            // Throw an exception in case of any errors.
            String msg = e.getMessage();
            throw new UdfException(0, msg);
        }

        // Update the number of bytes processed so far by the data buffer.
        output.offset +=offset;

        // See end of data source has been reached, or less data was read
        // than can fit in the buffer
        if(offset == -1 || offset < output.buf.length) {
            // No more data to read.
            return StreamState.DONE;
        }else{
            // Tell HP Vertica to call again when buffer has been emptied
            return StreamState.OUTPUT_NEEDED;
        }
    }
}
```

Developing Filter Functions in Java

UDL Filter functions (often referred to as UDFilters) manipulate data read from a data source. For example, a UDFilter can decompress data in a compression format not natively supported by HP Vertica, or take UTF-16 encoded data and transcode it to UTF-8 encoding, or perform search and replace operations on data before it is loaded into HP Vertica.

You can use multiple filters in a single COPY statement. For instance, you could unzip a file compressed with 7Zip, convert the content from UTF-16 to UTF-8, and finally search and replace various data before passing the data on to the parser stage of the load process.

UDFilters work with UDSource and UDParser functions as well as the native data source and parser in the COPY statement.

To create a UDFilter function, you must implement subclasses of the `UDFilter` and `FilterFactory` classes.

The HP Vertica Java SDK provides an example filter function in `/opt/vertica/sdk/examples/JavaUDx/UDLFuctions`. The example explained in this section was derived from the sample in the SDK.

Java UDL Filter Example Overview

The examples the following sections demonstrate creating a UDFilter that replaces any occurrences of a character in the input stream with another character in the output stream. This example is highly simplified, and assumes the input stream is ASCII data. You should always remember that the input and output stream in a UDFilter is actually binary data. If you are performing character transformations using a UDFilter, you should converts the data stream from a string of bytes into a properly encoded string. For example, if the input stream consists of UTF-8 encoded text, you should be sure to transform the raw binary being read from the buffer into a UTF string before manipulating it.

The example UDFilter has two required parameters: from_char specifies the character to be replaced, and to_char specifies the replacement character. The following example demonstrates loading the UDFilter and filtering several lines of data.

```
=> CREATE LIBRARY JavaLib AS '/home/dbadmin/JavaUDlLib.jar'
->LANGUAGE 'JAVA';
CREATE LIBRARY
=> CREATE FILTER ReplaceCharFilter as LANGUAGE 'JAVA'
->name 'com.mycompany.UDL.ReplaceCharFilterFactory' library JavaLib;
CREATE FILTER FUNCTION
=> CREATE TABLE t (text VARCHAR);
CREATE TABLE
=> COPY t FROM STDIN WITH FILTER ReplaceCharFilter(from_char='a', to_char='z');
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
```

```
>> Mary had a little lamb
>> a man, a plan, a canal, Panama
>> \.

=> SELECT * FROM t;
              text
--------------------------------
 Mzry hzd z little lzmb
 z mzn, z plzn, z cznzl, Pznzmz
(2 rows)

=> --Calling the filter with incorrect parameters returns errors
=> COPY t from stdin with filter ReplaceCharFilter();
ERROR 3399:  Failure in UDx RPC call InvokePlanUDL(): Error in User Defined Object [
ReplaceCharFilter], error code: 0
com.vertica.sdk.UdfException: You must supply two parameters to ReplaceChar: 'from_char'
and 'to_char'
        at com.vertica.JavaLibs.ReplaceCharFilterFactory.plan(ReplaceCharFilterFactory.ja
va:22)
        at com.vertica.udxfence.UDxExecContext.planUDFilter(UDxExecContext.java:889)
        at com.vertica.udxfence.UDxExecContext.planCurrentUDLType(UDxExecContext.java:86
5)
        at com.vertica.udxfence.UDxExecContext.planUDL(UDxExecContext.java:821)
        at com.vertica.udxfence.UDxExecContext.run(UDxExecContext.java:242)
        at java.lang.Thread.run(Thread.java:662)
```

Subclassing FilterFactory in Java

Your subclasse of the `Filter Factory` class performs the initial validation and planning of the function execution and instantiates `UDFilter` objects on each host that will be filtering data.

FilterFactory Methods

The `FilterFactory` class has the following methods that you can override in your subclass. You must override `prepare()`.

- `plan()` - HP Vertica calls this method once on the initiator node. It should perform the following tasks:

  - Check the parameters that have been passed from the function call in the COPY statement and provide a helpful error message if there are any issues. You read the parameters by getting a `ParamReader` object from the instance of `ServerInterface` passed into your `plan()` method.

  - Store any information that the individual hosts need in order to filter data in the `PlanContext` instance passed in the `planCtxt` parameter. For example, you could store details of the input format the filter will read and output the format that the filter should produce. This object is the only means of communication between the `plan()` method (which only runs on the initiator node) and the `prepare()` method (which runs on each host reading from a data source).

You store data in the `PlanContext` by getting a `ParamWriter` object from the `getWriter()` method. You then write parameters by calling methods on the `ParamWriter` such as `setString`.

Note: `ParamWriter` only offers the ability to store simple data types. For complex types, you need to serialize the data in some manner and store it as a string or long string.

- `prepare()` - instantiates a member of your `UDFilter` subclass to perform the data filtering. It can pass information your filter will need to perform its work (such as the values of the parameters the user supplied) in the call to your `UDSource` subclass's constructor.
- `getParameterType()` - defines the name and types of parameters that your function uses. HP Vertica uses this information to warn function callers that any unknown parameters that they provide will have no effect, or that parameters they did not provide will use default values. You should define the types and parameters for your function, but overriding this method is optional.

Users will supply the name of your subclass of `FilterFactory` to the CREATE FILTER statement when defining your function in the HP Vertica catalog, so you should choose a logical name for it.

Example FilterFactory

The following example subclass of `FilterFactory` named `ReplaceCharFilterFactory` requires two parameters (`from_char` and `to_char`). The `plan()` method ensures the function call contained these parameters, and tha they are single-character strings. It then stores them in the plan context. The `prepare()` method gets the parameter values and passes them to the `ReplaceCharFilter` objects it instantiates to perform the filtering.

```
package com.vertica.JavaLibs;

import java.util.ArrayList;
import java.util.Vector;

import com.vertica.sdk.FilterFactory;
import com.vertica.sdk.PlanContext;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.SizedColumnTypes;
import com.vertica.sdk.UDFilter;
import com.vertica.sdk.UdfException;

public class ReplaceCharFilterFactory extends FilterFactory {

    // Run on the initiator node to perform varification and basic setup.
    @Override
    public void plan(ServerInterface srvInterface,PlanContext planCtxt)
                     throws UdfException {
        ArrayList<String> args =
                          srvInterface.getParamReader().getParamNames();

        // Ensure user supplied two arguments
        if (!(args.contains("from_char") && args.contains("to_char"))) {
            throw new UdfException(0, "You must supply two parameters" +
                        " to ReplaceChar: 'from_char' and 'to_char'");
        }

        // Verify that the from_char is a single character.
        String fromChar = srvInterface.getParamReader().getString("from_char");
        if (fromChar.length() != 1) {
            String message =  String.format("Replacechar expects a single " +
                "character in the 'from_char' parameter. Got length %d",
                fromChar.length());
            throw new UdfException(0, message);
        }

        // Save the from character in the plan context, to be read by
        // prepare() method.
        planCtxt.getWriter().setString("fromChar",fromChar);

        // Ensure to character parameter is a single characater
        String toChar = srvInterface.getParamReader().getString("to_char");
        if (toChar.length() != 1) {
            String message =  String.format("Replacechar expects a single "
                 + "character in the 'to_char' parameter. Got length %d",
                toChar.length());
            throw new UdfException(0, message);
        }
        // Save the to character in the plan data
        planCtxt.getWriter().setString("toChar",toChar);
    }

    // Called on every host that will filter data. Must instantiate the
    // UDFilter subclass.
    @Override
    public UDFilter prepare(ServerInterface srvInterface, PlanContext planCtxt)
                        throws UdfException {
        // Get data stored in the context by the plan() method.
        String fromChar = planCtxt.getWriter().getString("fromChar");
        String toChar = planCtxt.getWriter().getString("toChar");

        // Instantiate a filter object to perform filtering.
        return new ReplaceCharFilter(fromChar, toChar);
    }

    // Describe the parameters accepted by this filter.
    @Override
    public void getParameterType(ServerInterface srvInterface,
             SizedColumnTypes parameterTypes) {
        parameterTypes.addVarchar(1, "from_char");
        parameterTypes.addVarchar(1, "to_char");
    }
}
```

Subclassing UDFilter in Java

Your subclass of the `UDFilter` class is where you implement the filtering that you want your UDL Filter function to perform. Your subclass is instantiated by the `FilterFactory.setup()` method on each host in the HP Vertica cluster that is performing filtering for the data source.

UDFilter Methods

Your subclass of `UDFilter` can override the following methods. It must override the `process()` method.

- `setup()` - performs any initial setup tasks (such as retrieving parameters from the class context structure or initializing data structures that will be used during filtering) that your class needs to filter data. This method is called before HP Vertica calls the `process()` method for the first time.

  Note: `UDFilter` objects must be restartable. Once an instance of your subclass finishes filtering all of the data in a data source, Vertica Analytics Platform calls its `destroy()` method so it can deallocate any resources it allocated. If there are still data sources that have not yet been processed, Vertica Analytics Platform may later call `setup()` on the object again and have it filter the data in a new data stream. Therefore, your `destroy()` method should leave an object of your `UDFilter` subclass in a state where the `setup()` method can prepare it to be reused.

- `destroy()` - frees any resources used by your `UDFilter` subclass (which are usually set up in the `setup()` method). Vertica Analytics Platform calls this method after the `process()` method indicates it has finished filtering all of the data in the data stream.

- `process()` - performs the actual filtering of data. It gets two instances of the `DataBuffer` class among its parameters. Your override of this method should read data from the input `DataBuffer.buf` byte array, manipulate it on some manner (decompress it, filter out bytes, etc.), and write the result to the output `DataBuffer` object. Since there may not be a 1 to 1 correlation between the number of bytes your implementation reads and the number it writes,it should process data until it either runs out of data to read, or until it runs out of space in the output buffer. When one of these conditions occur, your method should return one of the values defined by `StreamState`:

  - OUTPUT_NEEDED if it needs more room in its output buffer.
  - INPUT_NEEDED if it has run out of input data (but the data source has not yet been fully processed).
  - DONE if it has processed all of the data in the data source.
  - KEEP_GOING if it cannot proceed for an extended period of time. It will be called again. Do not block indefinitely. If you do, then you prevent the user from canceling the query.

  Before returning, your `process()` method must set the input `DataBuffer` object's `offset` property to the number of bytes that it successfully read from the input, and the offset property of the output `DataBuffer` to the number of bytes it wrote. Setting these properties ensures that the next call to `process()` will resume reading and writing data at the correct points in the buffers. Your process() method also needs to check the `InputState` object passed to it to determine if there is more data in the data source. When this object is equal to END_OF_FILE, then the data remaining in the input data is the last data in the data source. Once it has processed all of the remaining data, `process()` must return DONE.

Example UDFilter

The following example `UDFilter` subclass, named `ReplaceCharFilter`, corresponds to the factory class explained in Subclassing FilterFactory. It reads the data stream, replacing each occurrence of a user-specified character with another character.

```
package com.vertica.JavaLibs;

import com.vertica.sdk.DataBuffer;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.State.InputState;
import com.vertica.sdk.State.StreamState;
import com.vertica.sdk.UDFilter;

public class ReplaceCharFilter extends UDFilter {

    private byte[] fromChar;
    private byte[] toChar;

    public ReplaceCharFilter(String fromChar, String toChar){
        // Stores the from char and to char as byte arrays. This is
        // not a robust method of doing this, but works for this simple
        // example.
        this.fromChar= fromChar.getBytes();
        this.toChar=toChar.getBytes();
    }
    @Override
    public StreamState process(ServerInterface srvInterface, DataBuffer input,
            InputState input_state, DataBuffer output) {

        // Check if there is no more input and the input buffer has been completely
        // processed. If so, filtering is done.
        if (input_state == InputState.END_OF_FILE && input.buf.length == 0) {
            return StreamState.DONE;
        }

        // Get current position in the input buffer
        int offset = output.offset;

        // Determine how many bytes to process. This is either until input
        // buffer is exhausted or output buffer is filled
        int limit = Math.min((input.buf.length - input.offset),
                (output.buf.length - output.offset));

        for (int i = input.offset; i < limit; i++) {
            // This example just replaces each instance of from_char
            // with to_char. It does not consider things such as multi-byte
            // UTF-8 characters.
            if (input.buf[i] == fromChar[0]) {
                output.buf[i+offset] = toChar[0];
            } else {
                // Did not find from_char, so copy input to the output
                output.buf[i+offset]=input.buf[i];
            }
        }

        input.offset += limit;
        output.offset += input.offset;

        if (input.buf.length - input.offset < output.buf.length - output.offset) {
            return StreamState.INPUT_NEEDED;
        } else {
            return StreamState.OUTPUT_NEEDED;
        }
    }
}
```

Developing UDL Parser Functions in Java

User Defined Load Parser functions (also referred to as UDParsers) are used within a COPY statement to read a stream of bytes and output a sequence of tuples that HP Vertica inserts into a table. UDL Parser functions can parse data in formats not understood by the COPY statement's built-in parser, or for data that require more specific control that the built-in parser allows. For example, you could load a CSV file using a specific CSV library.

You can use a single UDL Parser function within a COPY statement. You can use UDL Parser functions in conjunction with UDSource and UDFilter functions as well as the COPY statement's built-in data source and filter.

To create a UDL Parser function, you must implement a subclass of both UDParser and ParserFactory classes defined by the Vertica Analytics Platform Java SDK.

The HP Vertica Java SDK provides an example parser functions in `/opt/vertica/sdk/examples/JavaUDx/UDLFuctions`.

Java UDL Parser Example Overview

The example UDL Parser (named `NumericTextParser`) shown in the following sections parses integer values spelled out in words rather than digits (for example "One Two Three" for one-hundred twenty three). The parser:

- accepts a single parameter to set the character that separates columns in a row of data. The separator defaults to the pipe (|) character.
- ignores extra spaces and the capitalization of the words used to spell out the digits.
- recognizes the digits using the following words: zero, one, two, three, four, five, six, seven, eight, nine.
- expects the words spelling out an integer to be separated by at least one space.

- rejects any row of data that cannot be completely parsed into integers.
- generates an error if the output table has a non-integer column.

The following example demonstrates loading the library and defining the `NumericTextParser` function in the HP Vertica catalog, and then using the parser to interactively load data using the COPY statement.

```
=> CREATE LIBRARY JavaLib AS '/home/dbadmin/JavaLib.jar' LANGUAGE 'JAVA';
CREATE LIBRARY

=> CREATE PARSER NumericTextParser AS LANGUAGE 'java'
->    NAME 'com.myCompany.UDParser.NumericTextParserFactory'
->    LIBRARY JavaLib;
CREATE PARSER FUNCTION
=> CREATE TABLE t (i INTEGER);
CREATE TABLE
=> COPY t FROM STDIN WITH PARSER NumericTextParser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> One
>> Two
>> One Two Three
>> \.
=> SELECT * FROM t ORDER BY i;
  i
-----
   1
   2
 123
(3 rows)

=> DROP TABLE t;
DROP TABLE
=> -- Parse multi-column input
=> CREATE TABLE t (i INTEGER, j INTEGER);
CREATE TABLE
=> COPY t FROM stdin WITH PARSER NumericTextParser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> One | Two
>> Two | Three
>> One Two Three | four Five Six
>> \.
=> SELECT * FROM t ORDER BY i;
  i  |  j
-----+-----
   1 |   2
   2 |   3
 123 | 456
(3 rows)

=> TRUNCATE TABLE t;
TRUNCATE TABLE
=> -- Use alternate separator character
=> COPY t FROM STDIN WITH PARSER NumericTextParser(separator='*');
```

```
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Five * Six
>> seven * eight
>> nine * one zero
>> \.
=> SELECT * FROM t ORDER BY i;
 i | j
---+----
 5 |  6
 7 |  8
 9 | 10
(3 rows)

=> TRUNCATE TABLE t;
TRUNCATE TABLE

=> -- Rows containing data that does not parse into digits is rejected.
=> DROP TABLE t;
DROP TABLE
=> CREATE TABLE t (i INTEGER);
CREATE TABLE
=> COPY t FROM STDIN WITH PARSER NumericTextParser();
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> One Zero Zero
>> Two Zero Zero
>> Three Zed Zed
>> Four Zero Zero
>> Five Zed Zed
>> \.
SELECT * FROM t ORDER BY i;
  i
-----
 100
 200
 400
(3 rows)

=> -- Generate an error by trying to copy into a table with a non-integer column
=> DROP TABLE t;
DROP TABLE
=> CREATE TABLE t (i INTEGER, j VARCHAR);
CREATE TABLE
=> COPY t FROM STDIN WITH PARSER NumericTextParser();
vsql:UDParse.sql:94: ERROR 3399:  Failure in UDx RPC call
InvokeGetReturnTypeParser(): Error in User Defined Object [NumericTextParser],
error code: 0
com.vertica.sdk.UdfException: Column 2 of output table is not an Int
        at com.myCompany.UDParser.NumericTextParserFactory.getParserReturnType
        (NumericTextParserFactory.java:70)
        at com.vertica.udxfence.UDxExecContext.getReturnTypeParser(
        UDxExecContext.java:1464)
        at com.vertica.udxfence.UDxExecContext.getReturnTypeParser(
        UDxExecContext.java:768)
        at com.vertica.udxfence.UDxExecContext.run(UDxExecContext.java:236)
        at java.lang.Thread.run(Thread.java:662)
```

Subclassing ParserFactory in Java

Your subclass of the HP Vertica SDK's `ParserFactory` class to is responsible for:

- performing the initial validation of the parameters in the function call your `UDParser` function.
- setting up any data structures your `UDPaerser` subclass instances need to perform their work. This information can include any parameters passed by the user to the function.
- creating an instance of your `UDParser` subclass.

ParserFactory Methods

You can override the following methods in your subclass of `ParserFactory`:

- `plan()` - HP Vertica calls this method once on the initiator node. It should perform the following tasks:
  - Check any parameters that have been passed from the function call in the COPY statement and provide a helpful error message if they have errors. You read the parameters by getting a `ParamReader` object from the instance of `ServerInterface` passed into your `plan()` method.
  - Store any information that the individual hosts need in order to parse the data. For example, you could store parameters in the `PlanContext` instance passed in via the `planCtxt` parameter. This object is the only means of communication between the `plan()` method (which only runs on the initiator node) and the `prepareUDSources()` method (which runs on each host reading from a data source).

    You store data in the `PlanContext` by getting a `ParamWriter` object from the `getWriter()` method. You then write parameters by calling methods on the `ParamWriter` such as `setString`.

    Note: `ParamWriter` only offers the ability to store simple data types. For complex types, you will need to serialize the data in some manner and store it as a string or long string.

- `prepare()` - instantiates a member of your `UDFilter` subclass to perform the data filtering.
- `getParameterType()` - defines the name and types of parameters that your function uses. HP Vertica uses this information to warn function callers that any unknown parameters that they provide will have no effect, or that parameters they did not provide will use default values. You should define the types and parameters for your function, but overriding this method is optional.
- `getParserReturnType()` - defines the data types (and if applicable, the size, precision, or scale of the data types) of the table columns that the parser will output. Usually, your implementation of this method reads data types of the output table from the `argType` and `perColumnParamReader` arguments, then verifies that it can output the appropriate data type. If it is prepared to output the data types, it calls methods on the `SizedColumnTypes` object passed

in the `returnType` argument. In addition to the data type of the output column, your method should also specify any additional information about the column's data type:

- For CHAR/VARCHAR types, specify its maximum length.
- For NUMERIC types, specify its precision and scale.
- For Time/Timestamp types (with or without time zone), specify its precision (-1 means unspecified).
- For all other types, no length/precision specification is required.

Example ParserFactory

The following example is a subclass of `ParserFactory` named `NumericTextParserFactory`. The `NumericTextParser` accepts a single optional parameter named separator, which is defined in the override of the `getParameterType()` method, and whose values is stored by the override of the `plan()` method. Since `NumericTextParser` only outputs integer values, the override of the `getParserReturnType()` method throws an exception if the output table contains a column whose data type is not integer.

```
package com.myCompany.UDParser;

import java.util.regex.Pattern;

import com.vertica.sdk.ParamReader;
import com.vertica.sdk.ParamWriter;
import com.vertica.sdk.ParserFactory;
import com.vertica.sdk.PerColumnParamReader;
import com.vertica.sdk.PlanContext;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.SizedColumnTypes;
import com.vertica.sdk.UDParser;
import com.vertica.sdk.UdfException;
import com.vertica.sdk.VerticaType;

public class NumericTextParserFactory extends ParserFactory {

    // Called once on the initiator host to check the parameters and set up the
    // context data that hosts performing processing will need later.
    @Override
    public void plan(ServerInterface srvInterface,
                PerColumnParamReader perColumnParamReader,
                PlanContext planCtxt) {

        String separator = "|"; // assume separator is pipe character

        // See if a parameter was given for column separator
        ParamReader args = srvInterface.getParamReader();
        if (args.containsParameter("separator")) {
            separator = args.getString("separator");
            if (separator.length() > 1) {
                throw new UdfException(0,
                    "Separator parameter must be a single character");
            }
            if (Pattern.quote(separator).matches("[a-zA-Z]")) {
                throw new UdfException(0,
                        "Separator parameter cannot be a letter");
            }
        }

        // Save separator character in the Plan Data
        ParamWriter context = planCtxt.getWriter();
        context.setString("separator", separator);
    }

    // Define the data types of the output table that the parser will return.
    // Mainly, this just ensures that all of the columns in the table which
    // is the target of the data load are integer.
    @Override
    public void getParserReturnType(ServerInterface srvInterface,
                PerColumnParamReader perColumnParamReader,
                PlanContext planCtxt,
                SizedColumnTypes argTypes,
                SizedColumnTypes returnType) {

        // Get access to the output table's columns
        for (int i = 0; i < argTypes.getColumnCount(); i++ ) {
            if (argTypes.getColumnType(i).isInt()) {
                // Column is integer... add it to the output
                 returnType.addInt(argTypes.getColumnName(i));
            } else {
                // Column isn't an int, so throw an exception.
                // Technically, not necessary since the
                // UDx framework will automatically error out when it sees a
                // Discrepancy between the type in the target table and the
                // types declared by this method. Throwing this exception will
                // provide a clearer error message to the user.
                String message = String.format(
                    "Column %d of output table is not an Int", i + 1);
                throw new UdfException(0, message);
            }
        }
    }

    // Instantiate the UDParser subclass named NumericTextParser. Passes the
    // separator characetr as a paramter to the constructor.
    @Override
    public UDParser prepare(ServerInterface srvInterface,
            PerColumnParamReader perColumnParamReader, PlanContext planCtxt,
            SizedColumnTypes returnType) throws UdfException {
        // Get the separator character from the context
        String separator = planCtxt.getReader().getString("separator");
        return new NumericTextParser(separator);
    }

    // Describe the parameters accepted by this parser.
    @Override
    public void getParameterType(ServerInterface srvInterface,
             SizedColumnTypes parameterTypes) {
        parameterTypes.addVarchar(1, "separator");
    }
}
```

Subclassing UDParser in Java

Your subclass of the `UDParser` class defines how your UDL Parser Function parses data. Its `process()` method parses an input stream into rows and tuples that the COPY statement inserts into an HP Vertica table. `UDParser` subclasses are typically used to parse data that is in a format that the COPY statement's native parser cannot handle.

UDParser Methods

The `UDParser` class defines the following methods your subclass can override. It must override the `process()` and `getRejectedRow()` methods.

- `setup()` - performs any initial setup tasks (such as retrieving parameters from the class context structure or initializing data structures that will be used during filtering) that your class needs to parse data. This method is called before HP Vertica calls the `process()` method for the first time.

  Note: `UDParser` objects must be restartable. Once an instance of your subclass finishes parsing all of the data in a data source, Vertica Analytics Platform calls its `destroy()` method so it can deallocate any resources it allocated. If there are still data sources that have not yet been processed, Vertica Analytics Platform may later call `setup()` on the object again and have it parse the data in a new data stream. Therefore, your `destroy()` method should leave an object of your `UDParser` subclass in a state where the `setup()` method can prepare it to be reused.

- `process()` - performs the parsing of data. HP Vertica passes this method a buffer of data it must parse into columns and rows. It must reject any data that it cannot parse, so that HP Vertica can write the reason for the rejection and the rejected data to files.

  The `process()` method needs to parse as much data as it can in the input buffer. There is no guarantee that the buffer ends on a complete row, so it may have to stop parsing in the middle of a row and ask Vertica Analytics Platformmore data.

  Once your parser has extracted a column value, it should write it to the output using the methods on the `StreamObject` object available from the `writer` field. This field is set up automatically by the `UDParser` constructor, so it is vital that if you create an override of the constructor that you have it call `super()`. The `StreamObject` class has data type-specific methods to add data to the parser's output stream.

  When your parser finishes processing a row of data (usually by encountering an end of row marker) it must call `writer.next()` to advance the output stream to a new row.

When your `process()` method finishes the buffer, it should determine if it has processed the entire input stream by checking the `InputState` object passed in the `input_state` parameter to see if it indicates the input data stream has been full read. Your `process()` method's return value tells HP Vertica its current state:

  - INPUT_NEEDED means it the parser has reached the end of the buffer and needs more data to parse.
  - DONE means it has reached the end of the input data stream.
  - REJECT means it has rejected the last row of data it read (see below).

- `destroy()` - frees up any resources reserved by the `setup()` or `process()` method. HP Vertica calls this method after the `process()` method indicates it has completed parsing the data source.
- `getRejectedRecord()` - returns information about the last rejected row of data. Usually, this method just returns a member of the `RejectedRecord` class with details of the rejected row.

Rejecting Rows

If your parser finds data it cannot parse, it rejects the row by:

1. Saving details about the rejected row data and the reason for the rejection. These pieces of information can be directly stored in a `RejectedRecord` object, or (as in the example shown below) in fields on your `UDParser` subclass until they are needed.
2. Updating its position in the input buffer by updating `input.offset` buffer so it can resume parsing after the rejected row.
3. Signaling that it has rejected a row by returning with the value `StreamState.REJECT`.
4. HP Vertica calls your `UDParser` subclass's `getRejectedRecord` method to get all of the information it needs about the rejected row. Your override of this method must return an instance of the `RejectedRecord` class with the details about the rejected row.

Example UDParser

The following example parses text strings representing integers as described in Java UDL Parser Example Overview.

```
package com.myCompany.UDParser;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

import com.vertica.sdk.DataBuffer;
import com.vertica.sdk.DestroyInvocation;
import com.vertica.sdk.RejectedRecord;
import com.vertica.sdk.ServerInterface;
import com.vertica.sdk.State.InputState;
import com.vertica.sdk.State.StreamState;
import com.vertica.sdk.StreamWriter;
import com.vertica.sdk.UDParser;
import com.vertica.sdk.UdfException;

public class NumericTextParser extends UDParser {

    private String separator; // Holds column separator character

    // List of strings that we accept as digits.
    private List<String> numbers = Arrays.asList("zero", "one",
            "two", "three", "four", "five", "six", "seven",
            "eight", "nine");

    // Hold information about the last rejected row.
    private String rejectedReason;
    private String rejectedRow;

    // Constructor gets the separator character from the Factory's prepare()
    // method.
    public NumericTextParser(String sepparam) {
        super();
        this.separator = sepparam;
    }

    // Called to perform the actual work of parsing. Gets a buffer of bytes
    // to turn into tuples.
    @Override
    public StreamState process(ServerInterface srvInterface, DataBuffer input,
            InputState input_state) throws UdfException, DestroyInvocation {

        int i=input.offset; // Current position in the input buffer
        // Flag to indicate whether we just found the end of a row.
        boolean lastCharNewline = false;
        // Buffer to hold the row of data being read.
        StringBuffer line = new StringBuffer();

        //Continue reading until end of buffer.
        for(; i < input.buf.length; i++){
            // Loop through input until we find a linebreak: marks end of row
            char inchar = (char) input.buf[i];
            // Note that this isn't a robust way to find rows. It should
            // accept a user-defined row separator. Also, the following
            // assumes ASCII line break metheods, which isn't a good idea
            // in the UTF world. But it is good enough for this simple example.
            if (inchar != '\n' && inchar != '\r') {
                // Keep adding to a line buffer until a full row of data is read
                line.append(inchar);
                lastCharNewline = false; // Last character not a new line
            } else {
                // Found a line break. Process the row.
                lastCharNewline = true; // indicate we got a complete row
                // Update the position in the input buffer. This is updated
                // whether the row is successfully processed or not.
                input.offset = i+1;
                // Call procesRow to extract values and write tuples to the
                // output. Returns false if there was an error.
                if (!processRow(line)) {
                    // Processing row failed. Save bad row to rejectedRow field
                    // and return to caller indicating a rejected row.
                    rejectedRow = line.toString();
                    // Update position where we processed the data.
                    return StreamState.REJECT;
                }
                line.delete(0, line.length()); // clear row buffer
            }
        }

        // At this point, process() has finished processing the input buffer.
        // There are two possibilities: need to get more data
        // from the input stream to finish processing, or there is
        // no more data to process. If at the end of the input stream and
        // the row was not terminated by a linefeed, it may need
        // to process the last row.

        if (input_state == InputState.END_OF_FILE && lastCharNewline) {
            // End of input and it ended on a newline. Nothing more to do
            return StreamState.DONE;
        } else if (input_state == InputState.END_OF_FILE && !lastCharNewline) {
            // At end of input stream but didn't get a final newline. Need to
            // process the final row that was read in, then exit for good.
            if (line.length() == 0) {
                // Nothing to process. Done parsing.
                return StreamState.DONE;
            }
            // Need to parse the last row, not terminated by a linefeed. This
            // can occur if the file being read didn't have a final line break.
            if (processRow(line)) {
                return StreamState.DONE;
            } else {
                // Processing last row failed. Save bad row to rejectedRow field
                // and return to caller indicating a rejected row.
                rejectedRow = line.toString();
                // Tell HP Vertica the entire buffer was processed so it won't
                // call again to have the line processed.
                input.offset = input.buf.length;
                return StreamState.REJECT;
            }
        } else {
            // Stream is not fully read, so tell Vertica to send more. If
            // process() didn't get a complete row before it hit the end of the
            // input buffer, it will end up re-processing that segment again
            // when more data is added to the buffer.
            return StreamState.INPUT_NEEDED;
        }
    }

    // Breaks a row into columns, then parses the content of the
    // columns. Returns false if there was an error parsing the
    // row, in which case it sets the rejected row to the input
    // line. Returns true if the row was successfully read.
    private boolean processRow(StringBuffer line)
```

```
                            throws UdfException, DestroyInvocation {
        String[] columns = line.toString().split(Pattern.quote(separator));
        // Loop through the columns, decoding their contents
        for (int col = 0; col < columns.length; col++) {
            // Call decodeColumn to extract value from this column
            Integer colval = decodeColumn(columns[col]);
            if (colval == null) {
                // Could not parse one of the columns. Indicate row should
                // be rejected.
                return false;
            }
            // Column parsed OK. Write it to the output. writer is a field
            // provided by the parent class. Since this parser only accepts
            // integers, there is no need to verify that data type of the parsed
            // data matches the data type of the column being written. In your
            // UDParsers, you may want to perform this verification.
            writer.setLong(col,colval);
        }
        // Done with the row of data. Advance output to next row.

        // Note that this example does not verify that all of the output columns
        // have values assigned to them. If there are missing values at the
        // end of a row, they get automatically get assigned a default value
        // (0 for integers). This isn't a robust solution. Your UDParser
        // should perform checks here to handle this situation and set values
        // (such as null) when appropriate.
        writer.next();
        return true; // Successfully processed the row.
    }

    // Gets a string with text numerals, i.e. "One Two Five Seven" and turns
    // it into an integer value, i.e. 1257. Returns null if the string could not
    // be parsed completely into numbers.
    private Integer decodeColumn(String text) {
        int value = 0; // Hold the value being parsed.

