Vertica® Analytic Database 5.0

Programmer's Guide

Date of Publication: June 20, 2011

CONFIDENTIAL

Contents

Technical Support

To submit problem reports, questions, comments, and suggestions, use the Technical Support page on the Vertica Web site.

Notes:

- You must be a registered user in order to access the ***MyVertica Portal http://myvertica.vertica.com/v-zone/overview***.
- If you are not a registered user, you can request access at the ***Technical Support Web page http://www.vertica.com/support***.

Before you report a problem, run the Diagnostics Utility described in the Troubleshooting Guide and attach the resulting `.zip` file to your ticket.

About the Documentation

This section describes how to access and print Vertica documentation. It also includes ***suggested reading paths*** (page 4).

Where to Find the Vertica Documentation

You can read or download the Vertica documentation for the current release of Vertica® Analytic Database from the ***Product Documentation Page http://www.vertica.com/v-zone/product_documentation***. You must be a registered user to access this page.

The documentation is available as a compressed tarball (`.tar`) or a zip archive (`.zip`) file. When you extract the file on the database server system or locally on the client, contents are placed in a `/vertica50_doc/` directory.

Notes:

- The documentation on the Vertica Web site is updated each time a new release is issued.
- A more recent version of the product documentation might be available online. To check for critical product or document information added after the product release, see the Vertica Product Documentation downloads site. You can download the PDF version or browse books online
- If you are using an older version of the software, refer to the documentation on your database server or client systems.

See Installing Vertica Documentation in the Installation Guide.

Reading the Online Documentation

Reading the HTML documentation files

The Vertica documentation files are provided in HTML browser format for platform independence. The HTML files require only a browser that displays frames properly with JavaScript enabled. The HTML files do not require a Web (HTTP) server.

The Vertica documentation is supported on the following browsers:

- Mozilla FireFox
- Internet Explorer
- Apple Safari
- Opera
- Google Chrome (server-side installations only)

The instructions that follow assume you have installed the documentation on a client or server machine.

Mozilla Firefox

1. Open a browser window.
2. Choose one of the following methods to access the documentation:
   - Select **File** > **Open File**, navigate to `..\HTML-WEBHELP\index.htm`, and click **Open**.
   - OR drag and drop `index.htm` into a browser window.
   - OR press **CTRL+O**, navigate to `index.htm`, and click **Open**.

Internet Explorer

Use one of the following methods:

1. Open a browser window.
2. Choose one of the following methods to access the documentation:
   - Select **File** > **Open** > **Browse**, navigate to `..\HTML-WEBHELP\index.htm`, click **Open**, and click **OK**.
   - OR drag and drop `index.htm` into the browser window.
   - OR press **CTRL+O**, Browse to the file, click **Open**, and click **OK**.

Note: If a message warns you that Internet Explorer has restricted the web page from running scripts or ActiveX controls, right-click anywhere within the message and select **Allow Blocked Content**.

Apple Safari

1. Open a browser window.
2. Choose one of the following methods to access the documentation:
   - Select **File** > **Open File**, navigate to `..\HTML-WEBHELP\index.htm`, and click Open.
   - OR drag and drop `index.htm` into the browser window.
   - OR press **CTRL+O**, navigate to `index.htm`, and click **Open**.

Opera

1. Open a browser window.
2. Position your cursor in the title bar and right click > **Customize** > **Appearance**, click the **Toolbar** tab and select **Main Bar**.
3. Choose one of the following methods to access the documentation:
   - Open a browser window and click Open, navigate to `..\HTML-WEBHELP\index.htm`, and click **Open**.
   - OR drag and drop `index.htm` into the browser window.
   - OR press **CTRL+O**, navigate to `index.htm`, and click **Open**.

Google Chrome

Google does not support access to client-side installations of the documentation. You'll have to point to the documentation installed on a server system.

1 Open a browser window.
2 Choose one of the following methods to access the documentation:

   In the address bar, type the location of the `index.htm` file on the server. For example: `file://<servername>//vertica50_doc//HTML/Master/index.htm`

   - OR drag and drop `index.htm` into the browser window.
   - OR press **CTRL+O**, navigate to `index.htm`, and click **Open**.

Notes

The `.tar` or `.zip` file you download contains a complete documentation set.

The documentation page of the ***Downloads Web site http://www.vertica.com/v-zone/download_vertica*** is updated as new versions of Vertica are released. When the version you download is no longer the most recent release, refer only to the documentation included in your RPM.

The Vertica documentation contains links to Web sites of other companies or organizations that Vertica does not own or control. If you find broken links, please let us know.

Report any script, image rendering, or text formatting problems to ***Technical Support*** (on page 1).

Printing Full Books

Vertica also publishes books as Adobe Acrobat™ PDF. The books are designed to be printed on standard 8½ x 11 paper using full duplex (two-sided) printing.