        // Split string into individual words. Eat extra spaces.
        String[] words = text.toLowerCase().trim().split("\\s+");

        // Loop through the words, matching them against the list of
        // digit strings.
        for (int i = 0; i < words.length; i++) {
            if (numbers.contains(words[i])) {
                // Matched a digit. Add the it to the value.
                int digit = numbers.indexOf(words[i]);
                value = (value * 10) + digit;
            } else {
                // The string didn't match one of the accepted string values
                // for digits. Need to reject the row. Set the rejected
                // reason string here so it can be incorporated into the
                // rejected reason object.
                //
                // Note that this example does not handle null column values.
                // In most cases, you want to differentiate between an
                // unparseable column value and a missing piece of input
                // data. This example just rejects the row if there is a missing
                // column value.
                rejectedReason = String.format(
                        "Could not parse '%s' into a digit",words[i]);
                return null;
            }
        }
        return value;
    }

    // HP Vertica calls this method if the parser rejected a row of data
    // to find out what went wrong and add to the proper logs. Just gathers
    // information stored in fields and returns it in an object.
    @Override
    public RejectedRecord getRejectedRecord() throws UdfException {
        return new RejectedRecord(rejectedReason,rejectedRow.toCharArray(),
                rejectedRow.length(), "\n");
    }
}
```

Updating UDx Libraries

There are two cases where you need to update libraries that you have already deployed:

- When you have upgraded HP Vertica to a new version that contains changes to the SDK API. For your libraries to work with the new server version, you need to recompile them with new version of the SDK. See UDx Library Compatibility with New Server Versions for more information.

- When you have made changes to your UDxs and you want to deploy these changes. Before updating your UDx library, you need to determine if you have changed the signature of any of the functions contained in the library. If you have, you need to drop the functions from the HP Vertica catalog before you update the library.

UDx Library Compatibility with New Server Versions

When you compile your User Defined Extension (UDx) library, you link it with the HP Vertica SDK version that accompanies the version of the HP Vertica server you have installed on your cluster. When you upgrade your HP Vertica database to a new server version, the UDx libraries defined in your HP Vertica catalog may be incompatible with the new server if the HP Vertica SDK changed between the two server versions. Libraries linked against an old version of the SDK do not work with a server that uses a new version of the SDK.

Note: Since the R language is interpreted, UDFs written in R are not linked to the HP Vertica SDK. Therefore, changes to the HP Vertica SDK between HP Vertica versions will not necessarily make your R-based UDF libraries incompatible with a new server version. They will only need to be updated if the APIs in the SDK that your UDFs call actually change (i.e. change the number or data types of their arguments).

Changes to the SDK usually only take place between major version upgrades (from version 6.0 and version 6.1, for example). Usually, there is no change to the SDK between minor server updates (for example, from version 7.0..0 to 7.0..1).

Note: In rare cases, there may be changes to the SDK between minor versions of the HP Vertica server. You should review the HP Vertica release notes before performing an upgrade to determine if your old UDx libraries will need to be recompiled to work with the new server version.

If you upgrade your HP Vertica server to a new version and there was no change to the SDK between the two versions of HP Vertica, your UDx libraries will continue to work without you having to take any further steps. If the SDK of the new server version is incompatible with the old SDK, you need to recompile and redeploy the UDx library before your extensions will work. In the meantime, any attempt to use UDFs or UDLs defined in the incompatible library result in an error message:

```
ERROR 2858:  Could not find function definitionHINT:
```

```
This usually happens due to missing or corrupt libraries, libraries built
with the wrong SDK version, or due to a concurrent session dropping the library
or function. Try recreating the library and function
```

If your UDx library is incompatible with the new version of the HP Vertica server, you must:

1. Recompile your UDx library using the new version of the HP Vertica SDK. See Compiling Your C++ UDF for more information.

2. Deploy the new version of your library. See Deploying A New Version of Your UDx Library.

Determining If a UDF Signature Has Changed

You need to be careful when making changes to UDF libraries that contain functions you have already deployed in your HP Vertica database. When you deploy a new version of your UDF library, HP Vertica does not ensure that the signatures of the functions that are defined in the library match the signature of the function that is already defined in the HP Vertica catalog. If you have changed the signature of a UDF in the library then update the library in the HP Vertica database, calls to the altered UDF will produce errors.

Making any of the following changes to a UDF alters its signature:

- Changing the number of arguments accepted or the data type of any argument accepted by your function (not including polymorphic functions).

- Changing the number or data types of any return values or output columns.

- Changing the name of the factory class that HP Vertica uses to create an instance of your function code.

- Changing the null handling or volatility behavior of your function.

- Removed the function's factory class from the library completely.

The following changes do not alter the signature of your function, and do not require you to drop the function before updating the library:

- Changing the number or type of arguments handled by a polymorphic function. HP Vertica does not process the arguments the user passes to a polymorphic function.

- Changing the the name, data type, or number of parameters accepted by your function. The parameters your function accepts are not determined by the function signature. Instead, HP Vertica passes all of the parameters the user included in the function call, and your function processes them at runtime. See UDF Parameters for more information about parameters.

- Changing any of the internal processing performed by your function.

- Adding new UDFs to the library.

After you drop any functions whose signatures have changed, you load the new library file, then re-create your altered functions. If you have not made any changes to the signature of your UDFs, you can just update the library file in your HP Vertica database without having to drop or alter your function definitions. As long as the UDF definitions in the HP Vertica catalog match the signatures of the functions in your library, function calls will work transparently after you have updated the library. See Deploying A New Version of Your UDx Library.

Deploying A New Version of Your UDx Library

You need to deploy a new version of your UDx library if:

- You have made changes to the library that you now want to roll out to your HP Vertica database.
- You have upgraded your HP Vertica to a new version whose SDK is incompatible with the previous version.

The process of deploying a new version of your library is similar to deploying it initially.

1. If you are deploying a UDx library developed in C++ or Java, you must compile it with the current version of the HP Vertica SDK. See Compiling Your C++ UDF for details.
2. Copy your UDx's library file (a `.so` file for libraries developed in C++, or a `.jar` file for libraries developed in Java) or R source file to a host in your HP Vertica database.
3. Connect to the host using **vsql**.
4. If you have changed the signature of any of the UDFs or UDLs in the shared library, you must drop them using DROP statements such as DROP FUNCTION or DROP SOURCE. If you are unsure whether any of the signatures of your functions have changed, see Determining If a UDF Signature Has Changed.

   Note: If all of the UDF or UDL signatures in your library have changed, you may find it more convenient to drop the library using the DROP LIBRARY statement with the CASCADE option to drop the library and all of the functions and loaders that reference it. This can save you the time it would take to drop each UDF or UDL individually. You can then reload the library and recreate all of the extensions using the same process you used to deploy the library in the first place. See CREATE LIBRARY in the SQL Reference Manual.

5. Use the ALTER LIBRARY statement to update the UDx library definition with the file you copied in step 1. For example, if you want to update the library named ScalarFunctions with a file named `ScalarFunctions-2.0.so` in the dbadmin user's home directory, you could use the command:

```
ALTER LIBRARY ScalarFunctions AS '/home/dbadmin/ScalarFunctions-2.0.so';
```

Once you have updated the UDx library definition to use the new version of your shared library, the UDFs and UDLs that are defined using classes in your UDx library begin using the new shared library file without any further changes.

6. If you had to drop any functions in step 4, recreate them using the new signature defined by the factory classes in your library. See CREATE FUNCTION Statements in the SQL Reference Manual.

Listing the UDxs Contained in a Library

Once a library has been loaded using the CREATE LIBRARY statement, you can find the UDFs and UDLs it contains by querying the USER_LIBRARY_MANIFEST system table:

```
=> CREATE LIBRARY ScalarFunctions AS '/home/dbadmin/ScalarFunctions.so';
CREATE LIBRARY
=> \x
Expanded display is on.
=> SELECT * FROM USER_LIBRARY_MANIFEST WHERE lib_name = 'ScalarFunctions';
-[ RECORD 1 ]-------------------
schema_name | public
lib_name    | ScalarFunctions
lib_oid     | 45035996273792402
obj_name    | RemoveSpaceFactory
obj_type    | Scalar Function
arg_types   | Varchar
return_type | Varchar
-[ RECORD 2 ]-------------------
schema_name | public
lib_name    | ScalarFunctions
lib_oid     | 45035996273792402
obj_name    | Div2intsInfo
obj_type    | Scalar Function
arg_types   | Integer, Integer
return_type | Integer
-[ RECORD 3 ]-------------------
schema_name | public
lib_name    | ScalarFunctions
lib_oid     | 45035996273792402
obj_name    | Add2intsInfo
obj_type    | Scalar Function
arg_types   | Integer, Integer
return_type | Integer
```

The `obj_name` column lists the factory classes contained in the library. These are the names you use to define UDFs and UDLs in the database catalog using statements such as CREATE FUNCTION and CREATE SOURCE.

Appendix: Error Codes

SQLSTATEs and Error Codes

HP Vertica reports warnings and errors via two different mechanisms: SQLSTATEs and error messages. SQLSTATEs are intended for use by client applications, such as those accessing HP Vertica via ODBC or JDBC. Error messages are displayed to interactive users (for example, users connected to HP Vertica through **vsql**) and written to error logs.

SQLSTATE

HP Vertica reports the success or failure of each statement it executes to client applications using a five-character SQLSTATE value. Many of these values are defined by the SQL standard. Others (identified by the letter "V" in their values) are HP Vertica-specific.

SQLSTATE values are grouped into classes which are defined by the first two characters in the SQLSTATE value. The last three characters indicate a specific condition within a class. For example, the SQLSTATE class 22 represents all data errors. The specific SQLSTATE value 22012 represents a division by zero error. SQLSTATE classes let an application that does not recognize a specific SQLSTATE value to still get a general idea of the result.

Warning and Error Messages

Each error and warning message displayed to interactive users or written to a log file by HP Vertica has its own numeric error code assigned to it. For example:

```
ERROR 3117: Division by zeroWARNING 4098: No projections found
ERROR 5617: Multiple WITH clauses not allowed
```

The error code number is not related to the SQLSTATE value. However, error and warning messages do correspond to a specific SQLSTATE. They are just a more-specific human-readable message compared to the SQLSTATE, which is mainly intended for client applications.

For example, all warning messages displayed by HP Vertica correspond to the SQLSTATE class 01. The warning message "`WARNING 3084: Design Workspace couldn't be dropped`" corresponds to the SQLSTATE value 01000 ERRCODE_WARNING.

Error codes do not change change between HP Vertica releases, but individual error and warning messages may be added or removed in new releases. Your client application should not depend on particular error code appearing from one release to the next. Instead, it should use the SQLSTATE value to determine the result of executing a statement.

See the SQL State List for a list of all the SQLSTATE classes and values defined by HP Vertica. This table also links to lists of error or warning messages that are associated with each SQLSTATE value.

SQL State List

The following table lists the SQLSTATE classes and individual SQLSTATE codes.

SQLState	Description	Details
Class 00—Successful Completion		
00000	ERRCODE_SUCCESSFUL_COMPLETION	
Class 01—Warning		
01000	ERRCODE_WARNING	associated warning messages
01003	ERRCODE_WARNING_NULL_VALUE_ELIMINATED_IN_SET_FUNCTION	
01004	ERRCODE_WARNING_STRING_DATA_RIGHT_TRUNCATION	
01006	ERRCODE_WARNING_PRIVILEGE_NOT_REVOKED	associated warning messages
01007	ERRCODE_WARNING_PRIVILEGE_NOT_GRANTED	associated warning messages
01008	ERRCODE_WARNING_PRIVILEGE_ALREADY_GRANTED	
01009	ERRCODE_WARNING_PRIVILEGE_ALREADY_REVOKED	
0100C	ERRCODE_WARNING_DYNAMIC_RESULT_SETS_RETURNED	
01V01	ERRCODE_WARNING_DEPRECATED_FEATURE	associated warning messages
01V02	ERRCODE_WARNING_QUERY_RETRIED	
Class 02—No Data		
02000	ERRCODE_NO_DATA	
02001	ERRCODE_NO_ADDITIONAL_DYNAMIC_RESULT_SETS_RETURNED	
Class 03—SQL Statement Not Yet Complete		
03000	ERRCODE_SQL_STATEMENT_NOT_YET_COMPLETE	

SQLState	Description	Details
Class 08—Client Connection Exception		
08000	ERRCODE_CONNECTION_EXCEPTION	associated error messages
08001	ERRCODE_SQLCLIENT_UNABLE_TO_ESTABLISH_ SQLCONNECTION	associated error messages
08003	ERRCODE_CONNECTION_DOES_NOT_EXIST	associated error messages
08004	ERRCODE_SQLSERVER_REJECTED_ ESTABLISHMENT_OF_SQLCONNECTION	
08006	ERRCODE_CONNECTION_FAILURE	associated error messages
08007	ERRCODE_TRANSACTION_RESOLUTION_UNKNOWN	
08V01	ERRCODE_PROTOCOL_VIOLATION	associated error messages
Class 09—Triggered Action Exception		
09000	ERRCODE_TRIGGERED_ACTION_EXCEPTION	
Class 0A—Feature Not Supported		
0A000	ERRCODE_FEATURE_NOT_SUPPORTED	associated error messages
Class 0B—Invalid Transaction Initiation		
0B000	ERRCODE_INVALID_TRANSACTION_INITIATION	associated error messages
Class 0F—Locator Exception		
0F000	ERRCODE_LOCATOR_EXCEPTION	
0F001	ERRCODE_L_E_INVALID_SPECIFICATION	
Class 0L—Invalid Grantor		
0L000	ERRCODE_INVALID_GRANTOR	
0LV01	ERRCODE_INVALID_GRANT_OPERATION	associated error messages
Class 0P—Invalid Role Specification		
0P000	ERRCODE_INVALID_ROLE_SPECIFICATION	

SQLState	Description	Details
Class 21—Cardinality Violation		
21000	ERRCODE_CARDINALITY_VIOLATION	
Class 22—Data Exception		
22000	ERRCODE_DATA_EXCEPTION	associated error messages
22001	ERRCODE_STRING_DATA_RIGHT_TRUNCATION	associated error messages
22002	ERRCODE_NULL_VALUE_NO_INDICATOR_ PARAMETER	
22003	ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE	associated error messages
22004	ERRCODE_NULL_VALUE_NOT_ALLOWED	associated error messages
22005	ERRCODE_ERROR_IN_ASSIGNMENT	
22007	ERRCODE_INVALID_DATETIME_FORMAT	associated error messages
22008	ERRCODE_DATETIME_FIELD_OVERFLOW	associated error messages
22009	ERRCODE_INVALID_TIME_ZONE_DISPLACEMENT_ VALUE	associated error messages
2200B	ERRCODE_ESCAPE_CHARACTER_CONFLICT	associated error messages
2200C	ERRCODE_INVALID_USE_OF_ESCAPE_CHARACTER	
2200D	ERRCODE_INVALID_ESCAPE_OCTET	associated error messages
2200F	ERRCODE_ZERO_LENGTH_CHARACTER_STRING	
2200G	ERRCODE_MOST_SPECIFIC_TYPE_MISMATCH	
22010	ERRCODE_INVALID_INDICATOR_PARAMETER_ VALUE	
22011	ERRCODE_SUBSTRING_ERROR	associated error messages
22012	ERRCODE_DIVISION_BY_ZERO	associated error messages

SQLState	Description	Details
22015	ERRCODE_INTERVAL_FIELD_OVERFLOW	associated error messages
22018	ERRCODE_INVALID_CHARACTER_VALUE_FOR_CAST	
22019	ERRCODE_INVALID_ESCAPE_CHARACTER	associated error messages
2201B	ERRCODE_INVALID_REGULAR_EXPRESSION	associated error messages
2201E	ERRCODE_INVALID_ARGUMENT_FOR_LOG	
2201F	ERRCODE_INVALID_ARGUMENT_FOR_POWER_FUNCTION	
2201G	ERRCODE_INVALID_ARGUMENT_FOR_WIDTH_BUCKET_FUNCTION	associated error messages
22020	ERRCODE_INVALID_LIMIT_VALUE	
22021	ERRCODE_CHARACTER_NOT_IN_REPERTOIRE	associated error messages
22022	ERRCODE_INDICATOR_OVERFLOW	
22023	ERRCODE_INVALID_PARAMETER_VALUE	associated error messages
22024	ERRCODE_UNTERMINATED_C_STRING	
22025	ERRCODE_INVALID_ESCAPE_SEQUENCE	associated error messages
22026	ERRCODE_STRING_DATA_LENGTH_MISMATCH	
22027	ERRCODE_TRIM_ERROR	
2202E	ERRCODE_ARRAY_ELEMENT_ERROR	
22906	ERRCODE_NONSTANDARD_USE_OF_ESCAPE_CHARACTER	associated error messages
22V01	ERRCODE_FLOATING_POINT_EXCEPTION	
22V02	ERRCODE_INVALID_TEXT_REPRESENTATION	associated error messages
22V03	ERRCODE_INVALID_BINARY_REPRESENTATION	associated error messages

SQLState	Description	Details
22V04	ERRCODE_BAD_COPY_FILE_FORMAT	associated error messages
22V05	ERRCODE_UNTRANSLATABLE_CHARACTER	
22V0B	ERRCODE_ESCAPE_CHARACTER_ON_NOESCAPE	associated error messages
22V21	ERRCODE_INVALID_EPOCH	associated error messages
22V22	ERRCODE_PLPGSQL_ERROR	
22V23	ERRCODE_RAISE_EXCEPTION	associated error messages
22V24	ERRCODE_COPY_PARSE_ERROR	associated error messages
Class 23—Integrity Constraint Violation		
23000	ERRCODE_INTEGRITY_CONSTRAINT_VIOLATION	
23001	ERRCODE_RESTRICT_VIOLATION	
23502	ERRCODE_NOT_NULL_VIOLATION	associated error messages
23503	ERRCODE_FOREIGN_KEY_VIOLATION	associated error messages
23505	ERRCODE_UNIQUE_VIOLATION	associated error messages
23514	ERRCODE_CHECK_VIOLATION	
Class 24—Invalid Cursor State		
24000	ERRCODE_INVALID_CURSOR_STATE	
Class 25—Invalid Transaction State		
25000	ERRCODE_INVALID_TRANSACTION_STATE	
25001	ERRCODE_ACTIVE_SQL_TRANSACTION	
25002	ERRCODE_BRANCH_TRANSACTION_ALREADY_ACTIVE	
25003	ERRCODE_INAPPROPRIATE_ACCESS_MODE_FOR_BRANCH_TRANSACTION	

SQLState	Description	Details
25004	ERRCODE_INAPPROPRIATE_ISOLATION_LEVEL_FOR_BRANCH_TRANSACTION	
25005	ERRCODE_NO_ACTIVE_SQL_TRANSACTION_FOR_BRANCH_TRANSACTION	
25006	ERRCODE_READ_ONLY_SQL_TRANSACTION	
25007	ERRCODE_SCHEMA_AND_DATA_STATEMENT_MIXING_NOT_SUPPORTED	
25008	ERRCODE_HELD_CURSOR_REQUIRES_SAME_ISOLATION_LEVEL	
25V01	ERRCODE_NO_ACTIVE_SQL_TRANSACTION	associated error messages
25V02	ERRCODE_IN_FAILED_SQL_TRANSACTION	
Class 26—Invalid SQL Statement Name		
26000	ERRCODE_INVALID_SQL_STATEMENT_NAME	
Class 27—Triggered Data Change Violation		
27000	ERRCODE_TRIGGERED_DATA_CHANGE_VIOLATION	
Class 28—Invalid Authorization Specification		
28000	ERRCODE_INVALID_AUTHORIZATION_SPECIFICATION	associated error messages
28001	ERRCODE_ACCOUNT_LOCKED	
28002	ERRCODE_PASSWORD_EXPIRED	
28003	ERRCODE_PASSWORD_IN_GRACE_PERIOD	
Class 2B—Dependent Privilege Descriptors Still Exist		
2B000	ERRCODE_DEPENDENT_PRIVILEGE_DESCRIPTORS_STILL_EXIST	
2BV01	ERRCODE_DEPENDENT_OBJECTS_STILL_EXIST	associated error messages
Class 2D—Invalid Transaction Termination		
2D000	ERRCODE_INVALID_TRANSACTION_TERMINATION	
Class 2F—SQL Routine Exception		
2F000	ERRCODE_SQL_ROUTINE_EXCEPTION	

SQLState	Description	Details
2F002	ERRCODE_S_R_E_MODIFYING_SQL_DATA_NOT_PERMITTED	
2F003	ERRCODE_S_R_E_PROHIBITED_SQL_STATEMENT_ATTEMPTED	
2F004	ERRCODE_S_R_E_READING_SQL_DATA_NOT_PERMITTED	
2F005	ERRCODE_S_R_E_FUNCTION_EXECUTED_NO_RETURN_STATEMENT	
Class 34—Invalid Cursor Name		
34000	ERRCODE_INVALID_CURSOR_NAME	
Class 38—External Routine Exception		
38000	ERRCODE_EXTERNAL_ROUTINE_EXCEPTION	
38001	ERRCODE_E_R_E_CONTAINING_SQL_NOT_PERMITTED	
38002	ERRCODE_E_R_E_MODIFYING_SQL_DATA_NOT_PERMITTED	
38003	ERRCODE_E_R_E_PROHIBITED_SQL_STATEMENT_ATTEMPTED	
38004	ERRCODE_E_R_E_READING_SQL_DATA_NOT_PERMITTED	
Class 39—External Routine Invocation Exception		
39000	ERRCODE_EXTERNAL_ROUTINE_INVOCATION_EXCEPTION	
39001	ERRCODE_E_R_I_E_INVALID_SQLSTATE_RETURNED	
39004	ERRCODE_E_R_I_E_NULL_VALUE_NOT_ALLOWED	
39V01	ERRCODE_E_R_I_E_TRIGGER_PROTOCOL_VIOLATED	
39V02	ERRCODE_E_R_I_E_SRF_PROTOCOL_VIOLATED	
Class 3B—Savepoint Exception		
3B000	ERRCODE_SAVEPOINT_EXCEPTION	
3B001	ERRCODE_S_E_INVALID_SPECIFICATION	

SQLState	Description	Details
Class 3D—Invalid Catalog Name		
3D000	ERRCODE_INVALID_CATALOG_NAME	
Class 3F—Invalid Schema Name		
3F000	ERRCODE_INVALID_SCHEMA_NAME	
Class 40—Transaction Rollback		
40000	ERRCODE_TRANSACTION_ROLLBACK	
40001	ERRCODE_T_R_SERIALIZATION_FAILURE	
40002	ERRCODE_T_R_INTEGRITY_CONSTRAINT_ VIOLATION	
40003	ERRCODE_T_R_STATEMENT_COMPLETION_ UNKNOWN	
40V01	ERRCODE_T_R_DEADLOCK_DETECTED	associated error messages
Class 42—Syntax Error or Access Rule Violation		
42000	ERRCODE_SYNTAX_ERROR_OR_ACCESS_RULE_ VIOLATION	
42501	ERRCODE_INSUFFICIENT_PRIVILEGE	associated error messages
42601	ERRCODE_SYNTAX_ERROR	associated error messages
42602	ERRCODE_INVALID_NAME	associated error messages
42611	ERRCODE_INVALID_COLUMN_DEFINITION	associated error messages
42622	ERRCODE_NAME_TOO_LONG	associated error messages
42701	ERRCODE_DUPLICATE_COLUMN	associated error messages
42702	ERRCODE_AMBIGUOUS_COLUMN	associated error messages
42703	ERRCODE_UNDEFINED_COLUMN	associated error messages

SQLState	Description	Details
42704	ERRCODE_UNDEFINED_OBJECT	associated error messages
42710	ERRCODE_DUPLICATE_OBJECT	associated error messages
42712	ERRCODE_DUPLICATE_ALIAS	associated error messages
42723	ERRCODE_DUPLICATE_FUNCTION	associated error messages
42725	ERRCODE_AMBIGUOUS_FUNCTION	associated error messages
42803	ERRCODE_GROUPING_ERROR	associated error messages
42804	ERRCODE_DATATYPE_MISMATCH	associated error messages
42809	ERRCODE_WRONG_OBJECT_TYPE	associated error messages
42830	ERRCODE_INVALID_FOREIGN_KEY	associated error messages
42846	ERRCODE_CANNOT_COERCE	associated error messages
42883	ERRCODE_UNDEFINED_FUNCTION	associated error messages
42939	ERRCODE_RESERVED_NAME	associated error messages
42P20	ERRCODE_WINDOWING_ERROR	associated error messages
42V01	ERRCODE_UNDEFINED_TABLE	associated error messages
42V02	ERRCODE_UNDEFINED_PARAMETER	associated error messages
42V03	ERRCODE_DUPLICATE_CURSOR	associated error messages
42V04	ERRCODE_DUPLICATE_DATABASE	associated error messages

SQLState	Description	Details
42V05	ERRCODE_DUPLICATE_PSTATEMENT	
42V06	ERRCODE_DUPLICATE_SCHEMA	associated error messages
42V07	ERRCODE_DUPLICATE_TABLE	associated error messages
42V08	ERRCODE_AMBIGUOUS_PARAMETER	associated error messages
42V09	ERRCODE_AMBIGUOUS_ALIAS	associated error messages
42V10	ERRCODE_INVALID_COLUMN_REFERENCE	associated error messages
42V11	ERRCODE_INVALID_CURSOR_DEFINITION	associated error messages
42V12	ERRCODE_INVALID_DATABASE_DEFINITION	
42V13	ERRCODE_INVALID_FUNCTION_DEFINITION	associated error messages
42V14	ERRCODE_INVALID_PSTATEMENT_DEFINITION	
42V15	ERRCODE_INVALID_SCHEMA_DEFINITION	associated error messages
42V16	ERRCODE_INVALID_TABLE_DEFINITION	associated error messages
42V17	ERRCODE_INVALID_OBJECT_DEFINITION	associated error messages
42V18	ERRCODE_INDETERMINATE_DATATYPE	associated error messages
42V21	ERRCODE_UNDEFINED_PROJECTION	associated error messages
42V22	ERRCODE_UNDEFINED_NODE	
42V23	ERRCODE_UNDEFINED_PERMUTATION	
42V24	ERRCODE_UNDEFINED_USER	
42V25	ERRCODE_PATTERN_MATCH_ERROR	associated error messages

SQLState	Description	Details
42V26	ERRCODE_DUPLICATE_NODE	associated error messages
Class 44—WITH CHECK OPTION Violation		
44000	ERRCODE_WITH_CHECK_OPTION_VIOLATION	
Class 53—Insufficient Resources		
53000	ERRCODE_INSUFFICIENT_RESOURCES	associated error messages
53100	ERRCODE_DISK_FULL	associated error messages
53200	ERRCODE_OUT_OF_MEMORY	associated error messages
53300	ERRCODE_TOO_MANY_CONNECTIONS	
Class 54—Program Limit Exceeded		
54000	ERRCODE_PROGRAM_LIMIT_EXCEEDED	associated error messages
54001	ERRCODE_STATEMENT_TOO_COMPLEX	associated error messages
54011	ERRCODE_TOO_MANY_COLUMNS	associated error messages
54023	ERRCODE_TOO_MANY_ARGUMENTS	associated error messages
Class 55—Object Not In Prerequisite State		
55000	ERRCODE_OBJECT_NOT_IN_PREREQUISITE_STATE	associated error messages
55006	ERRCODE_OBJECT_IN_USE	associated error messages
55V02	ERRCODE_CANT_CHANGE_RUNTIME_PARAM	associated error messages
55V03	ERRCODE_LOCK_NOT_AVAILABLE	associated error messages
55V04	ERRCODE_TM_MARKER_NOT_AVAILABLE	associated error messages
Class 57—Operator Intervention		

SQLState	Description	Details
57000	ERRCODE_OPERATOR_INTERVENTION	
57014	ERRCODE_QUERY_CANCELED	associated error messages
57015	ERRCODE_SLOW_DELETE	associated error messages
57V01	ERRCODE_ADMIN_SHUTDOWN	associated error messages
57V02	ERRCODE_CRASH_SHUTDOWN	
57V03	ERRCODE_CANNOT_CONNECT_NOW	associated error messages
Class 58—System Error		
58030	ERRCODE_IO_ERROR	associated error messages
58V01	ERRCODE_UNDEFINED_FILE	associated error messages
58V02	ERRCODE_DUPLICATE_FILE	
Class V1—Vertica-specific multi-node errors class		
V1001	ERRCODE_LOST_CONNECTIVITY	associated error messages
V1002	ERRCODE_K_SAFETY_VIOLATION	associated error messages
V1003	ERRCODE_CLUSTER_CHANGE	associated error messages
Class V2—Vertica-specific miscellaneous errors class		
V2000	ERRCODE_AUTH_FAILED	associated error messages
V2001	ERRCODE_LICENSE_ISSUE	associated error messages
V2002	ERRCODE_MOVEOUT_ABORTED	
Class VC—Configuration File Error		
VC001	ERRCODE_CONFIG_FILE_ERROR	associated error messages

SQLState	Description	Details
VC002	ERRCODE_LOCK_FILE_EXISTS	
Class VD—DB Designer errors		
VD001	ERRCODE_DESIGNER_FUNCTION_ERROR	associated error messages
Class VP—User procedure errors		
VP000	ERRCODE_USER_PROC_ERROR	associated error messages
VP001	ERRCODE_USER_PROC_EXEC_ERROR	associated error messages
Class VX—Internal Error		
VX001	ERRCODE_INTERNAL_ERROR	associated error messages
VX002	ERRCODE_DATA_CORRUPTED	associated error messages
VX003	ERRCODE_INDEX_CORRUPTED	associated error messages

Warning Messages Associated with SQLSTATE 01000

This topic lists the warning associated with the SQLSTATE 01000.

SQLSTATE 01000 Description

ERRCODE_WARNING

Warning messages associated with this SQLState

```
WARNING 2021: string Directory for errors files was not created.
                           Unable to write errors for this instance of COPY command

WARNING 2022: string Directory for exceptions files was not created.
                           Unable to write errors for this instance of COPY command

WARNING 2023: string Directory for rejected data files was not created.
                           Unable to write errors for this instance of COPY command

WARNING 2362: Cannot begin transaction; transaction is already running

WARNING 3084: Design couldn't be dropped

WARNING 3152: Duplicate values in columns marked as UNIQUE will now be ignored for the remaind
    er of your session or until reenable_duplicate_key_error() is called

WARNING 3372: Failed to disable profiling: string
```

WARNING 3373: Failed to enable profiling: *string*

WARNING 3539: Incorrect results are possible. Please contact Vertica Support if unsure

WARNING 3791: Invalid view *string*: *string*

WARNING 4071: NO COMMIT option will be ignored for external table "*string*"

WARNING 4088: No new valid default roles specified. Retaining previous set of default roles fo
r user *string*

WARNING 4098: No projections found

WARNING 4102: No rows are inserted into table "*string*"."*string*" because ON COMMIT DELETE ROWS
is the default for create temporary table

WARNING 4116: No super projections created for table *string*.

WARNING 4246: Only GLOBAL scope is supported for clearing *string* profiles

WARNING 4463: Projection *string* is not up to date

WARNING 4468: Projection <*string*> is not available for query processing. Execute the select st
art_refresh() function to copy data into this projection.
              The projection must have a sufficient number of buddy projections and all n
odes must be up before starting a refresh

WARNING 4792: Storage option "*string*" will be ignored for external table "*string*"

WARNING 4871: System view *string* for tuning rule *string* is currently invalid

WARNING 4873: System view for tuning rule *string* does not exist

WARNING 4996: This request may deadlock the system. Please report the details to technical su
pport

WARNING 5068: Total declared length of columns of one of the constraints exceeds the limit, tr
uncation may happen

WARNING 5119: UDx code didn't respond when Vertica tried to get function prototype for *string*
in library *string*: *string*

WARNING 5448: View *string* is currently invalid

WARNING 5451: Violations of some of foreign key constraints may not be reported because of no
privilege on the foreign tables

WARNING 5642: Projection *string* is not persistent or not up to date; it will not be copied

WARNING 5643: Projection *string* is prejoin projection; it will not be copied

WARNING 5717: No statistics has been exported. Either the DB is empty or you try to export an
external table or you do not have access to the available objects

WARNING 5724: Segmentation clause contains a *string* - data loads may be slowed significantly

WARNING 5727: Sort clause contains a *string* - data loads may be slowed significantly

WARNING 5741: View *string* depends on other relations

WARNING 5819: Design could not be reset

WARNING 5821: Detected keys sharing the same case-insensitive key name

WARNING 5860: Due to the data isolation of temp tables with an on-commit-delete-rows policy, t
he compute_flextable_keys() and compute_flextable_keys_and_build_view() functions cannot
access this table's data. The build_flextable_view() function can be used with a user-pro
vided keys table to create a view, but involves a DDL commit which will delete the table's
rows

WARNING 5873: Failed to add table *string* of hcatalog schema *string* to catalog: *string*

WARNING 5875: Failed to alter table *string* of hcatalog schema *string* to catalog: *string*

WARNING 5880: Failed to describe table *string* in hcatalog database *string*: *string*

```
WARNING 5881: Failed to describe table string in schema string: HCatalog database string does
    not exist
WARNING 5884: Failed to list hcatalog tables of hcatalog schema string: string
WARNING 5886: Failed to mirror table string in schema string: string
WARNING 5900: Files in the Vertica DFS are not rebalanced
WARNING 5909: Found and ignored keys with names longer than the maximum column-name length lim
    it
WARNING 5912: HASH() arguments contain irregular expressions
WARNING 5917: Ignored some keys since the total key count exceeds the view column limit
WARNING 5922: Insufficient privileges to alter table string
WARNING 5923: Insufficient privileges to drop table string
WARNING 5991: Projection basename "string" hint was ignored. "string" is used as the basename
WARNING 5993: Projection is irregularly segmented by column
WARNING 6053: The view creation involved a DDL commit which deleted the table's rows
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Warning Messages Associated with SQLSTATE 01006

This topic lists the warning associated with the SQLSTATE 01006.

SQLSTATE 01006 Description

ERRCODE_WARNING_PRIVILEGE_NOT_REVOKED

Warning messages associated with this SQLState

```
WARNING 4925: The string "string" cannot be string string "string"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Warning Messages Associated with SQLSTATE 01007

This topic lists the warning associated with the SQLSTATE 01007.

SQLSTATE 01007 Description

ERRCODE_WARNING_PRIVILEGE_NOT_GRANTED

Warning messages associated with this SQLState

```
WARNING 5682: USAGE privilege on schema "string" also needs to be granted to "string"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Warning Messages Associated with SQLSTATE 01V01

This topic lists the warning associated with the SQLSTATE 01V01.

SQLSTATE 01V01 Description

ERRCODE_WARNING_DEPRECATED_FEATURE

Warning messages associated with this SQLState

```
WARNING 2693: Configuration parameter string has been deprecated; setting it has no effect
WARNING 4736: set_local_segment_threshold has been deprecated; setting it has no effect
WARNING 5669: The command has been deprecated
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 08000

This topic lists the error associated with the SQLSTATE 08000.

SQLSTATE 08000 Description

ERRCODE_CONNECTION_EXCEPTION

Error messages associated with this SQLState

```
ERROR 2029: string from stdin failed: string
ERROR 2708: Connection to database [string] is invalid
ERROR 2896: Could not receive data from server:string
ERROR 2908: Could not send data to server: string
ERROR 3276: Error while waiting on socket. value
ERROR 4342: Password encryption failed
ERROR 5197: Unknown authentication method (value) requested by server
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 08001

This topic lists the error associated with the SQLSTATE 08001.

SQLSTATE 08001 Description

ERRCODE_SQLCLIENT_UNABLE_TO_ESTABLISH_SQLCONNECTION

Error messages associated with this SQLState

```
ERROR 2322: Cancel() -- connect() failed:

ERROR 2324: Cancel() -- socket() failed:

ERROR 2823: Could not connect to server [string]: string
            Is the server running and accepting
            TCP/IP connections on port string?


ERROR 2824: Could not connect to server: string
            Is the server running on host [string] and accepting
            TCP/IP connections on port string?


ERROR 2839: Could not create socket: string


ERROR 2865: Could not get client address from socket: string

ERROR 2869: Could not get socket error status: string

ERROR 2912: Could not set socket to close-on-exec mode: string

ERROR 2913: Could not set socket to non-blocking mode: string

ERROR 2914: Could not set socket to TCP no delay mode: string

ERROR 2921: Could not translate host name "string" to address: string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 08003

This topic lists the error associated with the SQLSTATE 08003.

SQLSTATE 08003 Description

ERRCODE_CONNECTION_DOES_NOT_EXIST

Error messages associated with this SQLState

```
ERROR 4717: Server closed the connection unexpectedly
            This probably means the server terminated abnormally
            before or while processing the request.
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 08006

This topic lists the error associated with the SQLSTATE 08006.

SQLSTATE 08006 Description

ERRCODE_CONNECTION_FAILURE

Error messages associated with this SQLState

```
ERROR 2323: Cancel() -- send() failed: string
```

```
ERROR 2606: Client failed when looking for pending signals
```

```
ERROR 2607: Client has disconnected
```

```
ERROR 4539: Received no response from stringstring
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 08V01

This topic lists the error associated with the SQLSTATE 08V01.

SQLSTATE 08V01 Description

ERRCODE_PROTOCOL_VIOLATION

Error messages associated with this SQLState

```
ERROR 2055: string Unexpected message type string reading from stdin
```

```
ERROR 2257: Bind message has value parameter formats but value parameters
```

```
ERROR 2258: Bind message has value result formats but query has value columns
```

```
ERROR 3334: Expected a RowDescription Message
```

```
ERROR 3335: Expected a SendExport Message
```

```
ERROR 3575: Insufficient data left in message
ERROR 3631: Invalid CLOSE message subtype value
ERROR 3651: Invalid DESCRIBE message subtype value
ERROR 3699: Invalid message format
ERROR 3701: Invalid message type
ERROR 3702: Invalid message type value
ERROR 3755: Invalid string in message
ERROR 3887: Lost synchronization with server: length value
ERROR 4074: No data left in message
ERROR 4718: Server did not identify with a pid & key
ERROR 5181: Unexpected message type 0xhex value
ERROR 5208: Unknown message from server

ERROR 5872: Expected to flush an end-of-batch client message but received a message of type va
    lue.  Attempting to recover...
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 0A000

This topic lists the error associated with the SQLSTATE 0A000.

SQLSTATE 0A000 Description

ERRCODE_FEATURE_NOT_SUPPORTED

Error messages associated with this SQLState

```
ERROR 2009: string can not be used in function string
ERROR 2013: string clause is not supported for expressions
ERROR 2014: string Concatenated GZIP/BZIP is not supported with NATIVE/NATIVE VARCHAR formats
ERROR 2036: string is not a legal time unit
ERROR 2058: string VIEW is not supported
ERROR 2074: (OEE) External Tables not supported in this context
ERROR 2089: A correlated column in a subquery expression is not supported
ERROR 2114: ADD COLUMN over temporary tables is not supported
ERROR 2130: Aggregate function string (value) is not supported
ERROR 2133: Aggregate function calls cannot contain subqueries
ERROR 2138: Aggregate functions can only be called on columns of a table
ERROR 2161: ALL subquery with a correlated expression is not supported
ERROR 2165: ALTER COLUMN TYPE over temporary tables is not supported
ERROR 2166: ALTER TABLE does not support ADD COLUMN with other clauses
ERROR 2167: ALTER TABLE does not support ALTER COLUMN TYPE with other clauses
ERROR 2168: ALTER TABLE does not support DROP COLUMN with other clauses

ERROR 2169: ALTER TABLE does not support SET SCHEMA with other clauses

ERROR 2178: An expression containing a correlated subquery with aggregate function is not supp
    orted

ERROR 2183: Analytic functions are not allowed in an ORDER BY on a UNION/INTERSECT/EXCEPT

ERROR 2184: Analytic functions are not supported in the ORDER BY of an analytic function OVER
    clause

ERROR 2190: Analytics query with having clause expression that involves aggregates and subquer
    y is not supported

ERROR 2192: ANALYZE_CONSTRAINTS is currently not supported in non-default locales

ERROR 2208: Another Design/Deployment is in progress

ERROR 2210: ANTI join with segmented inner not supported

ERROR 2220: Argument string must not contain subqueries

ERROR 2226: Argument to seeded random_must be a constant

ERROR 2233: Array References are not supported

ERROR 2235: ArrayExpr is not supported

ERROR 2329: Cannot accept a value of type any

ERROR 2330: Cannot accept a value of type anyarray

ERROR 2331: Cannot accept a value of type anyelement

ERROR 2332: Cannot accept a value of type internal

ERROR 2333: Cannot accept a value of type language_handler

ERROR 2334: Cannot accept a value of type opaque

ERROR 2335: Cannot accept a value of type trigger

ERROR 2340: Cannot add IDENTITY/AUTO-INCREMENT columns

ERROR 2345: Cannot alter a column's default when a node is down

ERROR 2350: Cannot alter type of column "string" since it is referenced in the constraint "str
    ing"

ERROR 2351: Cannot alter type of column "string" since it is referenced in the default express
    ion of column "string"

ERROR 2352: Cannot alter type of column "string" since it is referenced in the partition expre
    ssion

ERROR 2353: Cannot alter type of column "string" since it is referenced in the segmentation ex
    pression of projection "string"

ERROR 2354: Cannot alter type of column with a default expression

ERROR 2360: Cannot assign to system column "string"

ERROR 2363: Cannot broadcast non-subquery outer input to a join

ERROR 2368: Cannot change owner of temporary table

ERROR 2377: Cannot convert column "string" from "string" to type "string"

ERROR 2392: Cannot delete from a view

ERROR 2399: Cannot display a value of type any

ERROR 2400: Cannot display a value of type anyelement

ERROR 2401: Cannot display a value of type internal

ERROR 2402: Cannot display a value of type language_handler
```

ERROR 2403: Cannot display a value of type opaque

ERROR 2404: Cannot display a value of type trigger

ERROR 2407: Cannot drop a table column when a node is down

ERROR 2411: Cannot drop column "*string*" since it is referenced in the partition expression

ERROR 2412: Cannot drop column "*string*" since it is referenced in the primary key constraint

ERROR 2425: Cannot export virtual *string string*

ERROR 2443: Cannot insert into a view

ERROR 2458: Cannot mergeout uncommitted data in the presence of savepoints

ERROR 2461: Cannot moveout uncommitted data in the presence of savepoints

ERROR 2503: Cannot set a subfield to DEFAULT

ERROR 2504: Cannot set an array element to DEFAULT

ERROR 2532: Cannot update a view

ERROR 2533: Cannot Update/Merge with Limit clause without an Order By on all columns

ERROR 2546: Cannot use aggregate functions in default expressions

ERROR 2547: Cannot use analytic or time series aggregate functions in default expressions

ERROR 2549: Cannot use DISTINCT with user-defined transform functions

ERROR 2552: Cannot use meta function or non-deterministic function in PARTITION BY expression

ERROR 2556: Cannot use SAVEPOINT with uncommitted tuple mover enabled

ERROR 2557: Cannot use subqueries in default expressions

ERROR 2558: Cannot use subquery in EXECUTE parameter

ERROR 2559: Cannot use subquery in expressions within COPY

ERROR 2560: Cannot use subquery in PARTITION BY expression

ERROR 2561: Cannot use subquery in SEGMENTED BY expression

ERROR 2562: Cannot use Vertica's built-in file source and a UDSource in the same query

ERROR 2569: Catalog object *string* does not exist

ERROR 2602: Clause "NO PROJECTION" conflicts with the column list

ERROR 2603: Clause "NO PROJECTION" is supported only on temporary tables

ERROR 2618: CoerceToDomain is not supported

ERROR 2619: CoerceToDomainValue is not supported

ERROR 2628: Column "*string*" in PARTITION BY expression is not allowed, since it contains NULL
values

ERROR 2646: Column *string* has the NOT NULL constraint set and has no default value defined

ERROR 2648: Column *string* in PARTITION BY expression is not allowed, since it is not present i
n some projections

ERROR 2649: Column *string* in PARTITION BY expression is not allowed, since it may contain NULL
values

ERROR 2652: Column *string* occurred multiple times in the definition of Projection *string*

ERROR 2660: Column column *string* is no longer at position *value* in table *string*

ERROR 2667: Column name list is not allowed in CREATE TABLE / AS EXECUTE

ERROR 2672: Column type int2 is not supported

ERROR 2673: Column type int4 is not supported

ERROR 2676: Command *string* is not supported

ERROR 2679: COMMENT not supported for system objects

ERROR 2680: COMMENT not supported for this object type

ERROR 2692: Conditional UNION/INTERSECT/EXCEPT statements are not implemented

ERROR 2698: Conflicting or redundant column options

ERROR 2721: ConvertRowtypeExpr is not supported

ERROR 2725: Copy cannot return rejected rows from executor nodes

ERROR 2726: Copy cannot return rejected rows from more than one file

ERROR 2739: COPY force not null is available only in CSV mode, but CSV mode is not supported

ERROR 2740: COPY force quote is available only in CSV mode, but CSV mode is not supported

ERROR 2741: COPY FROM does not support the BINARY option

ERROR 2742: COPY FROM does not support the CSV option

ERROR 2743: COPY FROM does not support the OIDS option

ERROR 2744: COPY LOCAL does not support rejected row numbers with exceptions or rejected data options

ERROR 2751: COPY quote is available only in CSV mode, but CSV mode is not supported

ERROR 2770: Correlated EXISTS/NOT EXISTS subquery containing having clause with aggregates is not supported

ERROR 2772: Correlated EXISTS/NOT EXISTS subquery with limit 0 is not supported

ERROR 2773: Correlated EXISTS/NOT EXISTS with aggregate COUNT is not supported

ERROR 2776: Correlated EXISTS/NOT EXISTS with User Defined Aggregate is not supported

ERROR 2777: Correlated expression in ON clause is not supported

ERROR 2778: Correlated expression in set operator subquery is not supported

ERROR 2779: Correlated expressions in SELECT list of subquery are not supported

ERROR 2780: Correlated subqueries cannot have more than one level

ERROR 2781: Correlated subqueries with analytics in the select list is not supported

ERROR 2782: Correlated subqueries with no group by and a non-strict expression containing an a ggregate in the select list is not supported

ERROR 2783: Correlated subquery column in select/gby/oby not supported

ERROR 2784: Correlated subquery could not be flattened as a join

ERROR 2785: Correlated subquery could not get flattened, a correlated expression could not be treated as a join

ERROR 2786: Correlated subquery expression without aggregates and with limit is not supported

ERROR 2787: Correlated subquery expressions under OR not supported

ERROR 2788: Correlated subquery in expression with operator <> is not supported

ERROR 2790: Correlated subquery with aggregate and limit 0 is not supported

ERROR 2792: Correlated subquery with aggregate function COUNT is not supported

ERROR 2793: Correlated subquery with distinct/group by is not supported

ERROR 2794: Correlated subquery with having clause expression that involves aggregates and sub query is not supported

ERROR 2795: Correlated subquery with NOT IN is not supported

ERROR 2796: Correlated subquery with outer joins and uncorrelated exists is not supported

ERROR 2797: Correlated subquery with User Defined Aggregate is not supported

ERROR 2854: Could not find array type for data type *string*

ERROR 2856: Could not find column *string* in table *string*

ERROR 2942: CREATE ASSERTION is not supported

ERROR 2943: CREATE FUNCTION / INOUT parameters are not supported

ERROR 2944: CREATE FUNCTION / OUT parameters are not supported

ERROR 2980: Data type not supported

ERROR 2981: Data type not supported (*value*)

ERROR 2983: Database "*string*" does not exist

ERROR 2987: Database references are not supported: "*string.string.string*"

ERROR 3019: Default expressions may not refer to other columns with default expressions

ERROR 3020: Default expressions must not return a set

ERROR 3026: Defining query must have a from clause

ERROR 3115: DistinctExpr not supported

ERROR 3116: Distrib overrides are too restrictive. Can not find completed Join Order

ERROR 3118: DML on projection/view is not supported

ERROR 3119: DML query with a predicate that could not be pushed below joins and does not refer solely to the target table is not supported

ERROR 3123: DROP ASSERTION is not supported

ERROR 3126: DROP COLUMN over temporary tables is not supported

ERROR 3132: DROP SEQUENCE does not support CASCADE

ERROR 3135: drop_location for DATA locations is not supported

ERROR 3141: Dropping local and global objects in one statement is not supported

ERROR 3157: Dynamic load not supported

ERROR 3163: Embedded SQL involving local objects is not supported

ERROR 3174: ENCODED BY is supported in CREATE TABLE ... AS SELECT statement only

ERROR 3246: Error parsing distrib overrides (unexpected end of override); *string*

ERROR 3247: Error parsing distrib value; *string*

ERROR 3291: Event ANY_ROW is not supported

ERROR 3317: Executing when OPT:PLAN_ALL_NODES_ACTIVE option is set

ERROR 3343: Explicit JOIN clause contains a join predicate between relations previously joined

ERROR 3351: Expressions in COPY may not contain aggregate functions

ERROR 3352: Expressions in COPY may not contain analytic or Time Series Aggregate Functions

ERROR 3353: Expressions not supported in Times Series Aggregate Function

ERROR 3357: External tables only support files or a User Defined Source

ERROR 3403: FieldSelect is not supported

ERROR 3404: FieldStore is not supported

ERROR 3417: Final phase output size mismatch

ERROR 3420: First argument of date_part must be a constant string

ERROR 3434: For INSERT SELECT statement, replicated/broadcasted source data not supported

ERROR 3436: For SELECT DISTINCT, ORDER BY expressions must appear in the SELECT clause

```
ERROR 3451: Function string can't be used as a case expression

ERROR 3452: Function string can't be used in a boolean

ERROR 3453: Function string can't be used in a WHEN clause

ERROR 3454: Function string can't be used in as a segment expression

ERROR 3455: Function string can't be used with an operator

ERROR 3488: Group By, Order By, Aggregates, Having & limits not allowed in update/delete

ERROR 3510: IGNORE NULLS argument must be a Boolean constant

ERROR 3553: INHERITS not supported

ERROR 3566: Input of anonymous composite types is not implemented

ERROR 3600: Interpolated predicates can accept arguments of the same type only

ERROR 3601: Interpolated predicates can be part of AND expressions only

ERROR 3613: Interval units "string" not supported

ERROR 3821: Joins with an interpolated predicate can have a conjunctive expression containing
    equality predicates. The equality predicates cannot have expressions or column references
    with different modifiers

ERROR 3822: Joins with an interpolated predicate cannot have expressions or column references
    with different modifiers in any of the expressions

ERROR 3857: Library built with unsupported version of Vertica SDK [Version: string, Revision:
    string]

ERROR 3859: Library file [string] is not valid for language [string]

ERROR 3876: Locale must be a constant

ERROR 3900: MATCH PARTIAL is not supported

ERROR 3972: Multi-column subquery expressions can only be used with the =, <=> and <> operator
    s

ERROR 3973: Multi-column subquery type ALL can only be used with the = and <=> operators

ERROR 3974: Multi-column subquery type ANY can only be used with the =, <=> and <> operators

ERROR 4106: No single-source bulk loads have been executed in this session

ERROR 4147: Node issuing the query cannot be marked as down

ERROR 4160: Non-equality correlated subquery expression is not supported

ERROR 4170: Not a Star or Snow-Flake Query block

ERROR 4171: Not a Star or Snow-Flake Query block; dimension table not a star or snowflake

ERROR 4172: Not a Star or Snow-Flake Query block; no fact table found

ERROR 4173: Not a Star or Snow-Flake Query block; there are multiple fact tables

ERROR 4197: NULL value found in a column used by a subquery

ERROR 4228: ON COMMIT DROP not supported in CREATE TABLE

ERROR 4238: Only a temporary table projection can be pinned

ERROR 4248: Only inner joins are allowed in the projection defining query

ERROR 4256: Only relations and subqueries are allowed in the FROM clause

ERROR 4258: Only super user can call export_catalog with an output file name

ERROR 4259: Only super user can get the rebalance data script

ERROR 4263: Only superuser can drop system schema

ERROR 4264: Only superuser can rebalance data
```

ERROR 4265: Only superuser can rebalance data for replicated projections

ERROR 4266: Only superuser can rebalance data for segmented projections

ERROR 4280: Operator *string* (*value*) is not supported

ERROR 4281: Operator *string* is not supported for row expressions

ERROR 4298: ORDER BY on a UNION/INTERSECT/EXCEPT result must be on one of the result columns

ERROR 4299: ORDER mode not supported

ERROR 4306: OUTER join with broadcasted outer data not supported

ERROR 4307: OUTER or SEMI join - done through CROSS join and FILTER - with replicated outer an
d segmented inner not supported

ERROR 4308: OUTER relation in OUTER join is not the fact table nor a snowflake dimension table

ERROR 4309: Outer replicated/segmented input to a join cannot be resegmented

ERROR 4310: LEFTOUTER/SEMI/ANTI join with replicated/broadcasted outer data not supported

ERROR 4329: Partition Auto cannot be used with pattern matching

ERROR 4331: PARTITION BY expression cannot return a tuple

ERROR 4332: PARTITION BY expression has an unknown type

ERROR 4333: PARTITION BY expression may not contain aggregate functions

ERROR 4335: Partitioning expression not supported for temporary tables

ERROR 4336: Partitioning not supported for temporary tables

ERROR 4352: Pattern "E" is not supported

ERROR 4375: PINNED clause conflicts with KSAFE setting

ERROR 4376: PINNED clause is not supported in CREATE TABLE statement

ERROR 4412: Prepared statements are currently unsupported

ERROR 4465: Projection *string* of local temporary table cannot be created under user schema *str
ing*

ERROR 4471: Projection choices are too restrictive - cannot create correct join between tables

ERROR 4486: Projections are always created and persisted in the default Vertica locale. The cu
rrent locale is *string*

ERROR 4502: Query Repository has been deprecated

ERROR 4584: RENAME COLUMN over temporary tables is not supported

ERROR 4586: replicate_catalog has been shut off

ERROR 4628: Row Expressions are not supported in this context

ERROR 4631: ROW syntax is not supported

ERROR 4644: Scalar array expression cannot contain column references or subqueries

ERROR 4645: Scalar array op *string* (*value*) is not supported

ERROR 4664: Segmentation clause can not have offset in CREATE TABLE statement

ERROR 4665: Segmentation clause with offset conflicts with KSAFE setting

ERROR 4666: Segmentation expression must have integer type

ERROR 4671: SELECT FOR UPDATE cannot be applied to a function

ERROR 4672: SELECT FOR UPDATE cannot be applied to a join

ERROR 4673: SELECT FOR UPDATE cannot be applied to NEW or OLD

ERROR 4674: SELECT FOR UPDATE is not allowed with EXTERNAL TABLES

ERROR 4675: SELECT FOR UPDATE is not allowed with libraries

ERROR 4676: SELECT FOR UPDATE is not allowed with sequences

ERROR 4677: SELECT FOR UPDATE is not allowed with UNION/INTERSECT/EXCEPT

ERROR 4678: SELECT FOR UPDATE is not allowed with views

ERROR 4680: Self joins in UPDATE statements are not allowed

ERROR 4703: Sequence cannot be moved between system schema and user schema

ERROR 4711: Sequence or IDENTITY/AUTO_INCREMENT column in merge query is not supported

ERROR 4714: Sequences are not allowed in default expressions of local temp tables

ERROR 4715: Sequences cannot be called in views

ERROR 4716: Sequences cannot be created under system schemas

ERROR 4728: Set Operator *string* ALL not supported

ERROR 4730: Set Operator queries without a FROM clause are not supported

ERROR 4733: SET SCHEMA over temporary tables is not supported

ERROR 4735: Set-valued function called in context that cannot accept a set

ERROR 4747: SetToDefault is not supported

ERROR 4786: Statement *string* is not supported

ERROR 4808: Subqueries are not supported as the left hand argument to another subquery

ERROR 4809: Subqueries are not supported in the ORDER BY of a timeseries OVER clause

ERROR 4810: Subqueries are not supported in the ORDER BY of an analytic function OVER clause

ERROR 4812: Subqueries are not supported in the PARTITION BY of an analytic function OVER clau
se

ERROR 4816: Subqueries in the ON clause are not supported

ERROR 4817: Subqueries in the SELECT or ORDER BY are not supported if the query has aggregates
and the subquery is not part of the GROUP BY

ERROR 4818: Subqueries in the SELECT or ORDER BY are not supported if the subquery is not part
of the GROUP BY

ERROR 4820: Subqueries in UPDATE/DELETE/MERGE is not supported

ERROR 4821: Subqueries not allowed in target of insert

ERROR 4822: Subqueries referring to no outer columns in HAVING clause when query has aggregate
s and no GROUP BY are not supported

ERROR 4824: Subquery aggregate expression that refers a correlated column is not supported

ERROR 4839: Subquery type ARRAY is not supported

ERROR 4842: Subquery without a from clause is not supported

ERROR 4850: Support for UPDATE/DELETE/MERGE is not enabled

ERROR 4854: SyncMarkers are not supported

ERROR 4865: System table *string* cannot be created under user schema *string*

ERROR 4869: System view "*string*" cannot be dropped

ERROR 4870: System view *string* cannot be created under user schema *string*

ERROR 4884: Table *string* cannot be created under system schema *string*

ERROR 4897: Table cannot be moved between system schema and user schema

ERROR 4910: Table revalidation error

ERROR 4918: Temporary Sequences are not supported

```
ERROR 4933: The argument types in a subquery expression in the where/having clause do not matc
    h
```

```
ERROR 4938: The constant value following the LIMIT clause cannot be negative
```

```
ERROR 4939: The constant value following the OFFSET clause cannot be negative
```

```
ERROR 4948: The fourth input argument of TIME_SLICE must be START or END
```

```
ERROR 4960: The ORDER BY ... USING clause is not supported
```

```
ERROR 4966: The second parameter of export_catalog is invalid: string
```

```
ERROR 4968: The slice length parameter of TIME_SLICE must be a positive integer
```

```
ERROR 5005: Time Series Aggregate Function with interpolation scheme LINEAR may only have an I
    NTEGER or FLOAT type as its first argument
```

```
ERROR 5016: Time units "string" not supported
```

```
ERROR 5023: Timeseries output functions are not supported in the ORDER BY of a timeseries OVER
    clause
```

```
ERROR 5028: Timestamp units "string" not supported
```

```
ERROR 5110: Type string (value) is not supported
```

```
ERROR 5159: Uncorrelated EXISTS subqueries are not supported when the query has both HAVING cl
    ause subqueries involving aggregates and when the query has either OUTER JOINS or NOT IN
    subqueries
```

```
ERROR 5160: Uncorrelated EXISTS subqueries in HAVING clause when query has aggregates and no G
    ROUP BY are not supported
```

```
ERROR 5195: UNIQUE predicate is not supported
```

```
ERROR 5262: Unsafe use of string constant with Unicode escapes
```

```
ERROR 5264: Unsupported access to session-scoped (LOCAL) object
```

```
ERROR 5270: Unsupported COPY command clause
```

```
ERROR 5275: Unsupported Join in From clause
```

```
ERROR 5276: Unsupported Join in From clause: FULL OUTER JOINS not supported
```

```
ERROR 5278: Unsupported join of two non-alike segmented projections
```

```
ERROR 5280: Unsupported mix of Joins
```

```
ERROR 5284: Unsupported query syntax
```

```
ERROR 5289: Unsupported subquery expression
```

```
ERROR 5291: Unsupported use of aggregates
```

```
ERROR 5292: Unsupported use of cursors
```

```
ERROR 5293: Unsupported use of DISTINCT clause
```

```
ERROR 5294: Unsupported use of FROM clause
```

```
ERROR 5295: Unsupported use of GROUP BY or DISTINCT clause
```

```
ERROR 5296: Unsupported use of HAVING clause
```

```
ERROR 5297: Unsupported use of LIMIT/OFFSET clause
```

```
ERROR 5298: Unsupported use of ORDER BY clause
```

```
ERROR 5299: Unsupported use of outer joins
```

```
ERROR 5300: Unsupported use of query/subquery without FROM clause
```

```
ERROR 5301: Unsupported use of sub-queries
```

```
ERROR 5302: Unsupported use of target relation
```

ERROR 5303: Unsupported use of UDF in WHERE clause

ERROR 5304: Unsupported use of UNION/INTERSECT/EXCEPT

ERROR 5313: Update is disallowed on Primary/Foreign Keys columns. Use Delete followed by Insert instead

ERROR 5314: UPDATE may not refer to tables in prejoin projections

ERROR 5366: User defined aggregate cannot be used in query with other distinct aggregates

ERROR 5388: User has insufficient privilege on *string* *string*

ERROR 5392: User must have the DBDUSER role to run the database designer

ERROR 5396: User projection *string* cannot be created under system schema *string*

ERROR 5402: User-defined transform functions are not supported in the ORDER BY clause

ERROR 5407: VALINDEX column must be the first column in ORDER BY list

ERROR 5426: Vertica currently allows a maximum of *value* physical storage containers per projection

ERROR 5427: Vertica does not support GRANT / REVOKE ON LANGUAGE

ERROR 5428: Vertica does not support GRANT / REVOKE ON TABLESPACE

ERROR 5447: View *string* cannot be created under system schema *string*

ERROR 5456: Volatile functions may not be used in fillers when other computed columns refer to them

ERROR 5465: Window frame exclusion is not supported

ERROR 5530: Audit of external tables is not supported

ERROR 5537: Cannot alter user-defined type "*string*" of column "*string*"

ERROR 5550: COPY from UDSource does not support rejected row numbers with exceptions or rejected data options

ERROR 5551: COPY LOCAL cannot process more than ONE NATIVE or NATIVE VARCHAR file at a time

ERROR 5562: Creating temp tables by LIKE clause is not supported

ERROR 5595: Invalid argument type *string* in function *string*

ERROR 5607: Language of replacement library [*string*] must match language of existing library [*string*]

ERROR 5681: Unsupported base type *string* for User-defined type *string*

ERROR 5698: Cannot export statistics for the specified object

ERROR 5725: Size specification not supported for User Defined Type *string*

ERROR 5731: The second parameter must be a table/projection/column name

ERROR 5758: Can not drop Filesystem proc *string*

ERROR 5759: Can not drop library "*string*": referenced by storage locations

ERROR 5763: Can't create a managed external table with non-file sources

ERROR 5764: Cannot alter the data type of a table column when a node is down

ERROR 5781: Cannot use meta function or non-deterministic function in SEGMENTED BY expression

ERROR 5859: Due to the data isolation of temp tables with an on-commit-delete-rows policy, the compute_flextable_keys() and compute_flextable_keys_and_build_view() functions cannot access this table's data

ERROR 5864: Error parsing table (invalid table): *string*

ERROR 5914: HCatalog schema *string* not permitted in search path

ERROR 5990: Projection *string* cannot be created under hcatalog schema *string*

```
ERROR 5992: Projection cannot be created for HCatalog table string
ERROR 6005: Remote table string.string found in design query
ERROR 6019: Sequence string cannot be created under hcatalog schema string
ERROR 6020: Sequence string cannot be moved between system schema and hcatalog schema string
ERROR 6023: Setting the CPU affinity of the built-in pool "string" is not supported
ERROR 6038: Table string cannot be created under hcatalog schema string
ERROR 6039: Table string cannot be moved under hcatalog schema string
ERROR 6092: Unsupported access to flex table: No string support
ERROR 6108: View string cannot be created under hcatalog schema string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 0B000

This topic lists the error associated with the SQLSTATE 0B000.

SQLSTATE 0B000 Description

ERRCODE_INVALID_TRANSACTION_INITIATION

Error messages associated with this SQLState

```
ERROR 2321: Can't start a Transaction in this context
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 0LV01

This topic lists the error associated with the SQLSTATE 0LV01.