Note: Vertica manuals are topic driven and not meant to be read in a linear fashion. Therefore, the PDFs do not resemble the format of typical books.

Open and print the PDF documents using Acrobat Acrobat Reader. You can download the latest version of the free Reader from the ***Adobe Web site*** (***http://www.adobe.com/products/acrobat/readstep2.html***).

The following list provides links to the PDFs.

- Concepts Guide
- Installation Guide
- Getting Started Guide
- Administrator's Guide
- Programmer's Guide
- SQL Reference Manual
- Troubleshooting Guide

Suggested Reading Paths

This section provides a suggested reading path for various users. Vertica recommends that you read the manuals listed under All Users first.

All Users

- New Features — Release-specific information, including new features and behavior changes to the product and documentation
- Concepts Guide — Basic concepts critical to understanding Vertica
- Getting Started Guide — A tutorial that takes you through the process of configuring a Vertica database and running example queries
- Troubleshooting Guide — General troubleshooting information

System Administrators

- New Features — Release-specific information, including new features and behavior changes to the product and documentation
- Installation Guide — Platform configuration and software installation

Database Administrators

- Installation Guide — Platform configuration and software installation
- Administrator's Guide — Database configuration, loading, security, and maintenance

Application Developers

- Programmer's Guide — Connecting to a database, queries, transactions, and so on
- SQL Reference Manual — SQL and Vertica-specific language information

Where to Find Additional Information

Visit the ***Vertica Web site*** (***http://www.vertica.com***) to keep up to date with:

- Downloads
- Frequently Asked Questions (FAQs)
- Discussion forums
- News, tips, and techniques
- Training

Typographical Conventions

The following are the typographical and syntax conventions used in the Vertica documentation.

Typographical Convention	Description
Bold	Indicates areas of emphasis, such as a special menu command.
Button	Indicates the word is a button on the window or screen.
`Code`	SQL and program code displays in a monospaced (fixed-width) font.
`Database objects`	Names of database objects, such as tables, are shown in san-serif type.
Emphasis	Indicates emphasis and the titles of other documents or system files.
`monospace`	Indicates literal interactive or programmatic input/output.
`monospace italics`	Indicates user-supplied information in interactive or programmatic input/output.
`UPPERCASE`	Indicates the name of a SQL command or keyword. SQL keywords are case insensitive; `SELECT` is the same as `Select`, which is the same as `select`.
`User input`	Text entered by the user is shown in bold san serif type.
↵	indicates the Return/Enter key; implicit on all user input that includes text
Right-angle bracket >	Indicates a flow of events, usually from a drop-down menu.
Click	Indicates that the reader clicks options, such as menu command buttons, radio buttons, and mouse selections; for example, "Click OK to proceed."
Press	Indicates that the reader perform some action on the keyboard; for example, "Press Enter."

Syntax Convention	Description
`Text without brackets/braces`	Indicates content you type as shown.
`<` *`Text inside angle brackets`* `>`	Placeholder for which you must supply a value. The variable is usually shown in italics. See Placeholders below.
`[ Text inside brackets ]`	Indicates optional items; for example, CREATE TABLE [*schema_name.*]*table_name* The brackets indicate that the *schema_name* is optional. Do not type the square brackets.
`{ Text inside braces }`	Indicates a set of options from which you choose one; for example: `QUOTES { ON | OFF }` indicates that exactly one of ON or OFF must

	be provided.You do not type the braces: `QUOTES ON`
Backslash `\`	Continuation character used to indicate text that is too long to fit on a single line.
Ellipses `...`	Indicate a repetition of the previous parameter. For example, `option[,...]` means that you can enter multiple, comma-separated options. Note: Showing an ellipses in code examples might also mean that part of the text has been omitted for readability, such as in multi-row result sets.
`Indentation`	Is an attempt to maximize readability; SQL is a free-form language.
`Placeholders`	Items that must be replaced with appropriate identifiers or expressions are shown in italics.
Vertical bar `|`	Is a separator for mutually exclusive items. For example: `[ASC | DESC]` Choose one or neither. You do not type the square brackets.

Preface

This book describes how to connect to a Vertica database and run SQL statements.

Audience

This book is intended for anyone who retrieves information from a Vertica database. It assumes that you are familiar with the basic concepts and terminology of the SQL language and relational database management systems.

As a Vertica SQL programmer, most of your tasks are similar to those required by other relational database management systems.

Prerequisites

This document assumes that you have installed and configured Vertica as described in the Installation Guide.

Writing Queries

Vertica is designed to run queries that are suitable for a star schema or snowflake schema. You might need to modify existing normalized schema queries to run them against a Vertica database.

For information about the SQL language, see the SQL Reference Manual.

Installing the Vertica Client Drivers

Before you can access your Vertica database from a client, you need to install client drivers. These drivers create and maintain connections to the database and provide APIs that your applications use to access your data. These drivers support connections using JDBC, ODBC, and ***ADO.NET*** (page 109).