SQLSTATE 0LV01 Description

ERRCODE_INVALID_GRANT_OPERATION

Error messages associated with this SQLState

```
ERROR 2120: Admin option for a role cannot be granted to string"public"
ERROR 2601: Circular assignation of roles is not allowed
ERROR 3484: Grant option for a privilege cannot be granted to "public"
ERROR 3485: Grant option for a privilege cannot be granted to (and thus revoked from) "public"
ERROR 3486: Grant options cannot be granted back to your own grantor
ERROR 3616: Invalid string statement
ERROR 3719: Invalid option specified for string statement
ERROR 3723: Invalid privilege type "string"
ERROR 3724: Invalid privilege type string for aggregate function
ERROR 3725: Invalid privilege type string for analytic function
ERROR 3726: Invalid privilege type string for database
ERROR 3727: Invalid privilege type string for function
ERROR 3728: Invalid privilege type string for library
ERROR 3729: Invalid privilege type string for procedure
ERROR 3730: Invalid privilege type string for relation
ERROR 3731: Invalid privilege type string for resource pool
ERROR 3732: Invalid privilege type string for schema
ERROR 3733: Invalid privilege type string for sequence
ERROR 3734: Invalid privilege type string for storage location
ERROR 3735: Invalid privilege type string for transform
ERROR 3745: Invalid role name string
ERROR 4056: New string
ERROR 4613: Role "string" cannot be set as default
ERROR 5601: Invalid privilege type string for filter function
ERROR 5602: Invalid privilege type string for parser function
ERROR 5603: Invalid privilege type string for source function
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22000

This topic lists the error associated with the SQLSTATE 22000.

SQLSTATE 22000 Description

ERRCODE_DATA_EXCEPTION

Error messages associated with this SQLState

```
ERROR 3646: Invalid Datum pointer
ERROR 4163: Non-positive value supplied to randomint: value
ERROR 4921: Test Error @string
ERROR 4922: Test Error from @string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that

may help you resolve these errors.

Error Messages Associated with SQLSTATE 22001

This topic lists the error associated with the SQLSTATE 22001.

SQLSTATE 22001 Description

ERRCODE_STRING_DATA_RIGHT_TRUNCATION

Error messages associated with this SQLState

```
ERROR 2991: Date 'string'string too long for type string(value)
ERROR 3426: Float 'string'string too long for type string
ERROR 3589: Integer 'string'string is too long for type string(value)
ERROR 3605: Interval 'string'string too long for type string(value)
ERROR 4208: Numeric 'string' is too long for type string
ERROR 4315: Padded octet length (value) exceeds the value octet limit
ERROR 4604: Result (value characters) exceeds the field width (value)
ERROR 4800: String of value octets is too long for type string(value)
ERROR 5004: Time 'string'string too long for type string(value)
ERROR 5024: Timestamp 'string'string too long for type string(value)
ERROR 5032: Timestamptz 'string'string too long for type string(value)
ERROR 5035: Timetz 'string'string too long for type string(value)
ERROR 5417: Value too long for type character varying(value)
ERROR 5418: Value too long for type character(value)
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22003

This topic lists the error associated with the SQLSTATE 22003.

SQLSTATE 22003 Description

ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE

Error messages associated with this SQLState

```
ERROR 2429: Cannot find matching query in the system
ERROR 2828: Could not convert 'string'string to an int8
ERROR 3425: Float "value" is out of range for type string
ERROR 3675: Invalid input for string, exceeds 32 bits: "string"

ERROR 3676: Invalid input for string, exceeds 64 bits: "string"

ERROR 3786: Invalid value for float: "string"

ERROR 4200: Number of buckets must be a positive integer

ERROR 4361: Percentile value must be a number between 0 and 1

ERROR 4704: Sequence exceeded max value

ERROR 4705: Sequence exceeded min value

ERROR 4756: Smoothing factor must between 0 and 1

ERROR 4795: String "string"string is out of range as a float8

ERROR 4796: String "string"string is out of range as an int8

ERROR 4845: Sum() overflowed

ERROR 5408: Value "value" is out of range for type string

ERROR 5409: Value "string" is out of range for type int8

ERROR 5411: Value exceeds range of type string

ERROR 5412: Value is too long for type string: "value"

ERROR 6063: Total number of significant digits for value string is more than what is defined.
    Buffer size is value while actual length of word is value instead
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22004

This topic lists the error associated with the SQLSTATE 22004.

SQLSTATE 22004 Description

ERRCODE_NULL_VALUE_NOT_ALLOWED

Error messages associated with this SQLState

```
ERROR 2110: ACL arrays must not contain null values

ERROR 2501: Cannot set a NOT NULL column (value) to a NULL value in value statement

ERROR 2502: Cannot set a NOT NULL column (string) to a NULL value in INSERT/UPDATE statement

ERROR 2514: Cannot set NOT NULL columns (string) to a NULL value in INSERT/UPDATE statement

ERROR 4195: NULL value detected in data partitioning expression
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22007

This topic lists the error associated with the SQLSTATE 22007.

SQLSTATE 22007 Description

ERRCODE_INVALID_DATETIME_FORMAT

Error messages associated with this SQLState

```
ERROR 2171: AM/PM hour (value) must be between 1 and 12
ERROR 2364: Cannot calculate day of year without year information
ERROR 3439: Format string is invalid for an Interval value
ERROR 3535: Inconsistent use of year value and "BC"
ERROR 3647: Invalid day-of-week 'string'
ERROR 3679: Invalid input syntax for string: "string"
ERROR 3721: Invalid partition key
ERROR 3785: Invalid value for string: "string"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22008

This topic lists the error associated with the SQLSTATE 22008.

SQLSTATE 22008 Description

ERRCODE_DATETIME_FIELD_OVERFLOW

Error messages associated with this SQLState

```
ERROR 2992: Date/time field value out of range: "string"
ERROR 4065: next_day(infinity, DOW) is not defined
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22009

This topic lists the error associated with the SQLSTATE 22009.

SQLSTATE 22009 Description

ERRCODE_INVALID_TIME_ZONE_DISPLACEMENT_VALUE

Error messages associated with this SQLState

```
ERROR 3768: Invalid timezone interval displacement
```

```
ERROR 5044: Timezone displacement out of range: "string"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 2200B

This topic lists the error associated with the SQLSTATE 2200B.

SQLSTATE 2200B Description

ERRCODE_ESCAPE_CHARACTER_CONFLICT

Error messages associated with this SQLState

```
ERROR 2699: Conflicting or redundant options
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 2200D

This topic lists the error associated with the SQLSTATE 2200D.

SQLSTATE 2200D Description

ERRCODE_INVALID_ESCAPE_OCTET

Error messages associated with this SQLState

```
ERROR 3285: ESCAPE strings must be a single octet, not "value"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22011

This topic lists the error associated with the SQLSTATE 22011.

SQLSTATE 22011 Description

ERRCODE_SUBSTRING_ERROR

Error messages associated with this SQLState

```
ERROR 4034: Negative count not allowed
ERROR 4035: Negative length not allowed
ERROR 4036: Negative or zero substring start position not allowed
ERROR 4039: Negative substring length not allowed
ERROR 4784: Start position cannot be 0
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22012

This topic lists the error associated with the SQLSTATE 22012.

SQLSTATE 22012 Description

ERRCODE_DIVISION_BY_ZERO

Error messages associated with this SQLState

```
ERROR 3117: Division by zero
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22015

This topic lists the error associated with the SQLSTATE 22015.

SQLSTATE 22015 Description

ERRCODE_INTERVAL_FIELD_OVERFLOW

Error messages associated with this SQLState

```
ERROR 3606: Interval field value out of range: "string"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that

may help you resolve these errors.

Error Messages Associated with SQLSTATE 22019

This topic lists the error associated with the SQLSTATE 22019.

SQLSTATE 22019 Description

ERRCODE_INVALID_ESCAPE_CHARACTER

Error messages associated with this SQLState

```
ERROR 2729: COPY DELIMITER for column string must be a single character
ERROR 2730: COPY delimiter must be a single character
ERROR 2731: COPY ENCLOSED BY cannot be a whitespace character
ERROR 2732: COPY ENCLOSED BY for column string cannot be a whitespace character
ERROR 2733: COPY ENCLOSED BY for column string must be a single character
ERROR 2734: COPY ENCLOSED BY must be a single character
ERROR 2736: COPY ESCAPE AS for column string must be a single character
ERROR 2737: COPY ESCAPE must be a single character
ERROR 2758: COPY TRIM for column string must be an empty string or a single character
ERROR 2759: COPY trim must be an empty string or a single character
ERROR 3284: ESCAPE strings must be a single character, not "value"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 2201B

This topic lists the error associated with the SQLSTATE 2201B.

SQLSTATE 2201B Description

ERRCODE_INVALID_REGULAR_EXPRESSION

Error messages associated with this SQLState

```
ERROR 3742: Invalid regexp match_param: 'character'
ERROR 4552: Regexp match or recursion limit exceeded (rc value)
ERROR 4553: Regexp pattern error at offset value: string
ERROR 4554: Regexp pattern study error: string
ERROR 5064: Too many regular expression subexpressions
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 2201G

This topic lists the error associated with the SQLSTATE 2201G.

SQLSTATE 2201G Description

ERRCODE_INVALID_ARGUMENT_FOR_WIDTH_BUCKET_FUNCTION

Error messages associated with this SQLState

```
ERROR 2939: Count must be greater than zero
ERROR 3888: Lower and upper bounds must be finite
ERROR 3889: Lower bound cannot equal upper bound
ERROR 4277: Operand, lower bound and upper bound cannot be NaN
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22021

This topic lists the error associated with the SQLSTATE 22021.

SQLSTATE 22021 Description

ERRCODE_CHARACTER_NOT_IN_REPERTOIRE

Error messages associated with this SQLState

```
ERROR 4551: Regexp encountered an invalid UTF-8 character
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22023

This topic lists the error associated with the SQLSTATE 22023.

SQLSTATE 22023 Description

ERRCODE_INVALID_PARAMETER_VALUE

Error messages associated with this SQLState

ERROR 2007: *string* can not be greater than PASSWORD_MAX_LENGTH *value*

ERROR 2008: *string* can not be set to a negative number

ERROR 2028: *string* exceptions and rejected_data can not be the same filename

ERROR 2033: *string* input file and exceptions can not be the same filename

ERROR 2034: *string* input file and rejected_data can not be the same filename

ERROR 2042: *string* must be a positive integer

ERROR 2048: *string* Path [*string*] is a directory

ERROR 2049: *string* Path [*string*] is a socket

ERROR 2051: *string* Record terminator length (*value*) is larger than load read buffer size (*valu e*)

ERROR 2056: *string* Unrecognized format '*string*' for column *value*

ERROR 2071: '*string*' is not a valid size description

ERROR 2075: @INCLUDE without filename in timezone file "*string*", line *value*

ERROR 2077: [*string*] cannot be dropped. There will be no storage locations for data files

ERROR 2078: [*string*] cannot be dropped. There will be no storage locations for temporary files

ERROR 2079: [*string*] cannot be retired. There will be no storage locations for data files

ERROR 2080: [*string*] cannot be retired. There will be no storage locations for temporary files

ERROR 2081: [*string*] is not a valid storage location on node *string*

ERROR 2108: ACL array contains wrong data type

ERROR 2109: ACL arrays must be one-dimensional

ERROR 2158: All columns of soft unique key statistics must be from the same table

ERROR 2194: analyze_statistics: Can not analyze statistics of a non-local temporary table/proj ection '*string*'

ERROR 2195: analyze_statistics: Can not analyze statistics of a virtual table/projection *strin g*

ERROR 2196: analyze_statistics: Cannot analyze statistics of a virtual table *string*

ERROR 2197: analyze_statistics: invalid accuracy *value* A number between 0 and 100 is required

ERROR 2254: Bad snapshot name '*string*' (cannot contain / or start with a .)

ERROR 2298: Can not lock/unlock super user account

ERROR 2300: Can not reuse any recent passwords

ERROR 2301: Can not reuse current password

ERROR 2302: Can not reuse the previous *value* passwords

ERROR 2317: Can't purge projection(s); AHM is at epoch 0

ERROR 2319: Can't set a REJECTED file on node '*string*', which the current query is not executi ng on

ERROR 2320: Can't set an EXCEPTIONS file on node '*string*', which the current query is not exec uting on

ERROR 2365: Cannot calculate week number without year information

ERROR 2370: Cannot close a protected session

ERROR 2414: Cannot drop extended statistics on a projection (*string*)

ERROR 2415: Cannot drop extended statistics on projection *string*. Dropping base statistics onl y

ERROR 2452: Cannot load data from node *string* as it is down

ERROR 2457: Cannot merge partitions in multiple tables at the same time

ERROR 2468: Cannot partition by value multiple tables at the same time

ERROR 2478: Cannot release savepoint; no transaction in progress

ERROR 2500: Cannot set *string* maxConcurrency to unlimited

ERROR 2508: Cannot set maxConcurrency of *string* pool to 0

ERROR 2509: Cannot set maxMemorySize of *string* pool to *string* [*value* KB], as it is above 75%% [75%% = *value* KB]

ERROR 2510: Cannot set maxMemorySize of *string* pool to none, as this could prevent moveout fro m running

ERROR 2511: Cannot set maxMemorySize of recovery pool to *string* [*value* KB], as it is below 2 5%% [*value* KB]

ERROR 2513: Cannot set memorySize of general pool

ERROR 2523: Cannot specify exceptions or rejected-data files ON ANY NODE

ERROR 2540: Cannot use 0 for a key, used internally

ERROR 2548: Cannot use both COPY LOCAL and ON ANY NODE: LOCAL files are stored on the client, not on any Vertica node

ERROR 2621: Collection type must be specified

ERROR 2624: Column "*string*" does not exist

ERROR 2653: Column *string* of projection *string* has ACCESSRANK < 0

ERROR 2695: Conflicting "datestyle" keywords

ERROR 2720: Conversion to timezone "*string*" failed

ERROR 2722: COPY .. LOCAL cannot store *string* on a Vertica node

ERROR 2723: COPY ... LOCAL can read files from the client only

ERROR 2724: COPY ... LOCAL can read files with same compression only

ERROR 2727: COPY column option *string* not supported with format *string*

ERROR 2728: COPY delimiter *string*must not appear in the NULL specification

ERROR 2735: COPY ENCLOSING CHARACTER *string*must not appear in the NULL specification

ERROR 2748: COPY NULL must be an empty string or a single character for FIXED WIDTH data

ERROR 2749: COPY option *string* not supported

ERROR 2750: COPY option *string* not supported with format *string*

ERROR 2752: COPY RECORD TERMINATOR must be at least ONE character long

ERROR 2753: COPY REJECTMAX should be >= 0

ERROR 2756: COPY skip characters should be >= 0

ERROR 2757: COPY skip should be >= 0

ERROR 2760: COPY WITH PARSER Error (column *value*): Parser specified a column of type [*string*]; table needs [*string*]

ERROR 2761: COPY WITH PARSER Error: Parser specified *value* column(s); table needs *value* column (s)

ERROR 2765: COPY: width and length of null string does not match for column *string*

ERROR 2766: COPY: width for column *string* has to be greater than 0

ERROR 2830: Could not convert to timezone "*string*"

ERROR 2932: Couldn't find the specified task

ERROR 2950: Current design does not meet the requirements for K = *value*
        Current design is valid for K *string value*
        string

ERROR 2963: CURRENT_TIME(*value*) precision must not be negative

ERROR 2964: CURRENT_TIME(*value*) precision reduced to maximum allowed, *value*

ERROR 2965: CURRENT_TIMESTAMP(*value*) precision must not be negative

ERROR 2966: CURRENT_TIMESTAMP(*value*) precision reduced to maximum allowed, *value*

ERROR 2993: Datepart "*string*" not recognized

ERROR 2994: Datepart is invalid

ERROR 3006: DDL statement interfered with snapshot; an object no longer exists

ERROR 3007: DDL statement interfered with this statement

ERROR 3012: DECIMAL precision *value* must be between 1 and *value*

ERROR 3013: DECIMAL scale *value* must be between 0 and precision *value*

ERROR 3032: Delimiter and record terminator cannot be the same value

ERROR 3033: Delimiter and record terminator for *string* cannot be the same value

ERROR 3137: drop_statistics: Can not drop base or histogram statistics of a non-local temporar
    y table/projection *string*

ERROR 3138: drop_statistics: Can not drop statistics for a virtual table/projection *string*

ERROR 3139: drop_statistics: Invalid stats type '*string*'. Valid values are 'base', 'histogram
    s' and 'extended'

ERROR 3168: ENCLOSED BY and delimiter *string*can not be the same value

ERROR 3169: ENCLOSED BY and ESCAPE AS *string*can not be the same value

ERROR 3170: ENCLOSED BY and record terminator *string*can not be the same value

ERROR 3178: ENFORCELENGTH cannot be specified for *string*

ERROR 3280: ESCAPE AS and delimiter *string*can not be the same value

ERROR 3281: ESCAPE AS and NULL specification *string*can not be the same value

ERROR 3282: ESCAPE AS and record terminator *string*can not be the same value

ERROR 3383: Failed to parse object name string

ERROR 3423: Fixed width record size (*value*) is too large. Record size has to be lesser than *va
    lue* (0x*value*)

ERROR 3424: Fixed width record size is too large. Record size has to be lesser than *value* (0x*v
    alue*)

ERROR 3440: Format cannot be specified for *string*

ERROR 3503: ICU *string* error: '*string*'

ERROR 3505: ICU does not support locale '*string*'

ERROR 3513: Illegal argument to change_runtime_priority: NULL

ERROR 3514: Illegal argument to set_config_parameter: NULL

ERROR 3524: In the SAMPLE STORAGE n or SAMPLE STORAGE n,b clause, n must be a constant greater
    than or equal to 0

ERROR 3525: In the SAMPLE STORAGE n PERCENT or SAMPLE STORAGE n PERCENT,b clause, n must be a
    constant greater than or equal to 0 and less than or equal to 100

ERROR 3526: In the SAMPLE STORAGE n PERCENT,b clause, n must be a constant greater than or equal to 0 and less than or equal to 100, while b must be a constant greater than or equal to 0

ERROR 3527: In the SAMPLE STORAGE n,b clause, both n and b must be constants greater than or equal to 0

ERROR 3528: In the SAMPLE STORAGE n,b or SAMPLE STORAGE n PERCENT,b clause, b must be a constant greater than or equal to 0

ERROR 3540: Incorrect statement ID for session

ERROR 3541: Increase in pool size to *string* [*value* KB] causes general pool to fall below minimum [25%% = *value* KB]

ERROR 3607: INTERVAL leading field precision increased to *value*

ERROR 3608: INTERVAL leading field precision reduced to *value*

ERROR 3610: INTERVAL SECOND precision reduced to *value*

ERROR 3612: Interval units "*value*" not recognized

ERROR 3618: Invalid accuracy value for analyze_histogram

ERROR 3632: Invalid collection type *string* specified

ERROR 3652: Invalid Directives type: *string*

ERROR 3673: Invalid hint identifier '*string*'

ERROR 3686: Invalid interval value for timezone

ERROR 3688: Invalid K value: *value* K cannot be less than zero

ERROR 3689: Invalid K value: *value* Maximum K value for *value* nodes is: *value*

ERROR 3692: Invalid limit type (*string*): must be HIGH or LOW

ERROR 3695: Invalid list syntax for "datestyle"

ERROR 3707: Invalid node: [*string*]

ERROR 3710: Invalid number for timezone offset in timezone file "*string*", line *value*

ERROR 3741: Invalid range

ERROR 3743: Invalid resource type (*string*)

ERROR 3746: Invalid runtime priority string

ERROR 3750: Invalid service name for '*string*'

ERROR 3759: Invalid syntax in timezone file "*string*", line *value*

ERROR 3767: Invalid timezone file name "*string*"

ERROR 3777: Invalid Usage type: *string*

ERROR 3780: Invalid user/role name "*string*"

ERROR 3783: Invalid value *string*=*string*

ERROR 3787: Invalid value for parameter

ERROR 3788: Invalid value for parameter *string*: *string*

ERROR 3789: Invalid value for search path: "*string*"

ERROR 3840: Keyword '*string*' (*string*=*string*) is not supported

ERROR 3845: Latency should be > 0

ERROR 3852: Length for type *string* cannot exceed *value*

ERROR 3853: Length for type *string* must be at least 1

ERROR 3877: LOCALTIME(*value*) precision must not be negative

ERROR 3878: LOCALTIME(*value*) precision reduced to maximum allowed, *value*

ERROR 3879: LOCALTIMESTAMP(*value*) precision must not be negative

ERROR 3880: LOCALTIMESTAMP(*value*) precision reduced to maximum allowed, *value*

ERROR 3912: maxMemorySize of *string* [*value* KB] is not in bounds [max is *value* KB]

ERROR 3920: memoryCap of *string* (*value* KB) would exceed [*value* KB]

ERROR 3922: memorySize *string* [*value* KB] would exceed maxMemorySize *string* [*value* KB]

ERROR 3923: memorySize of *string* [*value* KB] would exceed [*value* KB]

ERROR 3960: Missing timezone abbreviation in timezone file "*string*", line *value*

ERROR 3961: Missing timezone offset in timezone file "*string*", line *value*

ERROR 3967: More than one *string* specified for a node

ERROR 4027: Must supply a CATALOGPATH

ERROR 4028: Must supply a HOSTNAME

ERROR 4037: Negative run time cap is not allowed

ERROR 4038: Negative runTimeCap is not allowed

ERROR 4084: No interruptible statement running

ERROR 4089: No objects specified

ERROR 4174: Not allowed to cancel statement

ERROR 4175: Not allowed to close session

ERROR 4186: NULL is an invalid K value

ERROR 4187: NULL is invalid object name for analyze_extended_statistics

ERROR 4188: NULL is invalid object name for analyze_histogram

ERROR 4189: NULL is invalid object name for drop_statistics

ERROR 4190: NULL is invalid scope type for analyze_extended_statistics

ERROR 4191: NULL is invalid statistics type for analyze_extended_statistics

ERROR 4192: NULL is invalid statistics type for drop_statistics

ERROR 4194: NULL string and record terminator *string*can not be the same value

ERROR 4211: NUMERIC precision *value* must be between 1 and *value*

ERROR 4212: NUMERIC scale *value* must be between 0 and precision *value*

ERROR 4222: Occurrence number must be > 0

ERROR 4250: Only ONE exception file should be specified for a LOCAL copy

ERROR 4252: Only ONE rejected data file should be specified for a LOCAL copy

ERROR 4318: Parameter *string* in default profile can not be set to DEFAULT

ERROR 4319: Parameter *string* may not exceed 9999

ERROR 4330: PARTITION BY clause must contain table columns in a valid expression

ERROR 4334: Partition key too long

ERROR 4344: PASSWORD_MAX_LENGTH must be within the range from *value* to *value*

ERROR 4345: PASSWORD_MIN_DIGITS + PASSWORD_MIN_SYMBOLS + PASSWORD_MIN_LETTERS *value* can not be greater than PASSWORD_MAX_LENGTH *value*

ERROR 4346: PASSWORD_MIN_DIGITS + PASSWORD_MIN_SYMBOLS + PASSWORD_MIN_LOWERCASE_LETTERS + PASSWORD_MIN_UPPERCASE_LETTERS *value* can not be greater than PASSWORD_MAX_LENGTH *value*

ERROR 4347: Path cannot be an empty string

ERROR 4399: populate_projection_statistics doesn't take empty parameter string

ERROR 4400: populate_projection_statistics: Can not populate statistics for a virtual table/projection *string*

ERROR 4401: populate_projection_statistics: Can only populate statistics for a projection, not for a table

ERROR 4402: populate_projection_statistics: Invalid table/projection name *string*

ERROR 4406: Precision for type float must be at least 1 bit

ERROR 4407: Precision for type float must be less than 54 bits

ERROR 4408: Precision must be less than *value*; result would be numeric(*value*,*value*)

ERROR 4454: Projection *string* cannot be analyzed, because it is not up to date

ERROR 4456: Projection *string* cannot drop statistics, because it is not up to date

ERROR 4529: Rebalance skew percent must be in the range [0,100]

ERROR 4556: Regexp starting position must be greater than zero

ERROR 4595: Resource pool "*string*" is an internal pool and cannot be dropped

ERROR 4606: Retention settings must be less than 2TB

ERROR 4639: Run time cap cannot exceed 1 year

ERROR 4642: runTimeCap cannot exceed 1 year

ERROR 4647: Scaling factor must be greater than zero

ERROR 4648: Scaling factor must be less than 33

ERROR 4653: Schema *string* is virtual

ERROR 4701: Sequence *string* is already owned by *string*

ERROR 4702: SEQUENCE CACHE should be greater than 0

ERROR 4708: SEQUENCE MAXVALUE is too large and will overflow

ERROR 4709: SEQUENCE MINVALUE is too small and will underflow

ERROR 4710: SEQUENCE MINVALUE should be lesser than MAXVALUE

ERROR 4712: SEQUENCE START WITH should be between MINVALUE and MAXVALUE

ERROR 4723: SET *string* takes only one argument

ERROR 4745: Setting sysdata maxMemorySize below 4 MB to *string* [*value* KB] will prevent system table queries from running

ERROR 4766: Specified too few widths for the given number of columns

ERROR 4770: Specify at least one table-column for soft unique key statistics

ERROR 4802: STROKE collations are not supported

ERROR 4807: Subnet mask is empty

ERROR 4862: System pool priority must be between -110 and 110 inclusive

ERROR 4889: Table *string* is already owned by *string*

ERROR 4892: Table *string* is not partitioned

ERROR 4893: Table *string* is session scoped

ERROR 4894: Table *string* is virtual

ERROR 4923: That password is not acceptable

ERROR 4937: The confidence level must be between 0 and 100 inclusive.
string

```
ERROR 4961: The permissible error must between 0 and 100 inclusive.
        string

ERROR 4985: There is no reason to set string.string.  Consult documentation

ERROR 5002: Throughput should be > 0

ERROR 5014: Time units "value" not recognized

ERROR 5015: Time units "string" not recognized

ERROR 5019: TIME(value)string precision must not be negative

ERROR 5020: TIME(value)string precision reduced to maximum allowed, value

ERROR 5026: Timestamp units "value" not recognized

ERROR 5027: Timestamp units "string" not recognized

ERROR 5029: TIMESTAMP(value) precision reduced to maximum allowed, value

ERROR 5030: TIMESTAMP(value)string precision must not be negative

ERROR 5031: TIMESTAMP(value)string precision reduced to maximum allowed, value

ERROR 5034: TIMESTAMPTZ(value) precision must not be negative

ERROR 5036: TIMETZ(value) precision must not be negative

ERROR 5037: TIMETZ(value) precision reduced to maximum allowed, value

ERROR 5038: Timezone "string" not recognized

ERROR 5039: Timezone "string" uses leap seconds

ERROR 5041: Timezone abbreviation "string" is multiply defined

ERROR 5042: Timezone abbreviation "string" is too long (maximum value characters) in timezone
    file "string", line value

ERROR 5045: Timezone file recursion limit exceeded in file "string"

ERROR 5046: Timezone offset value is not a multiple of 900 sec (15 min) in timezone file "stri
    ng", line value

ERROR 5047: Timezone offset value is out of range in timezone file "string", line value

ERROR 5048: Timezone value "string" is more than value hours

ERROR 5067: Total data collector memory retention of valueKB is too large given system memory
    size

ERROR 5106: TuningRecommendations data collection is disabled

ERROR 5118: UDL specified no execution nodes; at least one execution node must be specified

ERROR 5136: Unable to log this tuning analysis event

ERROR 5202: Unknown configuration parameter

ERROR 5209: Unknown node: string

ERROR 5211: Unknown or unsupported object: string

ERROR 5213: Unknown session ID

ERROR 5215: Unknown value string=string

ERROR 5220: Unrecognized "datestyle" keyword: "string"

ERROR 5229: Unrecognized format 'string'

ERROR 5248: Unrecognized privilege type: "string"

ERROR 5258: Unrecognized timezone name: "string"

ERROR 5271: Unsupported format code: value

ERROR 5316: Usage cannot be an empty string
```

ERROR 5317: Usage of [*string*] cannot be changed from *string* to *string*

ERROR 5319: Usage of [*string*] cannot be changed to *string*. There will be no storage locations for data files

ERROR 5320: Usage of [*string*] cannot be changed to *string*. There will be no storage locations for temporary files

ERROR 5322: Usage:

ERROR 5393: User pool priority must be between -100 and 100 inclusive

ERROR 5437: Vertica should not be run with less than 1GB of RAM

ERROR 5520: *string* compresses network traffic. *string* does NOT compress network traffic. Pleas e change the configuration to be consistent

ERROR 5521: *string* does NOT compresses network traffic. *string* compresses network traffic. Ple ase change the configuration to be consistent

ERROR 5538: Cannot COPY user-defined types directly. Please compute them using copy expressio ns

ERROR 5542: Cannot INSERT or COPY user-defined types directly. Please compute them using appr opriate user-defined functions

ERROR 5545: Cluster layout must include all non-ephemeral nodes and should also not include an y ephemeral nodes

ERROR 5549: Conversion from *string* to DataType *string* failed. Invalid value

ERROR 5571: Empty storage tier label is not allowed

ERROR 5576: Every non-ephemeral node should only be listed once

ERROR 5598: Invalid or unavailable type 'LONG VARBINARY'

ERROR 5599: Invalid or unavailable type 'LONG VARCHAR'

ERROR 5605: Invalid projection createtype '*string*'

ERROR 5613: Length for type *string* must be between 1 and *value*

ERROR 5631: Object *string* does not exist or is not of supported type

ERROR 5632: Object *string* is not a table

ERROR 5634: Path [*string*] is a directory

ERROR 5644: Projection basename "*string*" is not a prefix of projection name "*string*"

ERROR 5645: Projection basename cannot be empty

ERROR 5646: Projection createtype cannot be empty

ERROR 5647: Provided Node "*string*" does not exist

ERROR 5648: Provided Node "*string*" is ephemeral

ERROR 5668: Target table name can not be empty

ERROR 5685: User Defined Filter expected but found *string*

ERROR 5686: User Defined Parser expected but found *string*

ERROR 5687: User Defined Source expected but found *string*

ERROR 5693: Using 1 year for QUEUETIMEOUT

ERROR 5703: Couldn't find the specified task, or the Resource Manager has not recieved the req uest

ERROR 5728: Specified too many widths (*value*) for the given number of columns (*value*)

ERROR 5740: '*string*' is not a valid value for database option *string*

ERROR 5746: analyze_statistics: invalid number of buckets *value*. A number > 0 is required

ERROR 5750: Attempt to configure CPU affinity mode conflicts with configuration of resource pool '*string*'

ERROR 5751: Attempt to configure CPU affinity set to '*string*' conflicts with configuration of resource pool '*string*'

ERROR 5752: Attempt to configure CPU affinity set to '*string*' in exclusive mode would not leave any CPUs available for system queries

ERROR 5753: Attempt to configure CPU affinity to exclusive mode would not leave any CPUs available for system queries

ERROR 5761: Can only specify shared storage for all nodes

ERROR 5762: Can only specify user defined file system for DATA and/or TEMP storage locations

ERROR 5767: Cannot do LOCAL and REJECTED DATA AS TABLE in the same query; rejected records can only be saved to one location

ERROR 5768: Cannot do RETURNREJECTED and REJECTED DATA AS TABLE in the same query; rejected records can only be saved to one location

ERROR 5774: Cannot resolve node address [*string*]

ERROR 5775: Cannot resolve node control address [*string*]

ERROR 5778: Cannot specify both a rejected file and a rejected table in the same statement

ERROR 5779: Cannot specify both an exceptions file and a rejected table in the same statement

ERROR 5780: Cannot specify shared storage for built-in linux file system

ERROR 5793: Control set size out of bounds -1 <= *value* <= 128

ERROR 5804: CPU #*value* is not available to this server, because of server-level processor pinning

ERROR 5885: Failed to load catalog file

ERROR 5918: Improperly formatted broadcast address [*string*]

ERROR 5925: Interface IPv4 address "*string*" is invalid

ERROR 5933: Invalid state for UDFilter: REJECT

ERROR 5934: Invalid state for UDSource: INPUT_NEEDED

ERROR 5935: Invalid state for UDSource: REJECT

ERROR 5954: memoryCap of *string* [*value* KB] would exceed [*value* KB]

ERROR 5962: Must request a positive key count to materialize: *value*

ERROR 5963: Must specify shared storage for built-in hadoop file system

ERROR 5976: Object already exists: *string*. Can't create a rejections table with the same name

ERROR 6006: Request for *value* percent of *value* CPUs rounds to zero CPUs

ERROR 6007: Request for reservation of CPU #*value* conflicts with another pool's reservation

ERROR 6010: Resource pool '*string*' not found

ERROR 6031: STRENGTH value must be in [0.0,1.0]

ERROR 6032: Subnet IPv4 address "*string*" is invalid

ERROR 6044: Table already exists: *string*. Can't create a rejections table with the same name

ERROR 6045: The CPU affinity mode cannot be SHARED or EXCLUSIVE if the affinity set is empty

ERROR 6073: Unable to allocate *value* CPUs for resource pool in *string* affinity mode

ERROR 6088: Unknown control mode *string*

ERROR 6090: Unknown database option '*string*'

```
ERROR 6097: User-Defined Load function indicated that it consumed value bytes, when only value
    were available
```

```
ERROR 6098: User-Defined Load function indicated that UDChunker returned an illegal state from
    process()
```

> **Note:** The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22025

This topic lists the error associated with the SQLSTATE 22025.

SQLSTATE 22025 Description

ERRCODE_INVALID_ESCAPE_SEQUENCE

Error messages associated with this SQLState

```
ERROR 3656: Invalid escape sequence
```

```
ERROR 3657: Invalid escape string
```

> **Note:** The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22906

This topic lists the error associated with the SQLSTATE 22906.

SQLSTATE 22906 Description

ERRCODE_NONSTANDARD_USE_OF_ESCAPE_CHARACTER

Error messages associated with this SQLState

```
ERROR 4166: Nonstandard use of \' in a string literal at or near "string"
```

```
ERROR 4167: Nonstandard use of \\ in a string literal at or near "string"
```

```
ERROR 4168: Nonstandard use of escape in a string literal at or near "string"
```

> **Note:** The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22V02

This topic lists the error associated with the SQLSTATE 22V02.

SQLSTATE 22V02 Description

ERRCODE_INVALID_TEXT_REPRESENTATION

Error messages associated with this SQLState

```
ERROR 2825: Could not convert "string"string to a boolean
ERROR 2826: Could not convert "string"string to a float8
ERROR 2827: Could not convert "string"string to an int8
ERROR 3677: Invalid input for string: "string"
ERROR 3680: Invalid input syntax for boolean: "string"
ERROR 3681: Invalid input syntax for integer: "string"
ERROR 3682: Invalid input syntax for numeric: "value"
ERROR 3711: Invalid number: "string"
ERROR 3712: Invalid numeric format string
ERROR 3714: Invalid numeric value: "string"
ERROR 3751: Invalid Session ID format
ERROR 3757: Invalid syntax for float: "string"
ERROR 3758: Invalid syntax for numeric: "string"
ERROR 3894: Malformed record literal: "string"
ERROR 4169: Not a number: "string"
ERROR 4198: Number exceeds format: "string"
ERROR 5930: Invalid numeric format string. Expected precision is value and scale is value
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22V03

This topic lists the error associated with the SQLSTATE 22V03.