In addition to the client drivers, there are language-specific interfaces for Perl and Python. See ***Using Perl*** (page 130) and ***Using Python*** (page 125) for details.

Client Driver Standards

The client drivers support the following standards:

- ***ODBC*** (page 26) drivers conform to ODBC 3.5.1 specifications.
- ***JDBC*** (page 66) drivers conform to JDK 5 specifications.
- ***ADO.NET*** (page 109) drivers conform to .NET framework 3.0 specifications.

About Client Drivers

Vertica supplies drivers for Windows, Linux, and Solaris clients. There are several different driver packages available from the ***Vertica download page http://www.vertica.com/v-zone/download_vertica***, each supporting a different operating system and system architecture:

- Complete client bundle for Windows 32-bit containing an InstallShield Wizard that installs the ODBC, JDBC, and ADO.NET drivers, plus a Visual Studio 2008 plug-in

  Note: The Visual Studio plug-in requires that the Visual Studio SDK be installed on the system. The plug-in is available at the ***Microsoft Download Center http://www.microsoft.com/downloads/details.aspx?FamilyID=30402623-93ca-479a-867c-04dc45164f5b&displaylang=en***.
- Complete client bundle for Windows 64-bit systems containing an InstallShield Wizard that installs the ODBC, JDBC, and ADO.NET drivers.
- Complete client bundle for Red Hat Enterprise Linux 32-bit and 64-bit that contains the ODBC and JDBC drivers as well as the vsql executable.
- Complete client bundle for SUSE Enterprise Linux 32-bit and 64-bit that contains the ODBC and JDBC drivers as well as the vsql executable.
- Individual packages for Linux 32-bit and 64-bit ODBC drivers.
- Individual packages for Solaris x86 and SPARC ODBC drivers.
- Individual packages for AIX 5.3 ODBC 32-bit and 64-bit drivers.
- A cross-platform .jar file containing the JDBC driver.

Note: The ODBC and JDBC client drivers are installed by the server `.rpm` files. If you have installed Vertica® Analytic Database 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 and Solaris Clients*** (page 27)). To use JDBC, you need to add the JDBC client driver to the Java CLASSPATH (see ***Modifying the CLASSPATH*** (page 24)).

Vertica drivers use a naming convention that reflects the version of the Vertica release. This is because the client driver version must match the version of the server on which the database runs. For example, all Vertica client drivers 3.0 require Vertica® Analytic Database server version 3.0 or later. If you are using a version of Vertica earlier than 5.0, you should download and install the drivers for your version of Vertica® Analytic Database. There is a link for earlier server and driver versions on the ***Vertica download page http://www.vertica.com/v-zone/download_vertica***.

Note: Installing new drivers does not alter existing DSN settings.

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

Driver Prerequisites

It is important that you read this section before you install a driver on the client machine.

Supported Third-party Software

The following table lists commonly-used Vertica-supported third-party software and the driver managers they use. For a full list of supported third-party software, refer to the ***Third Party Tools http://www.vertica.com/v-zone/downloads/client-tools/third-party-tools*** tab on the ***Vertica Web site download http://myvertica.vertica.com/v-zone/download_vertica*** page.

3rd-party Tool	Platform	Driver Manager	32 bit	64 bit
MicroStrategy	Linux Red Hat Enterprise 5	DataDirect Connect®	Yes	No
	Linux SUSE Enterprise 10	DataDirect Connect®	Yes	No
	Sparc Solaris 10	DataDirect Connect®	Yes	No
	Windows	Microsoft ODBC MDAC	Yes	No
Informatica 8.6.1	Linux Red Hat Enterprise 5	DataDirect Connect®	Yes	No
	Linux SUSE Enterprise 10	DataDirect Connect®	Yes	No
	Sparc Solaris 10	DataDirect Connect®	Yes	Yes
Informatica 9.0.1	Linux Red Hat Enterprise 5	DataDirect Connect®	Yes	No
	Linux SUSE Enterprise 10	DataDirect Connect®	Yes	No
	Sparc Solaris 10	DataDirect Connect®	No	Yes
Cognos	Linux Red Hat Enterprise 5	unixODBC	Yes	No
	Linux SUSE Enterprise 10	unixODBC	Yes	No

Sparc Solaris 10	iODBC	Yes	No
Windows	Microsoft ODBC MDAC	Yes	No

Note: In addition to using Informatica with the ODBC driver, you can also use the Vertica Plug-in for PowerCenter to use Vertica as a target for Informatica PowerCenter. See ***Using Informatica PowerCenter*** (page 347) for details.

See Also

The ***Vertica Web site download http://myvertica.vertica.com/v-zone/download_vertica*** page for supported third-party tools.

The ***Cognos Web site http://www.cognos.com/*** for more specific requirements about supported client interfaces and platforms.

ODBC Prerequisites

The Vertica driver for ODBC requires the software and hardware components listed in this section.

Operating System

The Vertica ODBC driver requires one of the following operating systems:

- AIX 5.3 (32-bit or 64-bit)
- Linux Red Hat Enterprise 5 (32 or 64 bit)
- Linux SUSE Enterprise 10 (32 or 64 bit)
- SPARC Solaris 10 (32 bit or 64 bit)
- Windows XP Professional
- Windows 2003 Server Standard Edition (32 or 64 bit)
- Windows 2003 Server Enterprise Edition (32 or 64 bit)
- Windows 2008 Server Standard Edition (32 or 64 bit)
- Windows 2008 Server Enterprise Edition (32 or 64 bit)

See also Supported Platforms.

ODBC Driver Manager

The Vertica ODBC driver requires one of the driver managers in the following table. The driver only works when used with a driver manager—you cannot directly link your application to the Vertica ODBC driver. On Windows, the driver manager is part of the MDAC component. For ODBC Driver Managers for AIX, Linux, or Solaris, see ***Installing AIX, Linux, and Solaris Driver Managers*** (page 16).

Platform	Driver Manager	32 bit	64 bit
AIX	unixODBC 2.2.12	Yes	Yes
Linux	unixODBC 2.2.11 or 2.2.12	Yes	Yes

	unixODBC 2.2.14	Yes	Yes (see note)
	iODBC 3.52.6	Yes	Yes
	DataDirect Connect® 5.3	Yes	No
	DataDirect Connect® 6.0	Yes	No
SPARC Solaris 10	unixODBC 2.2.12	Yes	No
	iODBC 3.52.6	Yes	No
	DataDirect Connect® 5.3	Yes	Yes
	DataDirect Connect® 6.0	Yes	Yes
x86 Solaris 10	unixODBC 2.2.12	Yes	No
	iODBC 3.52.6	Yes	No
	DataDirect Connect® 5.3	Yes	No
	DataDirect Connect® 6.0	Yes	No
SPARC Solaris 10	unixODBC 2.2.12	Yes	No
	iODBC 3.52.6	Yes	No
	DataDirect Connect® 5.3	Yes	Yes
	DataDirect Connect® 6.0	Yes	Yes
x86 Solaris 10	unixODBC 2.2.12	Yes	No
	iODBC 3.52.6	Yes	No
	DataDirect Connect® 5.3	Yes	No
	DataDirect Connect® 6.0	Yes	No
Windows XP, 2003, and 2008	Microsoft ODBC MDAC 2.8	Yes	Yes

Note: unixODBC 2.2.14 and above are only supported if they are compiled with `BUILD_LEGACY_64_BIT_MODE` enabled, to ensure sizeof(SQLLEN) is 4 bytes rather than 8 bytes. See ***Installing Linux and Solaris Driver Managers*** (page 16) for details.

DataDirect is certified only with specific tools that ship with the Data Direct driver manager. Vertica does not ship the Data Direct Driver manager.

See Also

Client Driver Install Procedures (page 16)

Using ODBC (page 26)

Creating an ODBC Data Source Name (DSN) (page 27)

ADO.NET Prerequisites

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

Operating System

The Vertica ADO.NET driver requires one of the following operating systems:

- Windows XP Professional
- Windows 2003 Server Standard Edition (32 or 64 bit)
- Windows 2003 Server Enterprise Edition (32 or 64 bit)
- Windows 2008 Server Standard Edition (32 or 64 bit)
- Windows 2008 Server Enterprise Edition (32 or 64 bit)

Visual Studio SDK (32-bit installs only)

The Visual Studio plug-in is automatically installed with the client driver. If you intend to use it, install the Visual Studio SDK prior to installing the client driver. The plug-in is available at the ***Microsoft Download Center http://www.microsoft.com/downloads/details.aspx?FamilyID=30402623-93ca-479a-867c-04dc45164f5b&displaylang=en***.

Memory

Vertica suggests a minimum of 512MB of RAM. If you intend to use the ***buffered data reader*** (page 123), you might require additional RAM.

Disk Space

If you intend to use the buffered data reader, be sure the system has enough disk space to support large streaming result sets. The space required for a streaming result set is temporary and is immediately released when the application that is using the result set is closed.

.NET Framework

The Vertica ADO.NET driver integrates with any of the following versions of .NET Framework:

- Microsoft .NET Framework 3.0 SP1 (minimum)
- Microsoft .NET Framework 3.5
- Microsoft .NET Framework 3.5 SP1

Note: The Vertica ADO.NET driver does not support later APIs provided with Microsoft .NET Framework 3.5 and 3.5 (SP1). For example, it does not support ADO.NET synchronization or paging.

See Also

Client Driver Install Procedures (page 16)

Using ADO.NET (page 109)

Python Prerequisites

Python is a free, agile, object-oriented, cross-platform programming language designed to emphasize rapid development and code readability.