SQLSTATE 22V03 Description

ERRCODE_INVALID_BINARY_REPRESENTATION

Error messages associated with this SQLState

```
ERROR 2829: Could not convert integer valuestring to a boolean
ERROR 3536: Incorrect binary data format in bind parameter value
ERROR 3623: Invalid binary input syntax: 'value'
ERROR 3624: Invalid bitstring "string"
ERROR 3671: Invalid hex string "string"
```

ERROR 3678: Invalid input syntax for *string*

ERROR 3716: Invalid octal string format "*string*"

ERROR 3717: Invalid octal string format (octal string length

ERROR 5416: Value too long for type *string*(*value*)

ERROR 5936: Invalid string format "*string*"

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22V04

This topic lists the error associated with the SQLSTATE 22V04.

SQLSTATE 22V04 Description

ERRCODE_BAD_COPY_FILE_FORMAT

Error messages associated with this SQLState

ERROR 2006: *string value* records have been rejected

ERROR 2031: *string* Header size (*value*) is corrupted

ERROR 2032: *string* Header size (*value*) is too small

ERROR 2035: *string* Input record *value* has been rejected (*string*)

ERROR 2053: *string* Row size (*value*) is corrupted

ERROR 2054: *string* Unexpected EOF while reading header. Expected *value* but read *value*

ERROR 2738: COPY file signature not recognized

ERROR 2767: COPY: Wrong Header size *value*. Expected *value*

ERROR 3562: Input has extra trailing bytes

ERROR 3640: Invalid COPY file header (unsupported Version Number)

ERROR 4206: Number of fields is *value*, expected *value*

ERROR 4627: Row delimiter not found; corrupt file input (read *value* bytes from input)

ERROR 5495: Wrong size *value* for bool column *value* (*string*)

ERROR 5496: Wrong size *value* for date column *value* (*string*)

ERROR 5497: Wrong size *value* for float column *value* (*string*)

ERROR 5498: Wrong size *value* for integer column *value* (*string*)

ERROR 5499: Wrong size *value* for Interval column *value* (*string*)

ERROR 5500: Wrong size *value* for Numeric column *value* (*string*)

ERROR 5501: Wrong size *value* for Time column *value* (*string*)

ERROR 5502: Wrong size *value* for Timestamp column *value* (*string*)

ERROR 5503: Wrong size *value* for TimestampTz column *value* (*string*)

ERROR 5504: Wrong size *value* for TimeTz column *value* (*string*)

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22V0B

This topic lists the error associated with the SQLSTATE 22V0B.

SQLSTATE 22V0B Description

ERRCODE_ESCAPE_CHARACTER_ON_NOESCAPE

Error messages associated with this SQLState

```
ERROR 2746: COPY NO ESCAPE cannot also contain an ESCAPE clause
ERROR 2747: COPY NO ESCAPE for column string cannot also contain an ESCAPE clause
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22V21

This topic lists the error associated with the SQLSTATE 22V21.

SQLSTATE 22V21 Description

ERRCODE_INVALID_EPOCH

Error messages associated with this SQLState

```
ERROR 2144: AHM can't advance past the cluster last backup epoch. (Last Backup Epoch: value)
ERROR 2145: AHM can't advance past the cluster last backup time. (Last Backup time: string)
ERROR 2146: AHM can't advance past the cluster last good epoch (LGE) time (Cluster LGE time: s
    tring)
ERROR 2147: AHM can't advance past the cluster last good epoch (LGE). (Cluster LGE: value)
ERROR 2148: AHM can't advance past the latest epoch time (Latest epoch time: string)
ERROR 2153: AHM must be less than the current epoch (Current Epoch: value)
ERROR 2154: AHM must lag behind the create epoch of unrefreshed projection string (Create epoc
    h: value)
ERROR 2155: AHM must lag behind the create time of unrefreshed projection string (Create time:
    string)
ERROR 2318: Can't run historical queries at epochs prior to the Ancient History Mark
ERROR 3184: Epoch specified is not in historical epoch range
ERROR 3559: Input epoch must be greater than or equal to the earliest epoch (earliest epoch: v
    alue)
```

```
ERROR 3560: Input epoch must be greater than the current AHM (Current AHM: value)
ERROR 3561: Input epoch must be less than or equal to the AHM epoch (AHM epoch: value)
ERROR 3567: Input time can't be rounded down to an epoch higher than the current AHM epoch (Cu
    rrent AHM epoch: value, Current AHM time: string)
ERROR 3568: Input time must be greater than or equal to the earliest epoch time (Earliest epoc
    h time: string)
ERROR 3569: Input time must be greater than the current AHM time (Current AHM time: string)
ERROR 3570: Input time must be less than or equal to the AHM epoch time (AHM epoch time: strin
    g)
ERROR 3654: Invalid epoch
ERROR 3844: Last good epoch not set
ERROR 3926: MergeOut start epoch (=value) greater than end epoch (=value)
ERROR 4940: The current AHM is already value
ERROR 5013: Time specified is not in historical epoch range
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22V23

This topic lists the error associated with the SQLSTATE 22V23.

SQLSTATE 22V23 Description

ERRCODE_RAISE_EXCEPTION

Error messages associated with this SQLState

```
ERROR 5783: Client error: string (in function string() at string:value)
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 22V24

This topic lists the error associated with the SQLSTATE 22V24.

SQLSTATE 22V24 Description

ERRCODE_COPY_PARSE_ERROR

Error messages associated with this SQLState

```
ERROR 2518: Cannot set trailing column to NULL as column value (string) is NOT NULL
ERROR 3401: Field size (value) is corrupted for column value (string)
ERROR 3402: Field size (value) is corrupted for column value (string). It does not fit within
    the row
ERROR 3565: Input numeric value OUT OF RANGE for column value (string)
ERROR 3588: int8 out of range 'string' for column value (string)
ERROR 3617: Invalid string value 'string' for column value (string).string
ERROR 3625: Invalid boolean format 'string' for column value (string)
ERROR 3643: Invalid date format 'string' for column value (string)
ERROR 3644: Invalid date format 'string' for column value (string).string
ERROR 3665: Invalid float format 'string' for column value (string)
ERROR 3666: Invalid float format 'string' for column value (string):No digits were found
ERROR 3683: Invalid integer format 'string' for column value (string)
ERROR 3684: Invalid integer format 'string' for column value (string):No digits were found
ERROR 3685: Invalid interval format 'string' for column value (string).string
ERROR 3713: Invalid numeric format 'string' for column value (string)
ERROR 3763: Invalid time format 'string' for column value (string).string
ERROR 3764: Invalid timestamp format 'string' for column value (string).string
ERROR 3765: Invalid timestamptz format 'string' for column value (string).string
ERROR 3766: Invalid timetz format 'string' for column value (string).string
ERROR 3784: Invalid value 'string' for column value (string).string
ERROR 4196: Null value for NOT NULL column value (string)
ERROR 4209: Numeric out of range 'string' for column value (string)
ERROR 4749: Size value too large for Binary/Varbinary column value (string)
ERROR 4750: Size value too large for Char/Varchar column value (string)
ERROR 4924: The value-byte value is too long for type string(value), column value (string)
ERROR 5017: Time value value microseconds OUT OF RANGE for column value (string)
ERROR 5018: Time value value OUT OF RANGE for column value (string)
ERROR 5040: Timezone value secs OUT OF RANGE for column value (string)
ERROR 5053: Too few columns found
ERROR 5059: Too many columns found
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 23502

This topic lists the error associated with the SQLSTATE 23502.

SQLSTATE 23502 Description

ERRCODE_NOT_NULL_VIOLATION

Error messages associated with this SQLState

```
ERROR 2416: Cannot drop NOT NULL constraint on column "string" when it is referenced in PARTIT
    ION BY expression
```

```
ERROR 2417: Cannot drop NOT NULL constraint on column "string" when it is referenced in primar
    y key constraint
```

```
ERROR 2623: Column "string" definition changed to NOT NULL
```

```
ERROR 4182: NOT NULL constraint on column "string" already exists
```

```
ERROR 4183: NOT NULL constraint on column "string" does not exist
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 23503

This topic lists the error associated with the SQLSTATE 23503.

SQLSTATE 23503 Description

ERRCODE_FOREIGN_KEY_VIOLATION

Error messages associated with this SQLState

```
ERROR 4165: Nonexistent foreign key value detected in FK-PK join [string]; value [string]
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 23505

This topic lists the error associated with the SQLSTATE 23505.

SQLSTATE 23505 Description

ERRCODE_UNIQUE_VIOLATION

Error messages associated with this SQLState

```
ERROR 3147: Duplicate MERGE key detected in join [string]; value [string]
```

```
ERROR 3149: Duplicate primary/unique key detected in join [string]; value [string]
```

```
ERROR 4840: Subquery used as an expression returned more than one row
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that

may help you resolve these errors.

Error Messages Associated with SQLSTATE 25V01

This topic lists the error associated with the SQLSTATE 25V01.

SQLSTATE 25V01 Description

ERRCODE_NO_ACTIVE_SQL_TRANSACTION

Error messages associated with this SQLState

```
ERROR 2342: Cannot advance epoch without a transaction
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 28000

This topic lists the error associated with the SQLSTATE 28000.

SQLSTATE 28000 Description

ERRCODE_INVALID_AUTHORIZATION_SPECIFICATION

Error messages associated with this SQLState

```
ERROR 2701: Conflicting, redundant or unsupported option: string
ERROR 2702: Conflicting, redundant or unsupported option: groupElts
ERROR 2959: Current user cannot be dropped
ERROR 4293: Option "string" not recognized
ERROR 4722: Session user cannot be dropped
ERROR 4846: Superuser cannot be dropped
ERROR 5387: User does not exist
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 2BV01

This topic lists the error associated with the SQLSTATE 2BV01.

SQLSTATE 2BV01 Description

ERRCODE_DEPENDENT_OBJECTS_STILL_EXIST

Error messages associated with this SQLState

```
ERROR 3052: Dependent privileges exist
ERROR 3128: DROP failed due to dependencies
ERROR 3130: DROP PROFILE failed due to dependencies
ERROR 3131: DROP ROLE failed due to dependencies
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 40V01

This topic lists the error associated with the SQLSTATE 40V01.

SQLSTATE 40V01 Description

ERRCODE_T_R_DEADLOCK_DETECTED

Error messages associated with this SQLState

```
ERROR 3010: Deadlock: string - string
ERROR 3011: Deadlock: [Txn value] string - string error string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42501

This topic lists the error associated with the SQLSTATE 42501.

SQLSTATE 42501 Description

ERRCODE_INSUFFICIENT_PRIVILEGE

Error messages associated with this SQLState

```
ERROR 2065: string: Invalid table/projection/column string
ERROR 2198: analyze_statistics: Requires modify permissions for table/projection/column string
ERROR 2347: Cannot alter predefined role "string"
```

ERROR 2348: Cannot alter superuser *string*'s default roles

ERROR 2349: Cannot alter superuser roles

ERROR 2389: Cannot create system built-in tuning rule

ERROR 2419: Cannot drop system built-in tuning rule

ERROR 2460: Cannot move user *string* to general pool, they lack privileges

ERROR 2481: Cannot remove memoryCap

ERROR 2482: Cannot remove runTimeCap

ERROR 2484: Cannot remove tempSpaceCap

ERROR 2515: Cannot set resource pool: user *string* lacks privileges on resource pool *string*

ERROR 2812: Could not add location [*string*]: Permission denied

ERROR 2935: Couldn't nice(*value*) thread: *value*

ERROR 2953: Current password must be supplied to set new password

ERROR 2958: Current user can't change runtime priority of another user's task

ERROR 2960: Current user doesn't have the privilege to change the task runtime priority to be higher than its resource pool

ERROR 3577: Insufficient permissions on projection "*string*"

ERROR 3578: Insufficient permissions on schema "*string*"

ERROR 3579: Insufficient permissions on table "*string*"

ERROR 3580: Insufficient privilege: USAGE on SCHEMA '*string*' not granted for current user

ERROR 3581: Insufficient privileges for projection *string*

ERROR 3582: Insufficient privileges for table *string*

ERROR 3583: Insufficient privileges on *string*

ERROR 3584: Insufficient privileges on *string*, modify privileges (INSERT|UPDATE|DELETE) needed

ERROR 3585: Insufficient privileges to populate statistics for projection *string*

ERROR 3722: Invalid passphrase: *string*

ERROR 3919: memoryCap of *value* KB would exceed user limit of *value* KB

ERROR 3989: Must be owner of *string* *string*

ERROR 3990: Must be owner of *string* [*string*]

ERROR 3991: Must be superuser to alter database

ERROR 3992: Must be superuser to alter profile

ERROR 3993: Must be superuser to alter tuning rule

ERROR 3994: Must be superuser to alter user default roles

ERROR 3995: Must be superuser to audit license size

ERROR 3996: Must be superuser to audit license term

ERROR 3998: Must be superuser to clear Query/EE profiles

ERROR 3999: Must be superuser to crash the database

ERROR 4000: Must be superuser to create interface

ERROR 4001: Must be superuser to create library

ERROR 4002: Must be superuser to create profile

ERROR 4003: Must be superuser to create subnet

ERROR 4004: Must be superuser to create tuning rule

ERROR 4005: Must be superuser to create users

ERROR 4006: Must be superuser to drop an interface

ERROR 4007: Must be superuser to drop library

ERROR 4008: Must be superuser to drop profile

ERROR 4009: Must be superuser to drop resource pool

ERROR 4010: Must be superuser to drop role

ERROR 4011: Must be superuser to drop subnet

ERROR 4012: Must be superuser to drop tuning rule

ERROR 4013: Must be superuser to drop users

ERROR 4014: Must be superuser to modify resource pools

ERROR 4015: Must be superuser to rename interface

ERROR 4016: Must be superuser to rename profile

ERROR 4017: Must be superuser to rename role

ERROR 4018: Must be superuser to rename subnet

ERROR 4019: Must be superuser to run *string*

ERROR 4020: Must be superuser to run analyze_workload*string*()

ERROR 4059: New runTimeCap *value* ms would exceed user limit of *value* ms

ERROR 4061: New tempSpaceCap *value* KB would exceed user limit of *value* KB

ERROR 4178: Not enough privileges for projection *string*

ERROR 4179: Not enough privileges for table *string*

ERROR 4244: Only database superuser can drop procedures

ERROR 4260: Only superuser can check privileges on other users

ERROR 4261: Only superuser can create roles

ERROR 4269: Only the database super user can create procedures

ERROR 4366: Permission denied

ERROR 4367: Permission denied for *string* *string*

ERROR 4368: Permission denied for *string* [*string*]

ERROR 4369: Permission denied to create temporary tables

ERROR 4370: Permission denied: "*string*" is a system catalog

ERROR 4453: Projection *string* already has statistics

ERROR 4546: RecvFiles on *string*: Can't write to file [*string*]

ERROR 4741: setThreadCPUNiceValue: couldn't nice(*value*) thread: *value*

ERROR 4742: setThreadIONiceValue: couldn't ionice(*value*) thread: *value*

ERROR 5149: Unable to set role "*string*"

ERROR 5389: User has insufficient privileges on schema *string*

ERROR 5458: We do not populate statistics for prejoin projections

ERROR 5488: Workspace schema *string* does not exist

ERROR 5517: Your Vertica license is invalid or has expired

ERROR 5618: Must be superuser to alter fault group

ERROR 5619: Must be superuser to create fault group

```
ERROR 5620: Must be superuser to drop fault group
ERROR 5622: Must be superuser to use remote_file_copy
ERROR 5635: Path to file [string] contains a symbolic link
ERROR 5715: Must be superuser to close_all_sockets
ERROR 5716: Must have create permissions in schema string to drop type
ERROR 5818: Deployment script will not be generated since the user does not have appropriate p
    ermissions to write to [string]
ERROR 5820: Design script will not be generated since the user does not have appropriate permi
    ssions to write to [string]
ERROR 5956: Must be superuser to ALTER NODE
ERROR 5957: Must be superuser to create filesystem
ERROR 5958: Must be superuser to create location
ERROR 5959: Must be superuser to CREATE NODEs
ERROR 5960: Must be superuser to realign_control_nodes
ERROR 5961: Must be superuser to supply 'user_name' argument to HAS_ROLE() function
        HINT: Non-superusers run HAS_ROLE('role_name')
ERROR 5975: Not enough privileges for string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42601

This topic lists the error associated with the SQLSTATE 42601.

SQLSTATE 42601 Description

ERRCODE_SYNTAX_ERROR

Error messages associated with this SQLState

```
ERROR 2030: string has been deprecated as string string Vertica option
ERROR 2069: 'string' is not a table name in the current search_path
ERROR 2085: A column cannot occur in an equality predicate and an interpolation predicate
ERROR 2086: A column definition list is only allowed for functions that return "record"
ERROR 2087: A column definition list is required for functions returning "record"
ERROR 2093: A join can have only one set of interpolated predicates
ERROR 2100: A query with Time Series Aggregate Function string must have a timeseries clause
ERROR 2156: All columns are evaluated by expressions. At least one column should be read from
    input
ERROR 2157: All columns in select list must be columns used by projection
ERROR 2164: Alter Column Type driver: Unrecognized command type
ERROR 2180: Analytic function string must have an OVER clause
```

ERROR 2191: ANALYZE CONSTRAINT is not supported

ERROR 2203: Anchor table not found

ERROR 2214: Argument *value* has invalid type *value* in ANALYZE_WORKLOAD

ERROR 2215: Argument *value* in ANALYZE_WORKLOAD must be constant

ERROR 2223: Argument in ANALYZE_CONSTRAINTS must be constant

ERROR 2230: Arguments of row IN must all be row expressions

ERROR 2238: At least two arguments are required

ERROR 2239: At most one path number can be entered

ERROR 2346: Cannot alter a sequence with START

ERROR 2374: Cannot compare row expressions of zero length

ERROR 2381: Cannot create a sequence with RESTART

ERROR 2444: Cannot insert into or update IDENTITY/AUTO_INCREMENT column "*string*"

ERROR 2445: Cannot insert into system column "*string*"

ERROR 2446: Cannot insert multiple commands into a prepared statement

ERROR 2521: Cannot specify anything other than user defined transforms *string* in the *string* li
st

ERROR 2525: Cannot specify more than one user-defined transform function in the SELECT list

ERROR 2526: Cannot specify more than one window clause with a user defined transform

ERROR 2534: Cannot use "PR" with "S"/"PL"/"MI"/"SG"

ERROR 2535: Cannot use "S" with "MI"

ERROR 2536: Cannot use "S" with "PL"

ERROR 2537: Cannot use "S" with "PL"/"MI"/"SG"/"PR"

ERROR 2538: Cannot use "S" with "SG"

ERROR 2539: Cannot use "V" with a decimal point

ERROR 2545: Cannot use aggregate function in VALUES

ERROR 2627: Column "*string*" in ENCODED BY clause is not found in the table

ERROR 2641: Column "*string.string*" must appear in the PARTITION BY list of Timeseries clause o
r be used in a Time Series Aggregate Function

ERROR 2642: Column *string* cannot be evaluated

ERROR 2645: Column *string* has other computed columns in its expression

ERROR 2647: Column *string* in ORDER BY list is not found in TABLE

ERROR 2659: Column alias list for "*string*" has too many entries

ERROR 2669: COLUMN OPTION is not supported

ERROR 2670: Column options are not supported

ERROR 2696: Conflicting INTERVAL subtypes

ERROR 2697: Conflicting NULL/NOT NULL declarations for column "*string*" of table "*string*"

ERROR 2715: Constraint declared INITIALLY DEFERRED must be DEFERRABLE

ERROR 2754: COPY requires a data source; either a FROM clause or a WITH SOURCE for a user-defi
ned source

ERROR 2764: COPY: Expression for column *string* cannot be coerced

ERROR 2946: CREATE TABLE AS specifies too many column names

ERROR 2947: CREATE VIEW specifies more column names than columns

ERROR 2986: Database name is required (too few dotted names): *string*

ERROR 3023: Default values specified for IDENTITY/AUTO_INCREMENT column "*string*" of table "*string*"

ERROR 3125: Drop Column driver: Unrecognized command type

ERROR 3142: Duplicate column "*string*" in create table statement

ERROR 3143: Duplicate column *string* in constraint

ERROR 3146: Duplicate columns in select list of projection not allowed

ERROR 3151: Duplicate tables in projection not allowed

ERROR 3155: Duplicated parameters *string* not allowed

ERROR 3158: Each *string* query must have the same number of columns

ERROR 3164: Empty column name is invalid

ERROR 3165: Empty constraint name is invalid

ERROR 3171: ENCODED BY is not supported in CREATE PROJECTION statement when column renaming list is defined

ERROR 3172: ENCODED BY is not supported in CREATE PROJECTION statement with column definition list

ERROR 3173: ENCODED BY is not supported in CREATE TABLE AS SELECT statement when column list is defined

ERROR 3176: End epoch (*value*) number out of range

ERROR 3177: End epoch (*value*) precedes start epoch (*value*)

ERROR 3183: Epoch number out of range

ERROR 3185: Epoch time out of range

ERROR 3261: Error setting *string* in *string*: Unknown Property

ERROR 3262: Error setting basic directives: '*string*

ERROR 3263: Error setting designer directives: '*string*

ERROR 3264: Error setting optimizer directives: '*string*

ERROR 3344: EXPORT ... SELECT may not specify INTO

ERROR 3348: Expression "(<*string*> - <*string*>) <interval qualifier>" is not supported

ERROR 3349: Expression for column *string* cannot be coerced

ERROR 3458: Function *string* is not allowed in Time Series queries

ERROR 3461: Function *string* requires at least one argument

ERROR 3487: Group by is not allowed in a projection

ERROR 3500: HAVING / GROUP BY not allowed with Time Series query

ERROR 3511: IGNORE NULLS can only be used with FIRST_VALUE or LAST_VALUE

ERROR 3517: Improper %%TYPE reference (too few dotted names): *string*

ERROR 3518: Improper %%TYPE reference (too many dotted names): *string*

ERROR 3519: Improper qualified column name: *string*

ERROR 3520: Improper qualified name (too many dot): *string*

ERROR 3521: Improper qualified name (too many dots): *string*

ERROR 3522: Improper qualified name (too many dotted names): *string*

ERROR 3523: Improper relation name (too many dotted names): *string*

ERROR 3538: Incorrect parameter type provided: *string* is supposed to be of type *string*

ERROR 3548: Indirection is not allowed in a target column

ERROR 3549: Indirection is not allowed in the name of a FILLER column

ERROR 3571: INSERT ... SELECT may not specify INTO

ERROR 3572: INSERT has more expressions than target columns

ERROR 3573: INSERT has more target columns than expressions

ERROR 3599: Interpolated predicates are allowed only in ON CLAUSE of ANSI Join syntax

ERROR 3602: Interpolated predicates should refer to columns from both relations of the join

ERROR 3615: INTO is only allowed on first SELECT of UNION/INTERSECT/EXCEPT

ERROR 3619: Invalid argument type *value* in ANALYZE_CONSTRAINTS

ERROR 3672: Invalid hexadecimal number at or near "*string*"

ERROR 3706: Invalid node name in hint

ERROR 3709: Invalid number at or near "*string*"

ERROR 3738: Invalid projection name in hint: *string*

ERROR 3775: Invalid Unicode escape character '*character*'

ERROR 3776: Invalid Unicode hex number "*string*"

ERROR 3812: Join condition in merge query must include at least one table attribute

ERROR 3841: Label can accept only one argument

ERROR 3865: LIMIT #,# syntax is not supported

ERROR 3944: Misplaced DEFERRABLE clause

ERROR 3945: Misplaced INITIALLY DEFERRED clause

ERROR 3946: Misplaced INITIALLY IMMEDIATE clause

ERROR 3947: Misplaced NOT DEFERRABLE clause

ERROR 3949: Missing argument

ERROR 3958: Missing savepoint name

ERROR 3959: Missing the path number

ERROR 3976: Multiple assignments to same column "*string*"

ERROR 3978: Multiple decimal points

ERROR 3979: Multiple default values specified for column "*string*" of table "*string*"

ERROR 3980: Multiple DEFERRABLE/NOT DEFERRABLE clauses not allowed

ERROR 3981: Multiple FOR UPDATE clauses are not allowed

ERROR 3982: Multiple INITIALLY IMMEDIATE/DEFERRED clauses not allowed

ERROR 3984: Multiple LIMIT clauses are not allowed

ERROR 3985: Multiple OFFSET clauses are not allowed

ERROR 3986: Multiple ORDER BY clauses are not allowed

ERROR 4023: Must specify memorySize parameter

ERROR 4024: Must specify one new name for each schema

ERROR 4025: Must specify one new name for each table

ERROR 4026: Must specify one new name for each view

ERROR 4062: NEW used in query that is not in a rule

ERROR 4066: No actions specified

ERROR 4070: No columns specified in select list

ERROR 4072: No constraints defined

ERROR 4105: No second argument needed when analyzing all constraints

ERROR 4136: Node "*string*" does not exist

ERROR 4161: Non-integer constant in *string*

ERROR 4164: Nonexistent columns: '*string*'

ERROR 4203: Number of columns defined in CREATE TABLE statement is less than in SELECT query o
utput

ERROR 4204: Number of columns defined in CREATE TABLE statement is more than in SELECT query o
utput

ERROR 4205: Number of columns in the PROJECTION statement must be the same as the number of co
lumns in the SELECT statement

ERROR 4225: OLD used in query that is not in a rule

ERROR 4227: ON COMMIT clause may only be specified for TEMPORARY tables

ERROR 4237: Only a single "S" is allowed

ERROR 4239: Only ASC is allowed in ORDER BY list of auto projection for CREATE TABLE

ERROR 4240: Only columns are allowed in ORDER BY list of auto projection for CREATE TABLE

ERROR 4241: Only columns are allowed in SELECT list of projection

ERROR 4247: Only inner joins are allowed in a projection defining query

ERROR 4253: Only one table allowed

ERROR 4268: Only tables are allowed in FROM clause of projection

ERROR 4291: Operator too long at or near "*string*"

ERROR 4294: Option *string* conflicts with prior options

ERROR 4296: Options not set

ERROR 4297: ORDER BY column in timeseries OVER clause must be Timestamp type

ERROR 4325: Parameters can only contain constants or constant expressions

ERROR 4327: Parsing error "*string*" at or near "*string*"

ERROR 4328: PARTITION AUTO can only be used with single-phase user defined transform functions

ERROR 4348: Path Number must be in [0, *value*]

ERROR 4350: Pattern "0" must come before "PR"

ERROR 4351: Pattern "9" must come before "PR"

ERROR 4383: plannedConcurrency must be greater than 0

ERROR 4487: Projections can only be sorted in ascending order

ERROR 4629: Row expressions being compared must have the same number of entries

ERROR 4669: SELECT * with no tables specified is not valid

ERROR 4670: SELECT DISTINCT ON is not standard SQL, use just SELECT DISTINCT

ERROR 4706: Sequence functions accept constant strings arguments only

ERROR 4707: Sequence Manipulation functions are allowed in OUTER SELECT LIST only and cannot b
e in SELECT LIST of a WITH clause

ERROR 4732: Set Operators are not allowed in a projection

ERROR 4761: Sort key *string* should be in the target list

ERROR 4814: Subqueries in a MERGE statement are not allowed

ERROR 4815: Subqueries in MERGE statement are not allowed

ERROR 4828: Subquery has too few columns

ERROR 4829: Subquery has too many columns

ERROR 4831: Subquery in FROM may not have SELECT INTO

ERROR 4833: Subquery in FROM must have an alias

ERROR 4835: Subquery must return a column

ERROR 4836: Subquery must return only one column

ERROR 4837: Subquery not allowed in a projection

ERROR 4838: Subquery not allowed in SELECT list and/or ORDER BY clause for Time Series queries

ERROR 4855: Syntactic Optimizer requires joins written using ANSI JOIN syntax

ERROR 4856: Syntax error at or near "*string*"

ERROR 4947: The foreign key in this constraint has already been defined as a foreign key for r
elation "*string*"

ERROR 4955: The number of target columns (*value*) does not match the number of columns (*value*)
in the EXPORT statement

ERROR 4956: The number of target columns (*value*) is less than the number of columns (*value*) in
the EXPORT statement

ERROR 5007: Time Series Aggregate Functions cannot be nested

ERROR 5008: Time Series queries cannot refer to column of outer query

ERROR 5009: Time Series queries cannot refer to column of outer query: "*string.string*"

ERROR 5011: Time slice length must be a positive integer constant

ERROR 5012: Time slice length must be an interval constant

ERROR 5161: Unequal number of entries in row expression

ERROR 5162: Unequal number of entries in row expressions

ERROR 5272: Unsupported From clause expression

ERROR 5285: Unsupported SET option

ERROR 5286: Unsupported SET option *string*

ERROR 5287: Unsupported SHOW option *string*

ERROR 5290: Unsupported transaction option *string*

ERROR 5305: Unterminated /* comment at or near "*string*"

ERROR 5306: Unterminated bit string literal at or near "*string*"

ERROR 5307: Unterminated dollar-quoted string at or near "*string*"

ERROR 5308: Unterminated hexadecimal string literal at or near "*string*"

ERROR 5310: Unterminated quoted identifier at or near "*string*"

ERROR 5311: Unterminated quoted string at or near "*string*"

ERROR 5323: Usage: clear_profiling(*string* , *string*)

ERROR 5324: Usage: disable_profiling(*string*)

ERROR 5325: Usage: enable_profiling(*string*)

ERROR 5326: Use "*string*(*)" to call this aggregate function

```
ERROR 5383: User Defined Transform Functions are allowed only in a SELECT list
ERROR 5386: User defined transform will return value columns, whereas value aliases provided
ERROR 5401: User-defined transform function string must have an OVER clause
ERROR 5413: Value must be either "units" or "plain"
ERROR 5415: Value must be either ON or OFF
ERROR 5452: Virtual tables are not allowed in FROM clause of projection
ERROR 5492: Wrong number of parameters for prepared statement "string"
ERROR 5493: Wrong number of parameters on left side of OVERLAPS expression
ERROR 5494: Wrong number of parameters on right side of OVERLAPS expression
ERROR 5505: You can specify a node name only once in a create projection statement, node strin
    g appears more than once
ERROR 5518: Zero-length delimited identifier at or near "string"
ERROR 5524: A projection can have only one basename
ERROR 5525: A projection can have only one createtype
ERROR 5566: Dimension tables may not have data that shorter lived than the fact table
ERROR 5577: Expression for user-defined type column string cannot be coerced
ERROR 5600: Invalid predicate in projection-select. Only PK=FK equijoins are allowed
ERROR 5617: Multiple WITH clauses not allowed
ERROR 5629: Not a Star or Snow-Flake Query
ERROR 5630: Nullable FKs are not allowed in projection definition
ERROR 5651: Recursive With is not supported
ERROR 5664: Subqueries not allowed in projection definition
ERROR 5665: Subquery in MERGE is not supported
ERROR 5670: The number of alias columns must be the same as the number of selected columns
ERROR 5691: User-defined function string is not a supported scalar function
ERROR 5696: WITH query name "string" specified more than once
ERROR 5711: Invalid function arguments
ERROR 5714: Missing the random seed
ERROR 5730: The second argument, sampling method, should be always be 1 -- naive sampling(bias
    ed)
ERROR 5733: The third argument must be large than 0
ERROR 5734: Three arguments at most: sampling seed, sampling method (optional, default 1), sam
    pling size (optional,default 10)
ERROR 5916: If specified, maximum error percentage must be a numeric constant
ERROR 5926: Internal error parsing function string
ERROR 5929: Invalid maximum error percentage specified
ERROR 6034: Syntax Error: 'string' is a built in type
ERROR 6048: The minimum value that may be specified for maximum error percentage is 0.0779
ERROR 6061: Too many arguments to string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that

may help you resolve these errors.

Error Messages Associated with SQLSTATE 42602

This topic lists the error associated with the SQLSTATE 42602.

SQLSTATE 42602 Description

ERRCODE_INVALID_NAME

Error messages associated with this SQLState

ERROR 2383: Cannot create projections due to naming conflicts with existing projections

ERROR 2398: Cannot determine the best encoding options for some columns in table *string.string* due to insufficient data

ERROR 3059: DEPRECATED syntax. Segment expression "*string*" is a projection column name, segmenting on attribute "*string*"*stringstringstring* instead

ERROR 3378: Failed to generate a unique relation or sequence name

ERROR 3674: Invalid identifier name (*value* octets) "*string*"

ERROR 3703: Invalid name syntax

ERROR 3747: Invalid savepoint identifier *string*

ERROR 4159: Non-ASCII characters in names are prohibited

ERROR 4267: Only table column names & filler column names can appear in the list

ERROR 4451: Projection "*string*" does not exist

ERROR 4506: Query weight must be positive

ERROR 5360: User "*string*" does not exist

ERROR 5403: User/role "*string*" already exists

ERROR 5569: Either column "*string*" does not exist or table alias "*string*" is not allowed in "WHEN MATCHED THEN UPDATE SET"

ERROR 5769: Cannot drop the main vertica license

ERROR 5968: No such license *string* to drop

ERROR 5970: Node *string* is not a control node

ERROR 6089: Unknown control node *string*

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42611

This topic lists the error associated with the SQLSTATE 42611.

SQLSTATE 42611 Description

ERRCODE_INVALID_COLUMN_DEFINITION

Error messages associated with this SQLState

```
ERROR 2506: Cannot set default for column "*string*" since it is referenced in default expressio
    n of column "*string*"
```

```
ERROR 3017: Default expression for column "*string*" may not refer to itself
```

```
ERROR 6099: Using LONG column '*string*' in a constraint
```

```
ERROR 6100: Using PARTITION expression that returns a *string* value
```

```
ERROR 6101: Using PARTITION expression that returns a LONG value
```

```
ERROR 6102: Using PARTITION expression that returns a LONG value: '*string*'
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42622

This topic lists the error associated with the SQLSTATE 42622.

SQLSTATE 42622 Description

ERRCODE_NAME_TOO_LONG

Error messages associated with this SQLState

```
ERROR 2462: Cannot open FileColumn because path is too long *string*
```

```
ERROR 3507: Identifier "*string*" is *value* octets long. Maximum limit is *value* octets
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42701

This topic lists the error associated with the SQLSTATE 42701.

SQLSTATE 42701 Description

ERRCODE_DUPLICATE_COLUMN

Error messages associated with this SQLState

```
ERROR 2629: Column "*string*" is already of type "*string*"
ERROR 2638: Column "string" specified more than once

ERROR 2654: Column string specified more than once

ERROR 2655: Column string specified more than once in options list

ERROR 2662: Column name "string" already exists

ERROR 2663: Column name "string" appears more than once in USING clause

ERROR 2664: Column name "string" does not exist

ERROR 3144: Duplicate column string in ORDER BY list

ERROR 3145: Duplicate column name

ERROR 3150: Duplicate projection column name (projection: string)

ERROR 3154: Duplicated parameter "string" in parameter list

ERROR 5450: View definition can not contain duplicate column names "string"

ERROR 5878: Failed to create table string: duplicate column name string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42702

This topic lists the error associated with the SQLSTATE 42702.

SQLSTATE 42702 Description

ERRCODE_AMBIGUOUS_COLUMN

Error messages associated with this SQLState

```
ERROR 2604: Clause string "string" is ambiguous

ERROR 2671: Column reference "string" is ambiguous

ERROR 2681: Common column name "string" appears more than once in left table

ERROR 2682: Common column name "string" appears more than once in right table

ERROR 5904: Flex table "string" has no internal "string" column
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42703

This topic lists the error associated with the SQLSTATE 42703.

SQLSTATE 42703 Description

ERRCODE_UNDEFINED_COLUMN

Error messages associated with this SQLState

```
ERROR 2359: Cannot assign to field "string" of column "string" because there is no such column
    in data type string
```

```
ERROR 2625: Column "string" does not exist;
            Vertica does not support 'SELECT <table_name> FROM <table_name>'
```

```
ERROR 2633: Column "string" named as primary key does not exist
```

```
ERROR 2634: Column "string" not found in data type string
```

```
ERROR 2635: Column "string" of relation "string" does not exist
```

```
ERROR 2636: Column "string" specified in USING clause does not exist in left table
```

```
ERROR 2637: Column "string" specified in USING clause does not exist in right table
```

```
ERROR 2639: Column "string"."string" does not exist as a projection column
```

```
ERROR 2643: Column string does not exist
```

```
ERROR 2644: Column string does not exist in table

```

```
ERROR 2651: Column string must be loaded or computed
```

```
ERROR 2656: Column string.string does not exist
```

```
ERROR 2870: Could not identify column "string" in record data type
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42704

This topic lists the error associated with the SQLSTATE 42704.

SQLSTATE 42704 Description

ERRCODE_UNDEFINED_OBJECT

Error messages associated with this SQLState

```
ERROR 2067: 'string' is not a known granularity for audits.
        string
```

```
ERROR 2068: 'string' is not a known TM task.
        string
```

```
ERROR 2070: 'string' is not a valid granularity for string.
        string
```

```
ERROR 2073: 'string' is not supported by index tool
```

```
ERROR 2274: Bootstrap error (most likely in Bootstrap.cpp): Unregistered name string
```

```
ERROR 2275: Bootstrap error (most likely in Bootstrap.cpp): Unregistered oid value
```

```
ERROR 2710: Constraint "string" does not exist
```

```
ERROR 2711: Constraint "string" does not exist on table "string"
```

ERROR 3001: DDL statement interfered with *string*.nextval

ERROR 3256: Error reported by client: *string*

ERROR 3442: Found eligible *value* processes to invite, but no matching nodes in catalog

ERROR 3637: Invalid Component Name '*string*'

ERROR 3655: Invalid epoch range

ERROR 3698: Invalid mergeout task identifier (Possible values are: [0, *value*])

ERROR 3715: Invalid object name

ERROR 3748: Invalid scope in ANALYZE_WORKLOAD*string*: schema or table *string* was altered

ERROR 3749: Invalid scope in ANALYZE_WORKLOAD: schema or table *string* does not exist

ERROR 3756: Invalid Sub-Component Name '*string*'

ERROR 3769: Invalid TM operation

ERROR 3779: Invalid user ID: *value*

ERROR 3842: Language does not exist: *string*

ERROR 3855: Library "*string*" does not exist

ERROR 3862: Library with name '*string*' does not exist

ERROR 4046: Network Interface "*string*" does not exist

ERROR 4047: Network Interface "*string*" is setup on another node

ERROR 4101: No role "*string*" exists

ERROR 4109: No storages in the specified epoch range

ERROR 4110: No such node *string*

ERROR 4111: No such object

ERROR 4112: No such projection

ERROR 4113: No such projection '*string*'

ERROR 4123: No user or role "*string*" exists

ERROR 4129: No value found for parameter "*string*"

ERROR 4130: No value found for parameter *value*

ERROR 4137: Node *string* does not exist

ERROR 4216: Object '*string*' is not a projection

ERROR 4217: Object '*string*' is not a table

ERROR 4218: Object '*string*' is not a table or projection

ERROR 4223: OID *value* is not a sequence

ERROR 4224: OID *value* is not a Table or a View

ERROR 4446: Profile "*string*" does not exist

ERROR 4447: Profile '*string*' does not exist

ERROR 4594: Resource pool "*string*" does not exist

ERROR 4596: Resource pool '*string*' does not exist

ERROR 4614: Role "*string*" does not exist

ERROR 4616: Role "*string*" not found

ERROR 4650: Schema "*string*" does not exist

ERROR 4656: Schema, table, or projection "*string*" does not exist.
        string

```
ERROR 4697: Sequence "string" does not exist

ERROR 4713: Sequence with name 'string' does not exist

ERROR 4806: Subnet "string" does not exist

ERROR 4876: Table "string" does not exist

ERROR 4926: The string "string" does not exist

ERROR 4928: The string ["string"] does not exist

ERROR 5105: Tuning rule "string" does not exist

ERROR 5108: Type "string" does not exist

ERROR 5109: Type "string" is only a shell

ERROR 5112: Type string is only a shell

ERROR 5115: Type with OID value does not exist

ERROR 5227: Unrecognized drop object type: value

ERROR 5362: User or Role "string" not found

ERROR 5365: User available location ["string"] does not exist on node ["string"]

ERROR 5446: View "string" does not exist

ERROR 5459: Window "string" does not exist

ERROR 5532: Can not find any eligible locations in tier string

ERROR 5585: Fault Group "string" does not exist

ERROR 5614: Library string does not exist

ERROR 5688: User Defined Type "string" does not exist

ERROR 5797: Could not find the JVM resource pool

ERROR 5913: HCatalog database string does not exist

ERROR 5931: Invalid Policy Name 'string'

ERROR 5965: New node cannot be placed in a non-existent Fault Group "string"

ERROR 5969: No table or projection named string exists

ERROR 5974: Node doesn't exist

ERROR 5977: Object does not exist

ERROR 6071: Type value with odbc_subtype value is not supported
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42710

This topic lists the error associated with the SQLSTATE 42710.

SQLSTATE 42710 Description

ERRCODE_DUPLICATE_OBJECT

Error messages associated with this SQLState

ERROR 2101: A sequence named "*string*" already exists

ERROR 2105: A table named "*string*" already exists

ERROR 2107: A view named "*string*" already exists

ERROR 2273: Bootstrap error (most likely in Bootstrap.cpp): Oid *value* is already registered

ERROR 2276: Bootstrap error (most likely in Bootstrap.cpp):Name *string* is already registered

ERROR 2713: Constraint *string* already exists

ERROR 3153: Duplicated local temp table found in design queries: *string*

ERROR 3327: Existing object "*string*" is not a view

ERROR 3881: Location [*string*] already exists for node *string*

ERROR 4043: Network Interface "*string*" already exists

ERROR 4135: Node "*string*" already exists

ERROR 4213: Object "*string*" already exists

ERROR 4445: Profile "*string*" already exists

ERROR 4482: Projection with base name "*string*" already exists

ERROR 4564: Relation "*string*" already exists

ERROR 4565: Relation "*string*" already exists in schema "*string*"

ERROR 4593: Resource pool "*string*" already exists

ERROR 4621: Role\User "*string*" already exists

ERROR 4804: Subnet "*string*" already exists

ERROR 4805: Subnet "*string*" already exists for [*string*]

ERROR 5582: Fault Group "*string*" already exists

ERROR 5584: Fault Group "*string*" cannot depend on itself directly or indirectly

ERROR 5615: Location [*string*] conflicts with existing location [*string*] on node *string*

ERROR 5623: Network Interface "*string*" already exists for [*string*]

ERROR 5736: Unable to guarantee the same base name for all replicated buddy projections

ERROR 5737: Unable to guarantee the same base name for all segmented buddy projections

ERROR 6009: Resource pool *string* already exists

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42712

This topic lists the error associated with the SQLSTATE 42712.

SQLSTATE 42712 Description

ERRCODE_DUPLICATE_ALIAS

Error messages associated with this SQLState

```
ERROR 4901: Table name "string" specified more than once
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42723

This topic lists the error associated with the SQLSTATE 42723.

SQLSTATE 42723 Description

ERRCODE_DUPLICATE_FUNCTION

Error messages associated with this SQLState

```
ERROR 2278: Built-in function with the same name already exists: string

ERROR 3472: Function with same name and number of parameters already exists: string

ERROR 4220: Object with same name and number of parameters already exists: string

ERROR 4428: Procedure/Function with same name and number of parameters already exists in schem
    a string

ERROR 4429: Procedure/Function with same name and number of parameters already exists: string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42725

This topic lists the error associated with the SQLSTATE 42725.

SQLSTATE 42725 Description

ERRCODE_AMBIGUOUS_FUNCTION

Error messages associated with this SQLState

```
ERROR 3459: Function string is not unique

ERROR 4289: Operator is not unique: string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42803

This topic lists the error associated with the SQLSTATE 42803.

SQLSTATE 42803 Description

ERRCODE_GROUPING_ERROR

Error messages associated with this SQLState

```
ERROR 2134: Aggregate function calls in subqueries cannot refer to columns in parent (outer) q
    uery
```

```
ERROR 2135: Aggregate function calls may not be nested
```

```
ERROR 2140: Aggregates not allowed in GROUP BY clause
```

```
ERROR 2141: Aggregates not allowed in JOIN conditions
```

```
ERROR 2142: Aggregates not allowed in WHERE clause
```

```
ERROR 2219: Argument string must not contain aggregates
```

```
ERROR 2543: Cannot use aggregate function in EXECUTE parameter
```

```
ERROR 2544: Cannot use aggregate function in function expression in FROM
```

```
ERROR 2640: Column "string.string" must appear in the GROUP BY clause or be used in an aggrega
    te function
```

```
ERROR 4634: Rule WHERE condition may not contain aggregate functions
```

```
ERROR 4667: SEGMENTED BY expression may not contain aggregate functions
```

```
ERROR 4841: Subquery uses ungrouped column "string.string" from outer query
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42804

This topic lists the error associated with the SQLSTATE 42804.

SQLSTATE 42804 Description

ERRCODE_DATATYPE_MISMATCH

Error messages associated with this SQLState

```
ERROR 2217: Argument string must be type float, not type string
```

```
ERROR 2218: Argument string must be type integer, not type string
```

```
ERROR 2222: Argument string must not return a set
```

```
ERROR 2224: Argument of string must be type boolean, not type string
```

```
ERROR 2225: Argument of string must not return a set
```

ERROR 2231: Array assignment requires type *string* but expression is of type *string*

ERROR 2232: Array assignment to "*string*" requires type *string* but expression is of type *string*

ERROR 2234: Array subscript must have type integer

ERROR 2358: Cannot assign to field "*string*" of column "*string*" because its type *string* is not a composite type

ERROR 2527: Cannot subscript type *string* because it is not an array

ERROR 2630: Column "*string*" is of type *string* but default expression is of type *string*

ERROR 2631: Column "*string*" is of type *string* but expression is of type *string*

ERROR 2846: Could not determine actual result type for function "*string*" declared to return ty pe *string*

ERROR 2850: Could not determine row description for function returning record

ERROR 3429: For '*string*', types *string* and *string* are inconsistent

ERROR 3447: Function "*string*" in FROM has unsupported return type *string*

ERROR 3545: Index expression may not return a set

ERROR 3801: IS DISTINCT FROM requires = operator to yield boolean

ERROR 3943: Mismatched types in VALUES LESS THAN expressions

ERROR 4069: No column alias was provided

ERROR 4199: Number of aliases does not match number of columns

ERROR 4284: Operator *string* must not return a set

ERROR 4285: Operator *string* must return type boolean, not type *string*

ERROR 4317: Parameter $*value* of type *string* cannot be coerced to the expected type *string*

ERROR 4625: Row comparison operator must not return a set

ERROR 4626: Row comparison operator must yield type boolean, not type *string*

ERROR 4803: Subfield "*string*" is of type *string* but expression is of type *string*

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42809

This topic lists the error associated with the SQLSTATE 42809.

SQLSTATE 42809 Description

ERRCODE_WRONG_OBJECT_TYPE

Error messages associated with this SQLState

ERROR 2037: *string* is not a supported analytic function

ERROR 2062: *string*(*) specified, but *string* is not an aggregate function

ERROR 2131: Aggregate function calls cannot contain analytic function calls

ERROR 2132: Aggregate function calls cannot contain sequence function calls

ERROR 2668: Column notation .*string* applied to type *string*, which is not a composite type

```
ERROR 2755: COPY requires relation string to be a Table, not a string
ERROR 2810: Could not add location [string]: Directory not empty
ERROR 2811: Could not add location [string]: Not a directory
ERROR 3114: DISTINCT specified, but string is not an aggregate function
ERROR 3421: First argument to modularhash_wrapper must be an integer constant
ERROR 3422: First argument to modularhash_wrapper must be of type integer, not string
ERROR 3463: Function string(string) is not an aggregate
ERROR 3552: Inherited relation "string" is not a table
ERROR 3669: Invalid function given
ERROR 3965: modularhash_wrapper must have two arguments: an integer constant and a call to mod
    ularhash_internal
ERROR 3966: modularhash_wrapper second argument is not modularhash_internal or a constant
ERROR 4215: Object "string" is not a projection
ERROR 4270: Op ANY/ALL (array) requires array on right side
ERROR 4271: Op ANY/ALL (array) requires operator not to return a set
ERROR 4272: Op ANY/ALL (array) requires operator to yield boolean
ERROR 4542: Record type has not been registered
ERROR 4657: Second argument to string must be a non-negative integer constant
ERROR 4931: The argument to string cannot be null
ERROR 4932: The argument to string must be a constant
ERROR 4987: Third argument to string must be a constant
ERROR 5111: Type string is not composite
ERROR 6036: Table "string" is not a flex table
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42830

This topic lists the error associated with the SQLSTATE 42830.

SQLSTATE 42830 Description

ERRCODE_INVALID_FOREIGN_KEY

Error messages associated with this SQLState

```
ERROR 3438: Foreign keys not specified
ERROR 3531: Incompatible data types between primary and foreign key columns: fk: string, pk: s
    tring
ERROR 4207: Number of primary and foreign keys must be the same
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42846

This topic lists the error associated with the SQLSTATE 42846.

SQLSTATE 42846 Description

ERRCODE_CANNOT_COERCE

Error messages associated with this SQLState

`ERROR 2015:` *`string`* `could not convert type` *`string`* `to` *`string`*

`ERROR 2366: Cannot cast type` *`string`* `to` *`string`*

`ERROR 2632: Column "`*`string`*`" is of type` *`string`* `but the default expression is of type` *`string`*

`ERROR 4986: Third argument of` *`string`* `could not be converted from type` *`string`* `to type` *`string`*

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42883

This topic lists the error associated with the SQLSTATE 42883.

SQLSTATE 42883 Description

ERRCODE_UNDEFINED_FUNCTION

Error messages associated with this SQLState

`ERROR 2126: Aggregate` *`string`*`(`*`string`*`) does not exist`

`ERROR 2127: Aggregate` *`string`*`(*) does not exist`

`ERROR 3456: Function` *`string`* `does not exist`

`ERROR 3457: Function` *`string`* `does not exist, or permission is denied for` *`string`*

`ERROR 3462: Function` *`string`* `with the specified arguments does not exist`

`ERROR 3930: Meta-function` *`string`* `cannot be used in COPY`

`ERROR 3931: Meta-function` *`string`* `cannot be used in INSERT`

`ERROR 3932: Meta-function` *`string`* `cannot be used in UPDATE`

`ERROR 3933: Meta-function` *`string`* `cannot be used with FROM`

`ERROR 3934: Meta-function ("`*`string`*`") can be used only in the Select clause`

`ERROR 3935: Meta-function ("`*`string`*`") cannot be used with non-Select clauses`

`ERROR 3936: Meta-functions cannot be used in default expressions`

```
ERROR 4067: No binary input function available for type string
ERROR 4068: No binary output function available for type string
ERROR 4083: No input function available for type string
ERROR 4091: No output function available for type string
ERROR 4286: Operator does not exist: string
ERROR 4290: Operator requires run-time type coercion: string
ERROR 5394: User procedure call (value) is not supported with FROM
ERROR 5455: VOLATILE functions cannot be used in a default expression when adding a column
ERROR 5910: Function string with the specified type and arguments does not exist
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42939

This topic lists the error associated with the SQLSTATE 42939.

SQLSTATE 42939 Description

ERRCODE_RESERVED_NAME

Error messages associated with this SQLState

```
ERROR 2297: Can not drop default profile
ERROR 2299: Can not rename default profile
ERROR 2418: Cannot drop role "string"
ERROR 2488: Cannot rename role string
ERROR 2489: Cannot rename system column epoch
ERROR 2665: Column name "string" is reserved
ERROR 2666: Column name string is reserved
ERROR 3778: Invalid use of reserved the column name "string"
ERROR 4030: Names starting with "v_" are reserved names
ERROR 4953: The name "string" is a reserved name
ERROR 4962: The prefix "sys_" is reserved for system tuning rule
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42P20

This topic lists the error associated with the SQLSTATE 42P20.

SQLSTATE 42P20 Description

ERRCODE_WINDOWING_ERROR

Error messages associated with this SQLState

ERROR 2011: *string* cannot use the WITHIN GROUP clause

ERROR 2041: *string* may only have one sort expression in the WITHIN GROUP clause

ERROR 2043: *string* must contain an ORDER BY clause within its analytic clause

ERROR 2044: *string* must NOT contain an ORDER BY clause or WINDOWING clause within its analytic clause

ERROR 2045: *string* must NOT contain WINDOWING clause within its analytic clause

ERROR 2047: *string* only supports the Integer, Float, Interval and Numeric data types

ERROR 2182: Analytic functions are allowed only in a SELECT list and/or ORDER BY clause

ERROR 2185: Analytic functions are not supported in the PARTITION BY of an OVER clause

ERROR 2187: Analytic functions cannot be nested

ERROR 2188: Analytic functions must have a FROM clause

ERROR 2189: Analytic functions not allowed in *string*

ERROR 2305: Can't cast the window bound into Int

ERROR 2306: Can't cast the window bound into the same data type of the ORDER BY column

ERROR 2465: Cannot override ORDER BY clause of window "*string*"

ERROR 2466: Cannot override PARTITION BY clause of window "*string*"

ERROR 2524: Cannot specify frame clause of window "*string*"

ERROR 3435: For range moving window, OrderBy expression must be one of Int, Float, Time, Times tamp, Interval, Date or Numeric

ERROR 3446: Frame clause not allowed without windowing order by

ERROR 3839: Keyword "ALL" is invalid in analytic functions

ERROR 4362: PERCENTILE_CONT/PERCENTILE_DISC must have the WITHIN GROUP clause

ERROR 4363: PERCENTILE_CONT/PERCENTILE_DISC must NOT contain an ORDER BY clause or WINDOWING c lause within its analytic clause

ERROR 4811: Subqueries are not supported in the PARTITION BY of a timeseries OVER clause

ERROR 5006: Time Series Aggregate Functions are not supported in the PARTITION BY of a timeser ies OVER clause

ERROR 5010: Time Series timestamp alias/Time Series Aggregate Functions not allowed in *string*

ERROR 5460: Window "*string*" is already defined

ERROR 5461: Window frame cannot end with PRECEDING if start is CURRENT ROW

ERROR 5462: Window frame cannot end with PRECEDING or CURRENT ROW if start is FOLLOWING

ERROR 5463: Window frame cannot end with UNBOUNDED PRECEDING

ERROR 5464: Window frame cannot start with UNBOUNDED FOLLOWING

ERROR 5466: Window frame logical offset must be a non-negative number to be consistent with th e sort column type

```
ERROR 5467: Window frame logical offset must be an Interval (Day to Second or Year to Month) t
    o be consistent with the sort column type
ERROR 5468: Window frame logical offset must be an Interval (Day to Second) to be consistent w
    ith the sort column type
ERROR 5469: Window frame logical offset must be an interval to be consistent with the sort col
    umn type
ERROR 5470: Window frame logical offset must be Int when the sort column type is Int
ERROR 5471: Window frame logical offset must be the same type as the sort column type (Interva
    l Day to Second)
ERROR 5472: Window frame logical offset must be the same type as the sort column type (Interva
    l Year to Month)
ERROR 5473: Window frame logical or physical offset must be a constant
ERROR 5474: Window frame logical or physical offset must be non-negative number or interval
ERROR 5475: Window frame physical offset must be non-negative number
ERROR 5477: Window ordering clause can only contain a single sort key if RANGE is used
ERROR 5478: Windowing not supported for User Defined Analytic functions
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V01

This topic lists the error associated with the SQLSTATE 42V01.

SQLSTATE 42V01 Description

ERRCODE_UNDEFINED_TABLE

Error messages associated with this SQLState

```
ERROR 2072: 'string' is not a valid table
ERROR 2308: Can't find anchor table
ERROR 2312: Can't find table
ERROR 2313: Can't find table "string"
ERROR 2714: Constraint string does not exist
ERROR 2948: CTAS: table "string" was dropped in another session (DDL interference)
ERROR 3367: Failed to create projection for 'string'
ERROR 3642: Invalid CTAS query: string
ERROR 3760: Invalid table name
ERROR 3761: Invalid table name "string"
ERROR 3762: Invalid table name string
ERROR 3953: Missing FROM-clause entry for table "string"
ERROR 3954: Missing FROM-clause entry in subquery for table "string"
ERROR 4416: Primary table "string" does not exist
ERROR 4566: Relation "string" does not exist
ERROR 4567: Relation "string" in FOR UPDATE clause not found in FROM clause
ERROR 4568: Relation "string.string" does not exist
ERROR 4570: Relation with OID value does not exist
ERROR 4883: Table "string.string" does not exist
ERROR 4898: Table does not exist (oid=value)
ERROR 4911: Table with OID value does not exist
ERROR 4912: Table/View with name 'string' does not exist
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V02

This topic lists the error associated with the SQLSTATE 42V02.

SQLSTATE 42V02 Description

ERRCODE_UNDEFINED_PARAMETER

Error messages associated with this SQLState

```
ERROR 3638: Invalid configuration parameter string; aborting configuration change
ERROR 4321: Parameter value is not set
ERROR 4984: There is no parameter $value
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V03

This topic lists the error associated with the SQLSTATE 42V03.

SQLSTATE 42V03 Description

ERRCODE_DUPLICATE_CURSOR

Error messages associated with this SQLState

```
ERROR 2615: Closing existing cursor "string"
ERROR 2968: Cursor "string" already exists
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that

may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V04

This topic lists the error associated with the SQLSTATE 42V04.

SQLSTATE 42V04 Description

ERRCODE_DUPLICATE_DATABASE

Error messages associated with this SQLState

```
ERROR 2706: Connection to database [string] already exists
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V06

This topic lists the error associated with the SQLSTATE 42V06.

SQLSTATE 42V06 Description

ERRCODE_DUPLICATE_SCHEMA

Error messages associated with this SQLState

```
ERROR 4649: Schema "string" already exists
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V07

This topic lists the error associated with the SQLSTATE 42V07.

SQLSTATE 42V07 Description

ERRCODE_DUPLICATE_TABLE

Error messages associated with this SQLState

```
ERROR 4753: Skip lazy projection creation since super projection for table string.string alrea
    dy exists
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V08

This topic lists the error associated with the SQLSTATE 42V08.

SQLSTATE 42V08 Description

ERRCODE_AMBIGUOUS_PARAMETER

Error messages associated with this SQLState

`ERROR 2848: Could not determine data type of parameter $`*`value`*

`ERROR 3534: Inconsistent types deduced for parameter $`*`value`*

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V09

This topic lists the error associated with the SQLSTATE 42V09.

SQLSTATE 42V09 Description

ERRCODE_AMBIGUOUS_ALIAS

Error messages associated with this SQLState

`ERROR 4908: Table reference "`*`string`*`" is ambiguous`

`ERROR 4909: Table reference `*`value`*` is ambiguous`

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V10

This topic lists the error associated with the SQLSTATE 42V10.

SQLSTATE 42V10 Description

ERRCODE_INVALID_COLUMN_REFERENCE

Error messages associated with this SQLState

```
ERROR 2046: string not allowed in string clause
ERROR 2050: string position value is not in select list
ERROR 2221: Argument string must not contain variables
ERROR 3467: Function expression in FROM may not refer to other relations of same query level
ERROR 3820: JOIN/ON clause refers to "string", which is not part of JOIN
ERROR 4832: Subquery in FROM may not refer to other relations of same query level
ERROR 4877: Table "string" has value columns available but value columns specified
ERROR 5057: Too many column aliases specified for function string
ERROR 5194: UNION/INTERSECT/EXCEPT member statement may not refer to other relations of same q
    uery level
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V11

This topic lists the error associated with the SQLSTATE 42V11.

SQLSTATE 42V11 Description

ERRCODE_INVALID_CURSOR_DEFINITION

Error messages associated with this SQLState

```
ERROR 2522: Cannot specify both SCROLL and NO SCROLL
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V13

This topic lists the error associated with the SQLSTATE 42V13.