Vertica supports the following Python versions:

- 2.4.6
- 2.5.4
- 2.6.2

Note: Vertica does not support Python version 3.x.

Python Driver

Vertica requires the pyodbc driver module version 2.1.6.

Supported Operating Systems

The Vertica ODBC driver requires one of the operating systems listed in ***ODBC Prerequisites*** (page 12).

ODBC Driver Manager

- On Linux — unixODBC or iODBC
- On Windows — Microsoft ODBC MDAC

See ***ODBC Prerequisites*** (page 12) for currently supported versions.

For usage and examples, see ***Using Python*** (page 125).

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).

Vertica supports the following Perl versions:

- 5.8
- 5.10

Perl Drivers

The following Perl driver modules are required:

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

Supported Operating Systems

The Vertica ODBC driver requires one of the operating systems listed in ***ODBC Prerequisites*** (page 12).

ODBC Driver Manager

- On Linux — unixODBC or iODBC
- On Windows — Microsoft ODBC MDAC

See ***ODBC Prerequisites*** (page 12) for currently supported versions.

For usage and examples, see ***Using Perl*** (page 130).

Client Driver Install Procedures

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

- For Linux clients, you must first ***install a Linux driver manager*** (page 16). 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 Vertica client RPM package to install the ODBC and JDBC drivers as well as the vsql client.
  - On other Linux platforms, you download the proper 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 Vertica® Analytic Database 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 and Solaris Clients*** (page 27)). To use JDBC, you need to add the JDBC client driver to the Java CLASSPATH (see ***Modifying the CLASSPATH*** (page 24)).
- On Solaris clients, you download and install the ODBC and JDBC drivers individually.
- On AIX clients, you download and install the ODBC and JDBC drivers individually.
- On Windows clients, you download an installer that contains the ODBC, ADO.NET, and JDBC drivers. There are separate installers for 32-bit and 64-bit clients.

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

Installing AIX, Linux, and Solaris Driver Managers

UnixODBC

Versions 2.2.11 and 2.2.12 of the UnixODBC driver managers are supported by Vertica in their default configurations. These versions are pre-installed on many Linux and Solaris installations. If they are not already installed, see if binary packages are available through your platform's package management system. Consult your platform's documentation for details on locating and installing packages.

For 64-bit Linux installations, Vertica requires that the UnixODBC driver version 2.2.14 be compiled using the `BUILD_LEGACY_64_BIT_MODE` option, which sets SQLLEN to 4 bytes, instead of the default 8 bytes. To be compatible with Vertica, you will need to recompile the UnixODBC 2.2.14 driver manager by following these steps:

1. If a binary UnixODBC 2.2.14 package is installed on your system, uninstall it using your distribution's package manager.
2. Download the UnixODBC 2.2.14 source from the the following link:

   http://sourceforge.net/projects/unixodbc/files/unixODBC/2.2.14/unixODBC-2.2.14.tar.gz/download

   http://sourceforge.net/projects/unixodbc/files/unixODBC/2.2.14/unixODBC-2.2.14.tar.gz/download

   (Alternately, you can see if your platform offers a source package for UnixODBC 2.2.14.)

3 As the root user, build UnixODBC-2.2.14 with -DBUILD_LEGACY_64_BIT_MODE and install it:

```
$ tar -xvzf unixODBC-2.2.14.tar.gz
$ cd unixODBC-2.2.14
$ export CPPFLAGS="-DBUILD_LEGACY_64_BIT_MODE -DSIZEOF_LONG_INT=8"
$ ./configure --enable-gui=no --enable-drivers=no
$ make
$ make install
```

Note: Compiling packages requires your platform to have compilers and development libraries installed. See your Linux or Solaris documentation for details.

iODBC

Download a package for iODBC 3.52.6 suitable to your platform from the ***iODBC.org http://www.iodbc.org/dataspace/iodbc/wiki/iODBC/*** Web site.

Installing the Client RPM on Red Hat 5 64-bit, and SUSE 64-bit

For Red Hat Enterprise Linux 5, 64-bit and SUSE Linux Enterprise Server 10/11 64-bit , you can download and install a client RPM package that installs both the ODBC and JDBC driver and the vsql client. There is one RPM for Red Hat (which also works on CentOS 5 64-bit) and another for SUSE.

To install the RPM package:

1 Open a browser and log in to the Vertica ***download Web site http://www.vertica.com/v-zone/download_vertica***.
2 Scroll to the Drivers for Vertica® Analytic Database 5.0 section.
3 Locate the heading for the client RPM package, and click **Download** next to the entry for your client's version of Linux.
4 Read the Agreement License and click **I Agree.**
5 When the download window loads, click **Save File**.
6 If you did not directly download to the client system, transfer the downloaded RPM file to it.
7 Log in to the client system as root.
8 Install the RPM package you downloaded:

```
# rpm -Uvh vertica-client-5.0.x86 64.platform.rpm
```

Once you have installed the client package, you need to ***create an ODBC DSN*** (page 27) to use ODBC, and ***change the Java CLASSPATH*** (page 24) 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 the full path to run it. You add it to your path by adding the following to your `.profile` file:

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

Installing ODBC on AIX, Linux, and Solaris

Read ***Driver Prerequisites*** (page 11) before you proceed.

Note: The ODBC and JDBC client drivers are installed by the server `.rpm` files. If you have installed Vertica® Analytic Database 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 and Solaris Clients*** (page 27)). To use JDBC, you need to add the JDBC client driver to the Java CLASSPATH (see ***Modifying the CLASSPATH*** (page 24)).

A new ODBC driver requires a new Data Source Name. You can decide when to use the new driver, by creating a new DSN, or by eliminating the old driver and creating a new one that uses the old name. See ***Creating an ODBC Data Source Name (DSN)*** (page 27) for details.

The download file for AIX, Linux, and Solaris operating systems includes the driver manager.

The list of downloads on the Vertica download website for Linux and Solaris clients are broken down by driver manager. Within each driver manager section are links for each Linux and Solaris architecture (for example, 64-bit Linux). The downloaded file is named based on its operating system, driver manager, and architecture (for example, `vertica_5.0.`*`xx`*`_unixodbc_x86_64_linux.tar.gz`)

Installation Procedure

1. Open a browser and log in to the Vertica ***download Web site http://www.vertica.com/v-zone/download_vertica***.
2. Scroll to the Drivers for Vertica® Analytic Database 5.0 section.
3. Within the heading for your driver manager (for example, **For unixODBC Driver Manager on Linux and Solaris**), click the link for your operating system and architecture. For example, **ODBC Driver, 64-bit Linux**.
4. Read the Agreement License and click **I Agree.**
5. When the download window loads, click **Save File**.
6. If you did not directly download to the client system, transfer the downloaded file to it.
7. Log in to the client system as root.
8. If the directory `/opt/vertica/` does not exist, create it:

   `# mkdir -p` `/opt/vertica/`
9. Copy the downloaded file to the `/opt/vertica/` directory. For example:

   `#` **`cp vertica_5.0.`***`xx`***`_unixodbc_x86_64_linux.tar.gz`**
10. Change to the `/opt/vertica/` directory:

    `#` **`cd`** `/opt/vertica/`
11. Uncompress the file you downloaded. For example:

    `$` **`tar vzxf vertica_5.0.`***`XX`***`_unixodbc_x86_64_linux.tar.gz`**

    Two folders will be created: one for the include file, and one for the library file. 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 library client download would create the directories:

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

Pointing to the ODBC Driver Configuration File

In a bash shell, where you will be running your application, type the following command (assuming `/etc/odbc.ini` is the location of your `odbc.ini` file):

```
$ export ODBCINI=/etc/odbc.ini
```

The following is a sample `odbc.ini` file. See also ***Creating an ODBC DSN for Linux and Solaris Clients*** (page 27).

```
[VerticaDSN]
Description = VerticaDSN ODBC driver
Driver = /opt/vertica/lib64/libverticaodbc_unixodbc.so
Database = vmartdb
Servername = host01
UserName = dbadmin
Password =
Port = 5433
[ODBC]
```

Installing JDBC Driver on Linux and Solaris

Note: The ODBC and JDBC client drivers are installed by the server `.rpm` files. If you have installed Vertica® Analytic Database 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 and Solaris Clients*** (page 27)). To use JDBC, you need to add the JDBC client driver to the Java CLASSPATH (see ***Modifying the CLASSPATH*** (page 24)).

The JDBC driver is available for download from http://myvertica.vertica.com/v-zone/download_vertica ***http://myvertica.vertica.com/v-zone/download_vertica***. There is a single `.jar` file that works on all platforms and architectures. To download and install the file:

1. Log into the Vertica Web site's download page: http://myvertica.vertica.com/v-zone/download_vertica ***http://myvertica.vertica.com/v-zone/download_vertica***.
2. Under the **Drivers for Vertica® Analytic Database 5.0** section, locate the **JDBC driver, 32/64 bit (all platforms)** entry and click **Download**.
3. Click **I Agree** to agree to the license agreement.
4. When prompted by your browser, save the `vertica_5.0.`*xx*`_jdk_5.jar` file to a location on your computer.
5. You need to copy the `.jar` file you downloaded file to a directory in your Java ***CLASSPATH http://en.wikipedia.org/wiki/Classpath_%28Java%29*** on every client system with which you want to access 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 CLASSPATH*** (page 24) 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 ***client RPM for Red Hat and SUSE*** (page 17) create this link when they install the JDBC library.

Installing ODBC, JDBC, and ADO.NET Drivers on Windows

This section contains procedures for both 32- and 64-bit Windows operating systems.