SQLSTATE 42V13 Description

ERRCODE_INVALID_FUNCTION_DEFINITION

Error messages associated with this SQLState

```
ERROR 2038: string is not a supported Time Series Aggregate Function
ERROR 2139: Aggregates may not return sets
ERROR 2173: An error occurred on node [string] when setting up the function [string]: [string]
ERROR 2177: An error occurred when setting up function "string"
ERROR 2397: Cannot determine result data type
ERROR 2451: Cannot load data from 0 sources; please specify 1 or more (on node [string])
ERROR 2494: Cannot RETURNREJECTED with multiple files or data sources
ERROR 3113: DISTINCT is supported only for single-argument aggregates
ERROR 3476: Functions in language string can be created only in fenced mode
ERROR 3604: Interpolation scheme string for Time Series Aggregate Function string is not suppo
    rted
ERROR 3708: Invalid null argument for TSA function string
ERROR 3843: Language(string) does not match the language associated with the library(string)
ERROR 3854: Length of a string in a return type must be greater than zero
ERROR 3860: Library file is not loaded
ERROR 3861: Library not found: string
ERROR 3929: Meta functions cannot be used in UDx definitions
ERROR 4086: No language specified
ERROR 4095: No procedure source specified
ERROR 4096: No procedure user specified
ERROR 4243: Only COUNT() can have star(*) as its argument
ERROR 4249: Only MIN/MAX are allowed to use DISTINCT
ERROR 4251: Only one expression is allowed
ERROR 4257: Only simple "RETURN expression" is allowed
ERROR 4409: Precision of a numeric in a return type must be greater than zero
ERROR 4608: Return type string is not supported for SQL functions
ERROR 4609: Return type mismatch in a function declared to return string
ERROR 4610: Return type mismatch in function declared to return string
ERROR 4746: Setting up function "string" failed
ERROR 4794: Strictness in the DDL and the function factory class don't match. Function was not
    created
ERROR 4858: Syntax error in syntax definition at offset value
ERROR 4949: The interpolation argument for Time Series Aggregate Function string must be a con
    stant string
ERROR 5457: Volatility in the DDL and the function factory class don't match. Function was not
    created
ERROR 5476: Window functions cannot return sets
ERROR 5777: Cannot set up function [string] on node: string
ERROR 6072: UDFileSystem only supports C++ unfenced mode
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V15

This topic lists the error associated with the SQLSTATE 42V15.

SQLSTATE 42V15 Description

ERRCODE_INVALID_SCHEMA_DEFINITION

Error messages associated with this SQLState

```
ERROR 2470: Cannot plan query because no super projections are safe, some node(s) are down
ERROR 2945: CREATE specifies a schema (string) different from the one being created (string)
ERROR 3365: Failed to create default projections for table "string"."string": string
ERROR 3586: Insufficient projections to answer query
ERROR 4097: No projections eligible to answer query
ERROR 4878: Table "string" has an out-of-date super projection "string"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V16

This topic lists the error associated with the SQLSTATE 42V16.

SQLSTATE 42V16 Description

ERRCODE_INVALID_TABLE_DEFINITION

Error messages associated with this SQLState

```
ERROR 2104: A table cannot have only IDENTITY/AUTO-INCREMENT columns
ERROR 2420: Cannot drop the constraint. (Table "string" has a foreign key constraint referenci
    ng the specified primary key constraint)
ERROR 2421: Cannot drop the constraint. (There is at least one prejoin projection dependent on
    the specified foreign key constraint)
ERROR 2588: CHECK constraints not supported
ERROR 2622: Column "string" cannot be declared SETOF
ERROR 2626: Column "string" from table "string" in the SEGMENTED BY expression is required to
    be present in the projection, but is not
ERROR 2712: Constraint "string" for relation "string" already exists
ERROR 3508: IDENTITY/AUTO-INCREMENT columns are not allowed in temporary tables
ERROR 3874: Local temporary table constraint cannot reference a non-local table
ERROR 3901: MATCH types other than SIMPLE (the default) are not supported for foreign key cons
    traints
```

```
ERROR 3987: Multiple primary keys for table "string" are not allowed
```

```
ERROR 4162: Non-local table constraint cannot reference a local temporary table
```

```
ERROR 4229: ON DELETE actions other than NO ACTION are not supported for foreign key constrain
    ts
```

```
ERROR 4234: ON UPDATE actions other than NO ACTION are not supported for foreign key constrain
    ts
```

```
ERROR 4413: Primary constraint for relation "string" already exists
```

```
ERROR 4415: Primary keys not specified
```

```
ERROR 4469: Projection anchor table is not partitioned
```

```
ERROR 4550: Referenced primary key constraint does not exist
```

```
ERROR 4881: Table "string" is not partitioned
```

```
ERROR 4899: Table is not partitioned
```

```
ERROR 4900: Table must have at least one column
```

```
ERROR 5269: Unsupported constraint type
```

```
ERROR 5548: Constraint not supported for user defined type column string
```

```
ERROR 5552: Correlation constraint not supported for user defined types
```

```
ERROR 5874: Failed to add table string of hcatalog schema string to catalog: no columns
```

```
ERROR 5876: Failed to alter table string of hcatalog schema string to catalog: no columns
```

```
ERROR 5879: Failed to describe hcatalog table
```

```
ERROR 5948: Local temporary objects may not specify a schema name
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V17

This topic lists the error associated with the SQLSTATE 42V17.

SQLSTATE 42V17 Description

ERRCODE_INVALID_OBJECT_DEFINITION

Error messages associated with this SQLState

```
ERROR 2387: Cannot create projections involving external table string
```

```
ERROR 3075: Design type string is invalid
```

```
ERROR 3078: Optimization objective string is invalid
```

```
ERROR 3199: Error during deployment querying deployment projections table for workspace string
```

```
ERROR 3200: Error during deployment querying design projections table for design string in wor
    kspace string
```

```
ERROR 3201: Error during deployment while querying deployment projections table for workspace
    string
```

```
ERROR 3204: Error during drop design from deployment for workspace string
```

```
ERROR 3206: Error during extend catalog while querying deployments table for workspace string
```

```
ERROR 3207: Error during getDesignTablesFromDeployment in workspace string
```

```
ERROR 3213: Error during remove deployment drops from deployment string for workspace string
```

```
ERROR 3227: Error in querying string.string
```

```
ERROR 3269: Error while checking whether there are only incremental design deployed for deploy
    ment string in workspace string
```

```
ERROR 3271: Error while querying designs table for workspace string
```

```
ERROR 3968: More than one IDENTITY/AUTO_INCREMENT column defined for table "string"
```

```
ERROR 3983: Multiple instances of deployment string in workspace string
```

```
ERROR 4128: No valid projections found
```

```
ERROR 4230: ON DELETE rule may not use NEW
```

```
ERROR 4231: ON INSERT rule may not use OLD
```

```
ERROR 4232: ON SELECT rule may not use NEW
```

```
ERROR 4233: ON SELECT rule may not use OLD
```

```
ERROR 4635: Rule WHERE condition may not contain references to other relations
```

```
ERROR 4636: Rules with WHERE conditions may only have SELECT, INSERT, UPDATE, or DELETE action
    s
```

```
ERROR 4919: Temporary table projections are not allowed for this operation
```

```
ERROR 4982: There is no deployment string in workspace string
```

```
ERROR 4989: This function cannot be called on design string located in design workspace string
```

```
ERROR 4990: This function cannot be called on design string, when its design mode is string
```

```
ERROR 5367: User defined aggregate must return exactly one column.Function string returns valu
    e
```

```
ERROR 5369: User defined analytic must return exactly one column
```

```
ERROR 5384: User defined transform must provide names or aliases for return columns
```

```
ERROR 5385: User defined transform must return at least one column
```

```
ERROR 5527: An error occurred on node string when setting up the type, message:
        string
```

```
ERROR 5721: Purge is not allowed on temporary tables
```

```
ERROR 6095: UseLongStrings has been deprecated
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V18

This topic lists the error associated with the SQLSTATE 42V18.

SQLSTATE 42V18 Description

ERRCODE_INDETERMINATE_DATATYPE

Error messages associated with this SQLState

```
ERROR 2847: Could not determine data type of column $value
ERROR 3609: Interval must be single datetime field
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V21

This topic lists the error associated with the SQLSTATE 42V21.

SQLSTATE 42V21 Description

ERRCODE_UNDEFINED_PROJECTION

Error messages associated with this SQLState

```
ERROR 2311: Can't find projection value
ERROR 2430: Cannot find projection column value
ERROR 3005: DDL statement interfered with refresh operation
ERROR 3736: Invalid projection name
ERROR 3737: Invalid projection name string
ERROR 4452: Projection "string" does not exist or was just dropped
ERROR 4474: Projection does not exist
ERROR 4905: Table or projection "string" does not exist
ERROR 5563: DDL statement interfered with this operation
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V25

This topic lists the error associated with the SQLSTATE 42V25.

SQLSTATE 42V25 Description

ERRCODE_PATTERN_MATCH_ERROR

Error messages associated with this SQLState

```
ERROR 2227: Argument to test_pattern_event_eval must be > 0 and less than the total number of
    events
ERROR 2228: Argument to test_pattern_event_eval must be a constant
ERROR 2553: Cannot use more than one pattern
ERROR 2555: Cannot use pattern test functions with pattern match functions
ERROR 3025: Defining more than 52 events is not supported
ERROR 3288: Event "string" in PATTERN clause is not defined in the DEFINE clause
ERROR 3289: Event ANY_ROW cannot be used under *, +, ?, or | when the select list contains the
    pattern function event_name()
ERROR 3290: Event ANY_ROW is a reserved event and cannot be user defined
ERROR 3294: Event expressions cannot contain analytic functions
ERROR 3295: Event expressions cannot contain correlated expressions
ERROR 3296: Event expressions cannot contain subqueries
ERROR 3297: Event name "string" defined more than once
ERROR 4353: Pattern events must be mutually exclusive
ERROR 4354: Pattern match query cannot contain having clause, group clause, aggregates, or dis
    tinct
ERROR 4355: Pattern match query cannot contain timeseries clause
ERROR 4356: Pattern matching recursion limit reached
ERROR 4358: PatternMatchingMaxPartition must be greater than 0
ERROR 4359: PatternMatchingMaxPartitionMatches must be greater than 0
ERROR 4360: PatternMatchingPerMatchWorkspaceSize must be greater than 0 and less than 1024
ERROR 4494: Queries with user-defined transform functions (string) cannot have a MATCH clause
ERROR 4507: Query with analytic function string cannot have a MATCH clause
ERROR 4509: Query with pattern matching function string must include a MATCH clause
ERROR 4605: RESULTS GROUPED BY MATCH is not supported
ERROR 5283: Unsupported pattern operator
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 42V26

This topic lists the error associated with the SQLSTATE 42V26.

SQLSTATE 42V26 Description

ERRCODE_DUPLICATE_NODE

Error messages associated with this SQLState

```
ERROR 4058: New node matches existing node string
ERROR 4063: New values for node string matches existing node string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 53000

This topic lists the error associated with the SQLSTATE 53000.

SQLSTATE 53000 Description

ERRCODE_INSUFFICIENT_RESOURCES

Error messages associated with this SQLState

```
ERROR 2245: Attempted to create too many ROS containers for projection string
ERROR 2843: Could not create thread for recoverProjectionLocal
ERROR 2844: Could not create thread for SubsessionHandler
ERROR 2845: Could not create thread for SubsessionHandler Hurry
ERROR 2997: DBDesigner memory usage (value bytes) exceeded system limit
ERROR 3300: Exceeded temp space cap, requested value with value remaining (used value) bytes
ERROR 3416: Filter tried to allocate too much memory (value, out of value allowed)
ERROR 3587: Insufficient resources to execute plan on pool string [string]
ERROR 3921: MemoryPool string used more memory than allowed
ERROR 3937: MIN/MAX window function could not operate in memory
ERROR 4764: Source tried to allocate too much memory (value, out of value allowed)
ERROR 5000: Thread limit value, but statement needs value threads
ERROR 5001: ThreadManager failed to create thread string: string
ERROR 5022: Timer service failed to run value: string
ERROR 5065: Too many ROS containers exist for the following projections: string
ERROR 5921: Insufficient memory available for database designer
ERROR 5924: Insufficient resources to get resource from JVM pool [string]
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 53100

This topic lists the error associated with the SQLSTATE 53100.

SQLSTATE 53100 Description

ERRCODE_DISK_FULL

Error messages associated with this SQLState

```
ERROR 2475: Cannot rebalance cluster. Insufficient disk space on the following nodes: string

ERROR 2927: Could not write to [string]: string

ERROR 5661: Storage Location string has value free bytes left, but the plan requires at least
    value bytes
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 53200

This topic lists the error associated with the SQLSTATE 53200.

SQLSTATE 53200 Description

ERRCODE_OUT_OF_MEMORY

Error messages associated with this SQLState

```
ERROR 2296: Calloc of value bytes for string failed

ERROR 2344: Cannot allocate sufficient memory for COPY statement (value requested, value permi
    tted)

ERROR 3499: Hash table out of memory

ERROR 3811: Join [string] inner partition did not fit in memory; value [string]

ERROR 3813: Join did not fit in memory

ERROR 3814: Join inner did not fit in memory

ERROR 3815: Join inner did not fit in memory [string]

ERROR 3816: Join NULLs did not fit in memory [string]

ERROR 3819: Join table did not fit in memory

ERROR 3895: Malloc of value bytes for string failed

ERROR 4176: Not enough memory for test directive numTopKHeaps

ERROR 4302: Out of memory

ERROR 4303: Out of memory when expanding glob: string

ERROR 4305: Out of system WOS memory during catalog SELECT

ERROR 4357: Pattern partition will not fit into memory

ERROR 4381: Plan memory limit exhausted: [string]

ERROR 4495: Query value exceeded memory usage limit. Design result for this query might be sub
    optimal
```

```
ERROR 4512: Ran out of WOS memory during string
ERROR 4524: Realloc of value bytes for string failed
ERROR 5062: Too many hash table entries
ERROR 5063: Too many matches in a single partition
ERROR 5147: Unable to reserve memory (value K) for the WOS
ERROR 5952: Malloc of value bytes in Block Memory Manager failed
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 54000

This topic lists the error associated with the SQLSTATE 54000.

SQLSTATE 54000 Description

ERRCODE_PROGRAM_LIMIT_EXCEEDED

Error messages associated with this SQLState

```
ERROR 2052: string Row size value is too large
ERROR 2472: Cannot prepare statement - too many prepared statements
ERROR 3460: Function string may give a value-octet result; the limit is value octets
ERROR 3626: Invalid buffer enlargement request size value
ERROR 3866: Line is too long in timezone file "string", line value
ERROR 4282: Operator string may give a value-octet Varbinary result; the limit is value octets
ERROR 4283: Operator string may give a value-octet Varchar result; the limit is value octets
ERROR 4557: regexp_replace result is too long
ERROR 4913: Target lists can have at most value entries
ERROR 5043: Timezone directory stack overflow
ERROR 5060: Too many data partitions
ERROR 5263: Unsupported access to external table
ERROR 5265: Unsupported access to virtual schema
ERROR 5266: Unsupported access to virtual table
ERROR 5267: Unsupported access to virtual view
ERROR 5749: Array size exceeds the maximum allowed (value)
ERROR 5961: Size of compressed serialized plan (value bytes) is too large
ERROR 6064: Transaction commit delta is too large (value)
ERROR 6076: Unable to fork to start spread: value
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that

may help you resolve these errors.

Error Messages Associated with SQLSTATE 54001

This topic lists the error associated with the SQLSTATE 54001.

SQLSTATE 54001 Description

ERRCODE_STATEMENT_TOO_COMPLEX

Error messages associated with this SQLState

```
ERROR 4588: Request size too big. Please try to simplify the query

ERROR 4963: The query contains a SET operation tree that is too complex to analyze

ERROR 4964: The query contains an expression that is too complex to analyze
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 54011

This topic lists the error associated with the SQLSTATE 54011.

SQLSTATE 54011 Description

ERRCODE_TOO_MANY_COLUMNS

Error messages associated with this SQLState

```
ERROR 2106: A table/projection/view can only have up to value columns -- this create statement
    has value


ERROR 2118: Adding column causes row size (value) to exceed MaxRowSize (value)

ERROR 2136: Aggregate function cannot have value input argument(s)

ERROR 2137: Aggregate function cannot have value return value(s)

ERROR 2181: Analytic function cannot have value return value(s)

ERROR 2291: Call to ColumnTypes.addAny() is not allowed in Aggregate functions

ERROR 3466: Function cannot have value return value(s)

ERROR 4202: Number of columns (value) exceeds limit (value)

ERROR 4481: Projection row size (value) exceeds MaxRowSize (value)

ERROR 4630: Row size exceeds MaxRowSize: value > value

ERROR 4875: Table "string" can only have up to value columns -- adding one will exceed this li
    mit

ERROR 5898: File system cannot have value input argument(s)
ERROR 5899: File system cannot have value return value(s)
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 54023

This topic lists the error associated with the SQLSTATE 54023.

SQLSTATE 54023 Description

ERRCODE_TOO_MANY_ARGUMENTS

Error messages associated with this SQLState

```
ERROR 2441: Cannot have more than value segmentation columns
ERROR 2469: Cannot pass more than value arguments to a function
ERROR 4431: Procedures cannot have more than value parameters
ERROR 4646: Scalar/Transform functions cannot have more than value parameters
ERROR 5055: Too many arguments
ERROR 5056: Too many arguments to evaluate_delete_performance function
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 55000

This topic lists the error associated with the SQLSTATE 55000.

SQLSTATE 55000 Description

ERRCODE_OBJECT_NOT_IN_PREREQUISITE_STATE

Error messages associated with this SQLState

```
ERROR 2088: A concurrent load into the partition or a concurrent mergeout operation interfered
    with this statement
ERROR 2143: AHM advanced beyond snapshot epoch
ERROR 2149: AHM can't be set
ERROR 2150: AHM can't be set while retentive refresh is running
ERROR 2151: AHM can't be set. (value nodes are down, out of value.)
ERROR 2152: AHM can't be set. (value nodes are down.)
ERROR 2159: All nodes must be UP to rebalance a cluster
ERROR 2163: Already released
```

ERROR 2174: An error occurred when loading library file *string*. Details: *string*

ERROR 2175: An error occurred when loading library file on node *string*, message: *string*

ERROR 2200: AnalyzeStatsPlanMaxColumns configuration parameter '*value*' invalid; must be greater than zero

ERROR 2201: AnalyzeStatsSampleBands configuration parameter '*value*' invalid; must be greater than zero

ERROR 2241: Attempt to create view using an invalid relation

ERROR 2242: Attempt to run multi-node KV plan

ERROR 2294: CALL_USE_SESSION_NODES used without setting nodes

ERROR 2303: Can not tell if tables have data, too few responses (*value*) to be conclusive

ERROR 2316: Can't match imported node '*string*' to node in current database

ERROR 2371: Cannot commit DML/DDL while a node is shutting down

ERROR 2378: Cannot convert column "*string*" to type "*string*"

ERROR 2380: Cannot create a library without an initialized LibraryPath on node: *string*

ERROR 2388: Cannot create projections on a temporary table that has data

ERROR 2409: Cannot drop any more columns in *string*

ERROR 2410: Cannot drop column "*string*" since it is referenced in the default expression of column "*string*"

ERROR 2413: Cannot drop column "*string*" since it was referenced in the default expression of added column "*string*"

ERROR 2422: Cannot Drop: *string string* depends on *string string*

ERROR 2424: Cannot execute query with temporary table because a node has recovered since the start of this session

ERROR 2448: Cannot issue this command in a read-only transaction

ERROR 2459: Cannot modify temporary table *string* because a node has recovered or rebalance data took place since the start of this *string*

ERROR 2467: Cannot overwrite object

ERROR 2476: Cannot reference Storage

ERROR 2483: Cannot remove snapshots without an initialized SnapshotPath

ERROR 2496: Cannot revoke EXECUTE permission from the owner: *string*

ERROR 2497: Cannot revoke EXECUTE permission from the super user

ERROR 2498: Cannot revoke USAGE permissions on the resource pool to which user *string* is assigned

ERROR 2505: Cannot set column "*string*" in table "*string*" to NOT NULL since it contains null values

ERROR 2512: Cannot set memoryCap for session whose current user has been dropped

ERROR 2516: Cannot set runTimeCap for session whose current user has been dropped

ERROR 2517: Cannot set tempSpaceCap for session whose current user has been dropped

ERROR 2531: Cannot undelete storage

ERROR 2541: Cannot use addAny() with any other input column types

ERROR 2542: Cannot use addAny() with any other output column types

ERROR 2550: Cannot use KV hint with a non-query

ERROR 2551: Cannot use KV query inside a transaction

ERROR 2563: Cannot validate DV storage

ERROR 2564: Cannot validate storage

ERROR 2587: Changes cannot be made to [*string*]. It has been retired

ERROR 2691: Concurrent DDL interfered with this statement

ERROR 2762: COPY: Cannot load into IDENTITY column "*string*"

ERROR 2763: COPY: Cannot specify parsing options for IDENTITY column "*string*"

ERROR 2903: Could not reset epoch because DML locks are held

ERROR 2904: Could not reset epoch because projections exist

ERROR 2933: Couldn't force partition projection *string*

ERROR 2934: Couldn't force partition projections *string*

ERROR 2954: Current phase of recovery failed due to missed event at epoch *value*

ERROR 2955: Current set of up nodes do not satisfy dependencies

ERROR 2956: Current set of up nodes do not satisfy dependencies for table *string*

ERROR 2961: Current user has been dropped so no defaults are available

ERROR 2962: Current user has been dropped so no roles are available

ERROR 2969: Cursor can only scan forward

ERROR 3000: DDL interfered with this statement

ERROR 3002: DDL statement interfered with alter column type

ERROR 3018: Default expression of IDENTITY/AUTO_INCREMENT column "*string*" cannot be altered

ERROR 3136: drop_partition failed for *string* on node *string*. The projection contains unpartiti
oned data

ERROR 3196: Error deserializing objects

ERROR 3216: Error during setting up function *string*, message: *string*

ERROR 3229: Error loading library file:[*string*]

ERROR 3254: Error reading from file

ERROR 3278: Error writing to file

ERROR 3318: Execution aborted by node state change

ERROR 3392: Failed to update local min/max objects for column "*string*"

ERROR 3807: JobTracker::getMarkedStorages(): Unknown job *value*

ERROR 3808: JobTracker::jobComplete(*string*): Unknown job *value*

ERROR 3809: JobTracker::setDetails(*value*,*value*,*value*): Unknown job *value*

ERROR 3810: JobTracker::setJobDescription(*string*): Unknown job *value*

ERROR 3838: Key *value* already in use

ERROR 3882: Location cannot be dropped as it stores data files

ERROR 3911: maxMemorySize for *string* can be changed only when the *string* WOS is empty

ERROR 3924: merge_partitions() failed on *string* because of unpartitioned data

ERROR 3925: Mergeout failed: projection *string* is not up-to-date

ERROR 4032: Naming conflict: *string* exists

ERROR 4092: No plan received at node

ERROR 4120: No transaction running on node

ERROR 4121: No transaction running, does previous load_snapshot_prep succeeded?

ERROR 4127: No valid cache found

ERROR 4138: Node *string* is not available for queries

ERROR 4144: Node has not been set up for plan execution

ERROR 4146: Node is not active or recovering, cannot plan query

ERROR 4148: Node not prepared to accept plan

ERROR 4151: Node unprepared for rebalance

ERROR 4177: Not enough nodes are up for Projection <*string*> to be available, marking it as out of date

ERROR 4219: Object oid *value* reused

ERROR 4403: Portal "*string*" cannot be run

ERROR 4457: Projection *string* checkpoint epoch lags snapshot epoch

ERROR 4458: Projection *string* contains data in the WOS

ERROR 4459: Projection *string* create epoch is greater than the epoch in the query

ERROR 4462: Projection *string* has HSE > snapshot epoch and buddy *string* has HSE <= snapshot epoch

ERROR 4464: Projection *string* is not up-to-date

ERROR 4467: Projection (name: *string*, oid: *value*) is newly added during current recovery

ERROR 4485: Projections *string* contain data in the WOS

ERROR 4530: Rebalance unable to moveout all data on projection *string*

ERROR 4592: reset_epoch is disabled because the EnableResetEpoch configuration parameter is 0

ERROR 4611: Returned string value '[*string*]' with length [*value*] is greater than declared field length of [*value*] of field [*string*] at output column index [*value*]

ERROR 4698: Sequence "*string*" has been created by an IDENTITY/AUTO_INCREMENT column and cannot be dropped

ERROR 4699: Sequence "*string*" has been created by an IDENTITY/AUTO_INCREMENT column and cannot be used in a default expression

ERROR 4700: Sequence *string* has not been accessed in the session

ERROR 4757: SnapshotMemento does not match. Oid conflicts are possible

ERROR 4760: Some nodes are down. These nodes will not receive the configuration change unless a manual step is taken, or the set_config_parameter utility is reissued after the node is brought back up

ERROR 4765: Specified K-safety for projection creation is insufficient to support currently down nodes

ERROR 4791: Storage extends beyond specified segment range

ERROR 4793: Stream error: *string*

ERROR 4860: System is not k-safe. DDL is disallowed

ERROR 4861: System is not k-safe. DDL/DML is disallowed

ERROR 4879: Table "*string*" has projections in non-up-to-date state

ERROR 4880: Table "*string*" has projections that are not up-to-date that can refresh from buddy

ERROR 4903: Table no longer exists

ERROR 4934: The attribute "*string*" in table "*string*" needs to be included in projection "*string*" because it is used in the partitioning expression

ERROR 4941: The data type requires length/precision specification

ERROR 4965: The restore violates K safety

ERROR 4972: The types/sizes of source column (index *value*, length *value*) and destination column (index *value*, length *value*) do not match

ERROR 5049: TM interfered with object-level backup

ERROR 5084: Tried to add field '*string*' that already exists

ERROR 5085: Tried to add unknown node '*string*' to user-defined query plan

ERROR 5132: Unable to evaluate the delete performance after dropping this column for projection "*string*"

ERROR 5151: Unable to validate data in *string*: *string*

ERROR 5204: Unknown data type

ERROR 5210: Unknown object: *string*

ERROR 5321: Usage of [*string*] cannot be changed. It has been retired

ERROR 5381: User Defined Scalar Function can only have 1 return column, but *value* is provided

ERROR 5491: Wrong MD5 checksum for library file *string*

ERROR 5522: A concurrent operation interfered with this statement

ERROR 5533: Can not move partition to the same table

ERROR 5534: Can't create table in specified target schema

ERROR 5535: Can't find target table's schema

ERROR 5543: Cannot use column type 'any' with any other input column types

ERROR 5544: Cannot use column type 'any' with any other output column types

ERROR 5568: DVWos can not be moved

ERROR 5572: Error during setting up type *string*, message: *string*

ERROR 5583: Fault Group "*string*" already exists in a fault group

ERROR 5587: Fault Group "*string*" not found in Fault Group "*string*"

ERROR 5590: Found *value* unsegmented projections with basename *string*; inconsistent with permanent nodes count *value*

ERROR 5626: Node "*string*" already exists in a fault group

ERROR 5627: Node "*string*" not found in Fault Group "*string*"

ERROR 5659: Source and target table does not match

ERROR 5660: Source table can not be temp, virtual, system, or external

ERROR 5662: Storage tier *string* has not been found on all nodes

ERROR 5666: Table "*string*" has prejoin projections

ERROR 5667: Target table can not be temp, virtual, system, or external

ERROR 5674: TM interfered with this statement

ERROR 5676: Unable to move partitions because some projection(s) contain unpartitioned data

ERROR 5705: Dvmergeout failed: projection *string* is not up-to-date

ERROR 5712: JobTracker::reportStart: Unknown job *value*

ERROR 5735: Tier *string* is referenced by storage policies. Can not make storage location changes as requested

ERROR 5742: A design/deployment process is currently executing in this design space

ERROR 5760: Can only change setting when all started nodes are UP

```
ERROR 5765: Cannot change control node away from self because other nodes depend on this node
    to be their control node
ERROR 5766: Cannot change final control node away from self until at least one other node is p
    romoted to be a control node
ERROR 5772: Cannot manually alter automatically generated fault groups
ERROR 5786: Column value does not have corresponding storages yet. A concurrent add column ope
    ration might be running
ERROR 5883: Failed to list hcatalog tables
ERROR 6001: Recovery failed because DVROS straddles discard epoch
ERROR 6002: Recovery failed because ROS value [0xvalue, 0xvalue] straddles endEpoch value to d
    iscard
ERROR 6003: Recovery failed because ROS straddles discard epoch
ERROR 6035: Table "string" has no non-null records under the column key_name
ERROR 6037: Table "string_string" cannot be found or was not created internally
ERROR 6065: Tried to allocate and initialize a value-byte string with value zero bytes; VStrin
    g is too small
ERROR 6066: Tried to copy a value-byte string to value-byte VString object; VString is too sma
    ll
ERROR 6070: Trying to set the column "string" to size of value All data type lengths in table
    "string" must not be greater than value, the current maximum raw size for flex table valu
    es. If you need a larger value, please contact your Feedback Program coordinators
ERROR 6078: Unable to move partitions as some nodes are doing recovery
ERROR 6105: View "string" is already linked to flex table "string".  Linked views will not be
    overwritten
ERROR 6106: View "string" is already linked to this table.  Linked views will not be overwritt
    en
ERROR 6107: View "string_string" cannot be found or was not created internally
ERROR 6109: WebHCat query (string) failed: string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 55006

This topic lists the error associated with the SQLSTATE 55006.

SQLSTATE 55006 Description

ERRCODE_OBJECT_IN_USE

Error messages associated with this SQLState

```
ERROR 2060: string WOS is not empty; cannot renew. Do a moveout
ERROR 2307: Can't drop self
ERROR 3003: DDL statement interfered with Database Designer
ERROR 3004: DDL statement interfered with query replan
ERROR 3896: Manual mergeout not supported while tuple mover is running
ERROR 3897: Manual moveout not supported while tuple mover is running
ERROR 4122: No up-to-date super projection left on the anchor table of projection string
ERROR 4139: Node string transitioned to state UP during this statement
ERROR 4145: Node is active and cannot be altered
ERROR 4455: Projection string cannot be dropped because K-safety would be violated
ERROR 4470: Projection cannot be dropped because history after AHM would be lost
ERROR 4488: Projections cannot be dropped or data would be lost due to down nodes
ERROR 4527: Rebalance is already running
ERROR 4528: Rebalance is already scheduled to run in the background
ERROR 4882: Table "string" is used as a dimension in a prejoined projection
ERROR 4896: Table (value) has been dropped
ERROR 4971: The status of one or more nodes changed during query planning
ERROR 6052: The system must retain at least one control node after the drop
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 55V02

This topic lists the error associated with the SQLSTATE 55V02.

SQLSTATE 55V02 Description

ERRCODE_CANT_CHANGE_RUNTIME_PARAM

Error messages associated with this SQLState

```
ERROR 4324: Parameter will not take effect until database restart
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 55V03

This topic lists the error associated with the SQLSTATE 55V03.

SQLSTATE 55V03 Description

ERRCODE_LOCK_NOT_AVAILABLE

Error messages associated with this SQLState

```
ERROR 5156: Unavailable: *string* - Locking failure: *string*
ERROR 5157: Unavailable: [Txn *value*] *string* - *string* error *string*
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 55V04

This topic lists the error associated with the SQLSTATE 55V04.

SQLSTATE 55V04 Description

ERRCODE_TM_MARKER_NOT_AVAILABLE

Error messages associated with this SQLState

```
ERROR 2082: A string operation is already in progress on projection string.string [container v
    alue txnid value session string]
ERROR 2083: A string operation is already in progress on projection string.string [txnid value
    session string]
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 57014

This topic lists the error associated with the SQLSTATE 57014.