IMPORTANT

When enabled, virus scanners and the User Account Control (UAC) can interfere with the installation of Vertica's client drivers. If you have an issue installing the Vertica driver package, follow these steps:

1. Temporarily disable any virus scanner installed on your system. See your virus scanner's documentation for details.
2. ***Temporarily disable the UAC http://windows.microsoft.com/en-US/windows-vista/Turn-User-Account-Control-on-or-off***.
3. Download the Vertica Windows driver package for your platform and install it, following the instructions in this section.
4. Re-enable the UAC and virus scanner.

There are two Windows driver packages on the Vertica web site: one for 32-bit clients and another for 64-bit. They are clearly labeled, making it easy for you to select the correct one for your platform.

The Vertica InstallShield Wizard installs the following drivers:

- On Windows 32-bit systems: ODBC, JDBC, and ADO.NET drivers, plus a Visual Studio 2008 plug-in.

  Note: The Visual Studio plug-in requires that the Visual Studio SDK be installed on the system. The plug-in is available at the ***Microsoft Download Center http://www.microsoft.com/downloads/details.aspx?FamilyID=30402623-93ca-479a-867c-04dc45164f5b&displaylang=en***
- On Windows 64-bit systems: ODBC, JDBC, and ADO.NET drivers.

Note: Use the same InstallShield Wizard to repair, modify, and remove installed drivers on Windows clients. Note that the uninstall option works only for Vertica drivers 2.5 and later that were installed with the InstallShield application. If you want to remove a Vertica driver that was installed before 2.5, use the Add/Remove Programs in the Windows Control Panel.

Installing Drivers on 32-bit Windows

The following procedure installs ODBC, JDBC, and ADO.NET drivers and the Visual Studio 2008 plug-in to the 32-bit client.

Note: The Visual Studio plug-in requires that the Visual Studio SDK be installed on the system. The plug-in is available at the ***Microsoft Download Center http://www.microsoft.com/downloads/details.aspx?FamilyID=30402623-93ca-479a-867c-04dc45164f5b&displaylang=en***.

Read ***Driver Prerequisites*** (page 11) before you proceed.

Installation Procedure

1. Temporarily disable any virus scanner or User Account Control (UAC) on the client, either of which can interfere with the installation of the Vertica driver.
2. Open a browser and log in to the Vertica ***download Web site http://www.vertica.com/v-zone/download_vertica***.
3. Scroll to the portion of the page labeled Drivers for Vertica® Analytic Database 5.0.
4. Under the Windows section, click **Download** next to the entry for the 32-bit client drivers bundle.
5. Read the Agreement License and click **I Agree**.
6. When the download window opens, click **Save File**, and the driver is saved to the default download location on the client machine.
7. Double-click the saved download and click **Next** after the InstallShield Wizard launches.
8. Click **Next** to begin the installation.
9. Read the license agreement (optionally clicking **Print** to print a copy of the agreement), select **I accept the terms of the license agreement**, and click **Next**.
10. Select Complete or Custom and click **Next**.
    - Complete — Installs ODBC, JDBC, and ADO.NET drivers and the Visual Studio 2008 plug-in to `C:\Program Files\Vertica Systems\Vertica Client Drivers 5.0`.
    - Custom — Lets you choose drivers and the plug-in. You can also specify a different installation path from the default.
11. Click **Install** and the Wizard copies the Vertica drivers to the client machine.

    Once the installation is complete, you are given the opportunity to view the Readme document and visit www.vertica.com. If you want to read the file now, click **View the Readme document**. Alternatively, you can read ODBC, JDBC, and ADO.NET documentation at `C:\Program Files\Vertica Systems\Vertica Client Drivers 5.0`.
12. Click **Finish** to exit the installation wizard.
13. Re-enable virus scanner or UAC that you disabled earlier.

Post Installation

Do one of the following:

- If you use ODBC, ***create a new Data Source Name*** (page 27) (DSN) to use the new driver.
- If you use JDBC, ***modify the CLASSPATH*** (page 24) to use the new driver.
- There are no post-installation requirements for ADO-NET users.

ADO.NET (page 109) users can run the nvsql command to connect to a database, which is similar to vsql, but with less functionality.

1. Open a command prompt
2. Change directories to the bin folder
   ```
   cd C:\Program Files\Vertica Systems\Vertica Client Drivers 5.0\bin
   ```
3. Specify a host, port, database, and user:
   ```
   nvsql 10.10.10.10:5433 DATABASENAME username
   ```
4. Run a simple query:
   ```
   nvsql> SELECT NOW();
   ```

Installing Drivers on 64-bit Windows

The following procedure installs ODBC, JDBC, and ADO.NET drivers to the 64-bit client.

Read ***Driver Prerequisites*** (page 11) before you proceed.