SQLSTATE 57014 Description

ERRCODE_QUERY_CANCELED

Error messages associated with this SQLState

```
ERROR 2246: Audit canceled
ERROR 2279: Bulk Import canceled
ERROR 2310: Can't find projection
ERROR 2325: Canceled (in string)
ERROR 2326: Canceled: string - Locking canceled: string
ERROR 2327: Canceled: [Txn value] string - string string
ERROR 2576: Catchup recovery interrupted
ERROR 2704: Connection canceled
ERROR 2996: DBDesigner canceled by user
ERROR 3086: Design/Deployment canceled by user
ERROR 3286: evaluate_delete_performance canceled
ERROR 3319: Execution canceled (compile)
ERROR 3320: Execution canceled (prepare)
ERROR 3321: Execution canceled (start)
ERROR 3322: Execution canceled by operator
ERROR 3323: Execution got unlucky!
ERROR 3324: Execution intentionally failed
ERROR 3326: Execution time exceeded run time cap of string
ERROR 3515: import_catalog_objects canceled
ERROR 4114: No super projection available for analyze_statistics
ERROR 4142: Node failure during execution
ERROR 4143: Node failure in string
ERROR 4287: Operator intervention on string
ERROR 4380: Plan canceled prior to execute call
ERROR 4439: Processing aborted by peer on string
ERROR 4496: Query canceled while waiting for resources
ERROR 4787: Statement abandoned due to subsequent DDL
ERROR 4789: Statement is canceled
ERROR 4843: Subsession interrupted
ERROR 5757: build_flextable_view canceled
ERROR 5787: compute_flextable_keys canceled
ERROR 5915: Hcatalog webservices query canceled
ERROR 5953: materialize_flextable_columns canceled
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 57015

This topic lists the error associated with the SQLSTATE 57015.

SQLSTATE 57015 Description

ERRCODE_SLOW_DELETE

Error messages associated with this SQLState

```
ERROR 5822: Detected slow delete
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 57V01

This topic lists the error associated with the SQLSTATE 57V01.

SQLSTATE 57V01 Description

ERRCODE_ADMIN_SHUTDOWN

Error messages associated with this SQLState

```
ERROR 3556: Initiating node is down
ERROR 4150: Node status is not UP
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 57V03

This topic lists the error associated with the SQLSTATE 57V03.

SQLSTATE 57V03 Description

ERRCODE_CANNOT_CONNECT_NOW

Error messages associated with this SQLState

```
ERROR 2863: Could not fork UDx zygote process, string
ERROR 2929: Couldn't create new UDx side process, failed to get UDx side process info from zyg
    ote: string
ERROR 2930: Couldn't create new UDx side process, the language string is not supported
ERROR 2937: Couldn't set TCP_NODELAY option, might get latency in RPC message delivery: string
ERROR 3363: Failed to connect to side process, string
ERROR 3364: Failed to connect to UDx zygote, string
ERROR 3366: Failed to create new UDx side process, couldn't connect to it: string
ERROR 4720: Session manager cannot add an external session - disabled
ERROR 4973: The UDx zygote process is down, restarting it...
ERROR 5699: Cannot find java binary: neither the Linux environment variable JAVA_HOME nor Vert
    ica config parameter JavaBinaryForUDx is set
ERROR 5702: Couldn't create new UDx side process: string
ERROR 5803: Couldn't create new UDx side process, failed to set locale information: string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 58030

This topic lists the error associated with the SQLSTATE 58030.

SQLSTATE 58030 Description

ERRCODE_IO_ERROR

Error messages associated with this SQLState

ERROR 2024: *string* Error occurred during BZIP decompression. BZIP error code: *value*

ERROR 2026: *string* Error occurred during ZLIB decompression. ZLIB error code: *value*, Message: *string*

ERROR 2253: Bad return from WaitForMultipleObjects: *value* (*value*)

ERROR 2432: Cannot get LibraryPath from node: *string*

ERROR 2433: Cannot get MD5 checksum from node: *string*

ERROR 2600: Checksums do not match (computed=0x*value*, fromdisk=0x*value*) discarding checkpoint!

ERROR 2674: ColumnAccessBase open error

ERROR 3197: Error deserializing snapshot info from file *string*

ERROR 3255: Error reading from file *string*

ERROR 3303: Exception during measurement deserialization

ERROR 3304: Exception during ProjectionSnapshot deserialization:*string*

ERROR 3305: Exception during Stats deserialization:*string*

ERROR 3370: Failed to create socket waiting event: *value*

ERROR 3385: Failed to reset socket waiting event: *value*

ERROR 3408: File size on disk does not match catalog for *string*

ERROR 3412: FileColumnReader: Get block *string* @ *value* error

ERROR 3478: getnameinfo_all() failed: *string*

ERROR 3550: Info file *string* does not exist

ERROR 3796: IO_ERROR writing data file [*string*]

ERROR 4364: Performance measurement of [*string*] failed

ERROR 4377: Pixw finish error

ERROR 4378: Pixw open error

ERROR 4379: Pixw write error

ERROR 4518: Read error when expanding glob: *string*

ERROR 4632: RowAccessBase open error

ERROR 5124: Unable to close catalog file [*string*]

ERROR 5126: Unable to create catalog file [*string*]

ERROR 5131: Unable to drop catalog file [*string*]

```
ERROR 5133: Unable to fsync catalog file [string] errno=value
ERROR 5141: Unable to open file [string]
ERROR 5152: Unable to write catalog file [string]
ERROR 5153: Unable to write checksum to catalog file [string]
ERROR 5154: Unable to write object to catalog file [string]
ERROR 5887: Failed to mount file system value: string
ERROR 5901: Filesystem does not pass basic test: string
ERROR 5902: Filesystem does not pass basic test: I/O data differ
ERROR 6074: Unable to close catalog file after fsync [string] errno=value
ERROR 6077: Unable to fsync catalog dir [string] errno=value
ERROR 6079: Unable to open catalog dir fd for fsync [string] errno=value
ERROR 6080: Unable to open catalog dir for fsync [string] errno=value
ERROR 6081: Unable to open catalog file for fsync [string] errno=value
ERROR 6082: Unable to open spread conf file string for writing
ERROR 6084: Unable to stat file string: string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE 58V01

This topic lists the error associated with the SQLSTATE 58V01.

SQLSTATE 58V01 Description

ERRCODE_UNDEFINED_FILE

Error messages associated with this SQLState

```
ERROR 3664: Invalid filename. Input filename is an empty string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE V1001

This topic lists the error associated with the SQLSTATE V1001.

SQLSTATE V1001 Description

ERRCODE_LOST_CONNECTIVITY

Error messages associated with this SQLState

```
ERROR 2709: Connection to spread closed
ERROR 4048: NetworkReceive: Decompression failed
ERROR 4054: NetworkSend on string: failed to open connection to node string (string)
ERROR 4055: NetworkSend on string: failed to send to node string [string]
ERROR 4140: Node string was not successfully added to the cluster
ERROR 4533: Receive: Decompression failed
ERROR 4534: Receive on string: Message receipt from string failed [string]
ERROR 4536: Receive on string: open failed for node string (string)
ERROR 4541: ReceiveFiles on string: Unexpected end of stream from string [string]
ERROR 4547: RecvFiles on string: Open failed on node [string] (string)
ERROR 4572: RemoteSend: Open failed on node [string] (string)
ERROR 4683: Send: Connection not open [string tag:value plan value]
ERROR 4684: Send: Open failed on node [string] (string)
ERROR 4689: SendFiles on string: Open failed on node [string] (string)
ERROR 5579: Failure in send on socket string: string
ERROR 5624: NetworkReceive on string: failed to open connection to node string (string)
ERROR 5625: NetworkReceive on string: Message receipt from string failed: string
ERROR 5658: Send on string: Open failed on node [string] (Address lookup for string(string) fa
    iled)
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE V1002

This topic lists the error associated with the SQLSTATE V1002.

SQLSTATE V1002 Description

ERRCODE_K_SAFETY_VIOLATION

Error messages associated with this SQLState

```
ERROR 2406: Cannot drop value nodes from a value node cluster with value nodes down - cluster
    would appear partitioned and database would shutdown.  Bring some nodes up and try again
ERROR 2529: Cannot support K=value on only value nodes
ERROR 2957: Current system KSAFE level is not fault tolerant
ERROR 4477: Projection KSAFE value can not be met with only value nodes
ERROR 4478: Projection KSAFE override value cannot be less than current system K-safe value va
    lue
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE V1003

This topic lists the error associated with the SQLSTATE V1003.

SQLSTATE V1003 Description

ERRCODE_CLUSTER_CHANGE

Error messages associated with this SQLState

```
ERROR 2094: A node has come UP which missed ALTER COLUMN check
ERROR 2095: A node has come UP which missed drop partition keys check
ERROR 2096: A node has come UP which missed partitioning check
ERROR 2097: A node has entered the cluster since the session started
ERROR 2098: A node has entered the cluster since the session was started
ERROR 2099: A node has entered/left the database cluster
ERROR 3428: Following nodes are UP but not in the backup node set: string
ERROR 3941: Mismatch between plan and session node states  (possibly because a node entered/le
    ft the cluster since the session was started)
ERROR 5312: Up node set changed in between load_snapshot_prep and load_snapshot
ERROR 5523: A node has come UP which missed ADD COLUMN statement
ERROR 6011: Restoring to a cluster with increased number of nodes requires  UP/DOWN states of
    all nodes during backup remain the same during restore; UP nodes during backup: string; UP
    nodes during restore (excluding new nodes): string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE V2000

This topic lists the error associated with the SQLSTATE V2000.

SQLSTATE V2000 Description

ERRCODE_AUTH_FAILED

Error messages associated with this SQLState

```
ERROR 3493: GSS error: string. Error details: (string/string)
ERROR 3718: Invalid old password
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE V2001

This topic lists the error associated with the SQLSTATE V2001.

SQLSTATE V2001 Description

ERRCODE_LICENSE_ISSUE

Error messages associated with this SQLState

```
ERROR 2382: Cannot create another node. The current license permits value node(s) and the data
    base catalog already contains value node(s)

ERROR 2447: Cannot install new license to the database. New license permits value node(s) but
    the database catalog already contains value node(s)

ERROR 3248: Error parsing license end date

ERROR 3863: License issue: string

ERROR 4943: The Enterprise Edition is installed. You cannot downgrade from the Enterprise Edit
    ion to the Community Edition

ERROR 5943: License corrupt: string requires license string, but it is corrupt

ERROR 5944: License expired: string requires license string, but it has expired

ERROR 5946: License issue: string (required by string)

ERROR 5955: Missing license: string requires license string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE VC001

This topic lists the error associated with the SQLSTATE VC001.

SQLSTATE VC001 Description

ERRCODE_CONFIG_FILE_ERROR

Error messages associated with this SQLState

```
ERROR 2879: Could not load server certificate file "string": string

ERROR 3833: Kerberos keytab file must be owned by the database user, and have no permissions f
    or "group" or "other"

ERROR 4951: The Kerberos keytab file is either empty or too small in size to be valid

ERROR 5261: Unsafe permissions on private key file "string"
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE VD001

This topic lists the error associated with the SQLSTATE VD001.

SQLSTATE VD001 Description

ERRCODE_DESIGNER_FUNCTION_ERROR

Error messages associated with this SQLState

ERROR 2010: *string* cannot be NULL

ERROR 2012: *string* clause does not exist in the query

ERROR 2202: Anchor table for projection *string* does not exist, so it cannot be added to deploy ment

ERROR 2204: Anchor table of projection *string* is a Session scoped table

ERROR 2205: Anchor table of projection *string* is a System table

ERROR 2211: API *string* not available in old DBD engine

ERROR 2212: API cannot take query input file and query string, only one can be set

ERROR 2304: Can only load *value string* under the *string* design type

ERROR 2328: Cannot *string* as design was created already

ERROR 2336: Cannot add another Comprehensive design to deployment *string*

ERROR 2337: Cannot add design projections in extend catalog type deployment *string* in workspac e *string*

ERROR 2338: Cannot add design tables to design *string* because there are populated designs

ERROR 2339: Cannot add design to deployment *string* because design *string* has not been populate d

ERROR 2369: Cannot clear design tables from design *string* because there are populated designs

ERROR 2375: Cannot compute projections to be dropped for only incremental designs deployment

ERROR 2384: Cannot create projections for a NULL table

ERROR 2385: Cannot create projections for a system table

ERROR 2386: Cannot create projections for a temporary table

ERROR 2394: Cannot design encoding for Projection *string* as it does not have any AUTO encoded columns

ERROR 2395: Cannot design encoding for Projection *string* as it is not SAFE -- Create its buddi es

ERROR 2396: Cannot design/deploy for virtual system schema *string*

ERROR 2423: Cannot execute deployment when there are non-up-to-date safe projections for table *string*

ERROR 2434: Cannot get script for a NULL table

ERROR 2435: Cannot get script for a system table

ERROR 2436: Cannot get script for a temporary table

ERROR 2454: Cannot load invalid query: *string*

ERROR 2456: Cannot load queries as design was populated already

ERROR 2463: Cannot output design projections because design is not available

ERROR 2464: Cannot output query because query id is invalid

ERROR 2471: Cannot populate drop projections in extend catalog type deployment *string* in works pace *string*

ERROR 2477: Cannot refresh projections for table *value* as it was dropped

ERROR 2480: Cannot remove any design table from design *string* because there are populated desi gns

ERROR 2485: Cannot remove workspace *string* because it does not exist

ERROR 2492: Cannot retrieve design tables for design *string* in workspace *string*

ERROR 2493: Cannot retrieve information for design *string* in workspace *string*

ERROR 2507: Cannot set k-safety when design *string* has been populated

ERROR 2657: Column '*string*' does not exist in Table *string*

ERROR 2658: Column '*string*' is duplicated in the column list

ERROR 3053: Deployment *string* already exists in workspace *string*

ERROR 3054: Deployment got canceled

ERROR 3056: Deployment ksafety should be equal or greater than design ksafety. Deployment ksa fety is *value* and design ksafety is *value*

ERROR 3057: Deployment name cannot be NULL

ERROR 3058: Deployment Projections status is set to Error

ERROR 3060: Design *string* already exists

ERROR 3061: Design *string* already exists for workspace *string*

ERROR 3063: Design *string* has already been added to deployment *string*

ERROR 3064: Design *string* has not been populated in workspace *string* so projection cannot be a dded

ERROR 3065: Design *string* hasn't been populated

ERROR 3066: Design *string* in workspace *string* is not available

ERROR 3067: Design *string* is already populated

ERROR 3068: Design *string* is populated, remove design first (designer_remove_design)

ERROR 3071: Design name cannot have more than *value* characters

ERROR 3072: Design name may contain only alphanumeric or underscore characters

ERROR 3073: Design did not complete successfully, so deployment did not start

ERROR 3074: Design K-safety should be 0

ERROR 3077: Design name cannot have character '.'

ERROR 3079: Optimization objective cannot be NULL

ERROR 3080: Design query with design_query_id *value* does not exist

ERROR 3081: Design Query with design_query_id *string* does not exist

ERROR 3082: Design *string* does not exist

ERROR 3087: design_override_type *string* for query (design_query_id *value*) already exists

ERROR 3088: design_override_type *string* for table *string* already exists

ERROR 3089: design_override_type *string* for table *string* does not exist

ERROR 3100: Did not find any projections to design encodings for

ERROR 3101: Did not find design projections for projection ids given

ERROR 3102: Did not find design projections for tablePattern *string*

ERROR 3103: Did not find design tables to add

ERROR 3104: Did not find design tables to remove

ERROR 3105: Did not find projection id *value* in deployment *string* in workspace *string*

ERROR 3106: Did not find projections for design *string* in workspace *string*

ERROR 3107: Did not find rows in deployment table for deployment *string* in workspace *string*

ERROR 3108: Did not find rows in designs table for workspace *string*

ERROR 3140: Dropping design without getting design projections, API call is of no use

ERROR 3166: Empty design name is not allowed

ERROR 3188: Error after projection refresh: *string*

ERROR 3194: Error creating workspace: Invalid workspace name

ERROR 3195: Error deleting deployment status table

ERROR 3202: Error during deployment while setting ksafety before deployment starts

ERROR 3203: Error during design: *string*

ERROR 3205: Error during drop projections: *string*

ERROR 3208: Error during projection creation: *string*

ERROR 3214: Error during remove design *string*

ERROR 3215: Error during rename projections: *string*

ERROR 3241: Error opening query input file [*string*]

ERROR 3250: Error querying deployment projections statements table

ERROR 3251: Error querying deployment projections table

ERROR 3252: Error querying design projections table for design *string* in workspace *string*

ERROR 3253: Error querying: *string*

ERROR 3266: Error status for projections to add for table *string*

ERROR 3267: Error status for projections to drop for table *string*

ERROR 3268: Error updating deployment projections table

ERROR 3270: Error while loading statistics into design tables for design *string*

ERROR 3277: Error writing to *string*

ERROR 3356: External table *string* is not a design table

ERROR 3358: Failed during select mark_design_ksafe(*value*)

ERROR 3415: Filename cannot be NULL

ERROR 3480: Given design *string* does not exist

ERROR 3489: Group-by override *value* on query *value* cannot be satisfied

ERROR 3543: Incremental design needs a query or an input query file to be set

ERROR 3574: INSERT query without SELECT is not supported: *string*

ERROR 3649: Invalid Deploy Operation string *string*

ERROR 3650: Invalid deploy status string *string*

```
ERROR 3740: Invalid query input file [string]

ERROR 3795: Invalid design creator name

ERROR 3817: Join override value on query value cannot be satisfied

ERROR 3824: K cannot be value (maximum allowed is value)

ERROR 3825: K must be equal to or greater than value, cannot reduce current k-safety level

ERROR 3826: K-safety can be between 0 and value

ERROR 3827: K-safety cannot be NULL

ERROR 3867: List of projections cannot be NULL

ERROR 3898: mark_design_ksafe(value) failed; some projections may not be k-safe

ERROR 4031: Namespace for LOCAL temporary tables cannot be used to add design tables

ERROR 4057: New ksafety cannot be less than 0

ERROR 4078: No deployment data in string.string

ERROR 4080: No drop entries found for deployment string in workspace string

ERROR 4099: No projections found for the projection ids string string

ERROR 4103: No rows in deployment projections table

ERROR 4117: No tables found in schema string

ERROR 4118: No tables found in the table pattern given

ERROR 4119: No tables to design projections for

ERROR 4235: One of the design tables no longer exist

ERROR 4311: Override (design_override_id value) is ignored because the table string is no long
    er a design table

ERROR 4312: Override (design_override_id value) is ignored because the table does not exist

ERROR 4313: override_type string for query (design_query_id value) does not exist

ERROR 4314: override_type string is invalid

ERROR 4460: Projection string does not exist

ERROR 4461: Projection string does not exist

ERROR 4466: Projection string to be refreshed was dropped

ERROR 4475: Projection id cannot be NULL

ERROR 4476: Projection id list cannot be NULL

ERROR 4479: Projection name cannot be NULL

ERROR 4497: Query Id cannot be NULL

ERROR 4498: Query referencing EPOCH column is not supported

ERROR 4499: Query referencing local temporary table string is not supported: string

ERROR 4500: Query referencing projection string is not supported: string

ERROR 4501: Query without referencing any catalog table is not supported: string

ERROR 4503: Query table string does not exist

ERROR 4504: Query table contains multiple entries with qid = value

ERROR 4505: Query weight must be a positive number

ERROR 4525: Rebalance data cannot proceed when there are non-up-to-date projections in the cat
    alog

ERROR 4526: Rebalance data failed during select mark_design_ksafe(value)
```

ERROR 4651: Schema *string* does not exist

ERROR 4652: Schema *string* is not a designer created schema, so it cannot be dropped

ERROR 4655: Schema name cannot be NULL

ERROR 4721: Session scoped table *string* is not a design table

ERROR 4783: Start deploy: deploy is already running on this node

ERROR 4819: Subqueries in UPDATE/DELETE is not supported: *string*

ERROR 4866: System table *string* is not a design table

ERROR 4874: Systems tables within system schema *string* cannot be added as design tables

ERROR 4885: Table *string* does not exist

ERROR 4886: Table *string* does not exist anymore in the catalog

ERROR 4888: Table *string* has no statistics or data. As a result, the proposed projections on this table may be suboptimal

ERROR 4890: Table *string* is not a design table

ERROR 4891: Table *string* is not a design table, referenced in query (qid=*value*): *string*

ERROR 4902: Table name cannot be NULL

ERROR 4907: Table pattern cannot be NULL

ERROR 4920: Terminated after SO enum. See log for the content of the SOs

ERROR 4942: The design table entry with table name *string.string* is corrupted, as that table has been renamed in the Vertica catalog

ERROR 4976: There are *value* nodes. Deployment K = *value* is not possible

ERROR 4977: There are no projections to add in deployment *string* for workspace *string* so no projections can be dropped

ERROR 4980: There is 1 node. Deployment K = *value* is not possible

ERROR 4981: There is more than one design *string* in workspace *string*

ERROR 4983: There is no design tables system table in workspace *string*

ERROR 4991: This invalid query cannot be loaded: *string*

ERROR 4994: This non-SELECT query is not supported: *string*

ERROR 4995: This query is not supported in DBDesigner

ERROR 5363: User *string* does not have privileges to access design table: *string*

ERROR 5364: User *string* does not have privileges to access table: *string*

ERROR 5390: User has insufficient privileges on table *string*

ERROR 5480: Workspace *string* cannot be a virtual system schema

ERROR 5481: Workspace *string* does not exist

ERROR 5482: Design *string* is configured for extend_catalog so no designs can be computed

ERROR 5483: Design *string* is configured for extend_catalog so remove drops is not supported

ERROR 5484: Design *string* is configured for extend_catalog so there are no design tables

ERROR 5485: Design *string* is configured for extend_catalog, there are no design tables

ERROR 5486: Workspace cannot be NULL

ERROR 5487: Design name cannot be NULL

ERROR 5564: Deployment Parallelism cannot be less than zero

ERROR 5565: Deployment parallelism cannot be NULL

ERROR 5573: Error generating results set

ERROR 5575: Error querying designs table

ERROR 5588: Fenced mode false is not supported for *string* functions

ERROR 5589: Fenced mode is not supported for SQL functions

ERROR 5591: Hurryup parameter cannot be NULL

ERROR 5597: Invalid input query: '*string*'

ERROR 5650: Query without referencing any design tables is not supported: *string*

ERROR 5657: Segmentation type of the projection *string* is not supported for encoding design, skipping

ERROR 5694: Weight for query_text '*string*' is '*value*'. Only positive weight values are accepted

ERROR 5747: analyzeStats flag cannot be NULL

ERROR 5773: Cannot output deployment script because design is not available

ERROR 5792: continueAfterError flag cannot be NULL

ERROR 5817: Deploy flag cannot be NULL

ERROR 5855: Did not find any tables to analyze correlations on

ERROR 5857: dropDesignAndCtx flag cannot be NULL

ERROR 5858: dropProjs flag cannot be NULL

ERROR 5866: Error while analyzing correlations for design table *string.string*

ERROR 5867: Error while analyzing count distincts for design table *string.string*

ERROR 5868: Error while analyzing count distincts on correlation sample for design table *string.string*

ERROR 5869: Error while analyzing segmentation skew for design table *string.string*

ERROR 5870: Error while dropping existing correlations in design table *string*

ERROR 5871: Error while loading or analyzing correlations in design tables for design *string*

ERROR 5907: Force option cannot be NULL

ERROR 5908: forceRecomputation flag cannot be NULL

ERROR 5919: Input cannot be NULL

ERROR 5938: isAdminUser flag cannot be NULL

ERROR 5939: K-safety of incremental designs must match the current system k-safety (which is *value*)

ERROR 5979: onlyScript flag cannot be NULL

ERROR 5980: outputScript flag cannot be NULL

ERROR 6040: Table *string* has no correlations

ERROR 6041: Table *string* has no data. As a result, no correlations were analyzed on this table

ERROR 6042: Table *string* has no statistics or data. As a result, no correlations were analyzed on this table

ERROR 6043: Table *string* has no statistics or data. As a result, no correlations were read into this table

ERROR 6049: The mode of analyzing correlations cannot be NULL

ERROR 6050: The mode of analyzing correlations is invalid

ERROR 6096: User has insufficient privileges on table *string.string*

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE VP000

This topic lists the error associated with the SQLSTATE VP000.

SQLSTATE VP000 Description

ERRCODE_USER_PROC_ERROR

Error messages associated with this SQLState

ERROR 2059: *string* with specified name and parameters does not exist: *string*

ERROR 2315: Can't have more than one parameters with the same name: *string*

ERROR 3354: External procedures directory has not been set

ERROR 3355: External procedures have not been installed

ERROR 3465: Function cannot be moved into a system schema

ERROR 4322: Parameter must have names

ERROR 4323: Parameter type is not valid for an external procedure: *string*

ERROR 4373: Phase *value* of multi-phase transform function marked prepass

ERROR 4430: Procedures cannot be created in a system schema

ERROR 5232: Unrecognized identifier: *string*

ERROR 5368: User Defined Aggregates do not support fenced execution mode

ERROR 5372: User Defined Function type not found

ERROR 5374: User Defined Scalar Function *string* is giving bad numeric precision *value*, the max imum is *value*

ERROR 5375: User Defined Scalar Function *string* is giving bad string typmod *value*, the minimum is *value*

ERROR 5376: User Defined Scalar Function *string* is giving typmod of precision *value*, larger th an the max precision *value*

ERROR 5377: User Defined Scalar Function *string* provided non-zero precision (*value*) for Interv al Year To Month

ERROR 5378: User Defined Scalar Function *string* provided precision *value*, larger than the maxi mum precision *value*

ERROR 5379: User Defined Scalar Function *string* provided range for Day To Second, but the func tion's return type is Interval Year To Month

ERROR 5380: User Defined Scalar Function *string* provided range for Year To Month, but the func tion's return type is Interval Day To Second

ERROR 5684: User Defined Extension cannot be created in a system schema

ERROR 6051: The schema has been dropped

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that

may help you resolve these errors.

Error Messages Associated with SQLSTATE VP001

This topic lists the error associated with the SQLSTATE VP001.

SQLSTATE VP001 Description

ERRCODE_USER_PROC_EXEC_ERROR

Error messages associated with this SQLState

```
ERROR 2376: Cannot connect to UDx side process (pid = value) during cancel: string

ERROR 2837: Could not create pipe for user procedure execution, errno=value

ERROR 2853: Could not execute user procedure: fork error

ERROR 2858: Could not find function definition

ERROR 2861: Could not find running procedure for string, proc ID=[value]

ERROR 3223: Error in calling string() for User Defined Function string at [string:value], erro
    r code: value, message: string

ERROR 3224: Error in calling string() for User Defined Scalar Function string at
    [string:value], error code: value, message: string

ERROR 3398: Failure in UDx RPC call string() (pid = value): string

ERROR 3399: Failure in UDx RPC call string(): string

ERROR 4424: Procedure execution error: exit status=value

ERROR 4425: Procedure execution error: procedure killed by signal (value)

ERROR 4538: Received message with unexpected type string

ERROR 5170: Unexpected exception from in calling string() for User Defined Scalar Function str
    ing

ERROR 5171: Unexpected exception in calling string() in  User Defined Function string

ERROR 5205: Unknown error killing procedure string

ERROR 5395: User procedure execution failed

ERROR 5398: User-defined Analytic Function string produced fewer output rows than input rows

ERROR 5399: User-defined Scalar Function string outputted a timezone (value) not in allowed ra
    nge (value, value)

ERROR 5400: User-defined Scalar Function string produced fewer output rows (value) than input
    rows (value)

ERROR 5430: Vertica process is not allowed to kill procedure string

ERROR 5580: Failure sending parameters block because the value parameters require value bytes,
    which exceeds the maximum size of value bytes

ERROR 5604: Invalid procedure file: [string]

ERROR 5638: Procedure file [string] cannot be owned by root

ERROR 5639: Procedure file [string] must be executable by vertica user (dbAdmin)

ERROR 5640: Procedure file [string] must be owned by specified procedure user
ERROR 5641: Procedure file [string] must enable set UID attribute

ERROR 5656: Root cannot execute external procedure

ERROR 5683: User 'string' not found on node

ERROR 5861: Error calling string() in User Function string at [string:value], error code: valu
    e, message: string

ERROR 5863: Error during setting up function, message: string

ERROR 6085: Unexpected exception calling string() User Function in string

ERROR 6086: Unexpected exception calling destroyUDxFenced()

ERROR 6087: Unexpected exception thrown by UDFileSystem at [string:value], error code: value,
    message: string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE VX001

This topic lists the error associated with the SQLSTATE VX001.

SQLSTATE VX001 Description

ERRCODE_INTERNAL_ERROR

Error messages associated with this SQLState

```
ERROR 2025: string Error occurred during BZIP initialization. BZIP error code: value

ERROR 2027: string Error occurred during ZLIB initialization. ZLIB error code: value, Message:
    string

ERROR 2405: Cannot do boundary analysis on type value

ERROR 2616: Cluster recovery failed, try again

ERROR 2928: Couldn't check this session's state

ERROR 3099: Did not find a variable

ERROR 3198: Error dropping table partition, data in WOS

ERROR 3211: Error during recovery running string queries, cannot continue: string

ERROR 3212: Error during recovery running string: string

ERROR 3220: Error generating query for: string

ERROR 3245: Error parsing string

ERROR 3257: Error retrieving string in string: string

ERROR 3292: Event apply failed

ERROR 3302: Exception decoding the response we just locally encoded

ERROR 3483: Got unexpected error code from spread: value, string

ERROR 3818: JOIN qualifications to not refer to the correct relation(s)

ERROR 3969: More than one variable found

ERROR 4372: pg_analyze_and_rewrite for View query failed
ERROR 4514: Raw parse of View query string failed

ERROR 4545: Recovery Error: Cannot get projections on local node

ERROR 5236: Unrecognized node type value

ERROR 5237: Unrecognized node type value in postprocess conditions

ERROR 5526: Already have a ready_recv string, ignoring

ERROR 5539: Cannot find buddy projection's statistics for collecting row counts, min and max

ERROR 5540: Cannot find buddy projections for collecting row counts, min and max

ERROR 5541: Cannot find the up-nodes of buddy projection for collecting row counts, min and ma
    x

ERROR 5679: Unrecognized order by expression

ERROR 5680: Unrecognized select column list

ERROR 5695: With query is not a Select Statement

ERROR 5719: Path Sampling failed. Try a different random seed for the pathSampling hint

ERROR 5802: Could not stop all dirty transactions

ERROR 5865: Error while analyzing approximate count distincts on table string.string

ERROR 6062: Too Many User defined types
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE VX002

This topic lists the error associated with the SQLSTATE VX002.

SQLSTATE VX002 Description

ERRCODE_DATA_CORRUPTED

Error messages associated with this SQLState

```
ERROR 2940: CRC Check Failure Details:
        File Name: string
        File Offset: value
        Compressed size in file: value
        Memory Address of Read Buffer: value
        Pointer to Compressed Data: value
        Memory Contents:
        string

ERROR 3030: Delete: could not find a data row to delete (data integrity violation?)

ERROR 3217: Error finalizing data file [string]

ERROR 3218: Error finalizing ROS DataTarget

ERROR 3219: Error flushing data file [string]

ERROR 3409: FileColumnReader: block string @ value 's CRC value doesn't match record value

ERROR 3410: FileColumnReader: Decompression error in string at offset value
ERROR 4762: Sort Order Violation:
        Row Position: value
        Column Index: value
        Last Row: string
        This Row: string
```

```
ERROR 5704: Delete (Join): could not find a data row to delete (data integrity violation?)
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

Error Messages Associated with SQLSTATE VX003

This topic lists the error associated with the SQLSTATE VX003.

SQLSTATE VX003 Description

ERRCODE_INDEX_CORRUPTED

Error messages associated with this SQLState

```
ERROR 3544: Index corruption. string: string
```

Note: The myVertica portal's Solutions tab contains helpful troubleshooting information that may help you resolve these errors.

We appreciate your feedback!

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on HP Vertica Programmer's Guide (Vertica Analytics Platform 7.0.x)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to vertica-docfeedback@hp.com.