Installation Procedure

1. Log in to Windows client as Administrator.
2. Temporarily disable any virus scanner or User Account Control (UAC), either of which can interfere with installing the Vertica drivers.
3. Open a browser and log in to the Vertica ***download Web site http://www.vertica.com/v-zone/download_vertica***.
4. Scroll to the Drivers for Vertica® Analytic Database 5.0 portion of the page.
5. Under Windows, click the **Download** button next to the 64-bit client drivers entry.
6. Read the Agreement License and click **I Agree**.
7. When the download window appears, click **Save File** to save the driver package to a location on the client system.
8. Double-click the downloaded install package to start the install process.
9. Read the license agreement (and optionally click **Print** to print a copy), then click **Yes** to accept the agreement.
10. Click **Next** to begin the installation.
11. Change the user name if you wish, and type your company's name in the **Company Name** box.
12. Select whether you want the installation to be available to all users or just your account, then click **Next.**
13. Select the type of setup and click **Next**.
    - Typical—Installs ODBC, JDBC, and ADO.NET drivers to `C:\Program Files\Vertica Systems\Vertica Client Drivers 5.0`.
    - Compact—Installs the minimum required options: ODBC, JDBC, and ADO.NET.
    - Custom — Lets you specify a destination folder.
14. Click **Next** again to begin the installation.
15. Click **Finish** to exit the installation wizard.
16. Re-enable any virus scanner and UAC you disabled earlier.

Post Installation

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

- For ODBC, ***create a new Data Source Name*** (page 27) (DSN).
- For JDBC, ***modify the CLASSPATH*** (page 24).
- For ADO.NET, there isn't a post-install step.

Modifying the CLASSPATH

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

Note: You should use the symbolic link that points to the JDBC library `.jar` file, rather than the `.jar` file itself in your CLASSPATH. 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/UNIX

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

```
# export CLASSPATH=/opt/vertica/java/lib/vertica_5.0_jdk_5.jar
```

Caution: 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_5.0_jdk_5.jar
```

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

You will need to either set the CLASSPATH environment variable for every login session, or place the command to set the variable into one of your startup scripts (such as `.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_5.0.xx_jdk_5.jar
```

As with the Linux/UNIX settings, this setting only lasts for the current session. To set the CLASSPATH permanently, you can 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 Vertica JDBC `.jar` file on your system (for example, `C:\Program Files\Vertica Systems\Vertica Client Drivers 5.0\lib\vertica_5.0.xx_jdk5.jar`)

Specifying the Library Directory in the Java Command

There is an alternative way to tell the Java runtime where to find the 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 you could start your client application using:

```
# java -classpath /opt/vertica/java/lib/vertica_5.0_jdk_5.jar myapplication.class
```

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

Using ODBC

Vertica provides the ODBC driver so applications can connect to the Vertica database. This Unicode 3.51 driver allows all string input and output to be presented in Unicode. This means that SQL queries can be run in Unicode and data can be returned from Vertica in Unicode.

This section details the process for configuring the Vertica ODBC driver. It also demonstrates options for using the ODBC driver to connect to Vertica programmatically and assumes you have already installed the ODBC driver. If you have not, see:

- ***Installing Client Drivers on AIX, Linux, and Solaris*** (page 17)
- ***Installing Client Drivers on Windows*** (page 20)

Note: If using DataDirect® driver manager, you should always use the SQL_DRIVER_NOPROMPT option when connecting to Vertica, as Vertica's ODBC driver on UNIX platforms doesn't contain a UI with which it can prompt you for a password.

ODBC Architecture

The ODBC architecture has four components:

- **Client Application**

  Is an application, which is written in C, that interacts with a database by opening a data source through a DSN reference, sending requests to the data source, and processing these results. Requests are made in the form of calls to ODBC functions, which submit these requests as SQL statements.
- **Driver Manager**

  Is a library that acts as an intermediary between a client application and one or more drivers. It is responsible for:
  - Resolving the Data Source Name (DSN) provided by the client application.
  - Loading the driver required to access the specific database defined within the DSN.
  - Processing ODBC function calls from the client or passing them to the driver.
  - Performing function call sequence checks.
  - Tracing each application call and its results.
  - Unloading drivers when they are no longer needed.

  See ***ODBC Prerequisites*** (page 12) for a list of driver managers that can be used with Vertica.
- **Driver**

  Is as a shared object (under Linux or UNIX) or a DLL (under Widows) that provides access to a specific database, for example Vertica. It translates incoming and outgoing information as follows: ODBC requests are translated into the format expected by the database, and database-specific results are translated back into ODBC for the client application.
- **Database**

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

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 drive and other information that is required to access data from a data source. Once you have installed the ODBC driver, you need to configure and test a DSN. The method you use depends upon the type of client operating system you're using:

- ***Creating an ODBC DSN for Linux and Solaris Clients*** (page 27)
- ***Creating an ODBC DSN for Windows Clients*** (page 29)

Creating an ODBC DSN for Linux and Solaris Clients

Creating a DSN for a Linux or Solaris client machine entails configuring the following files and then testing the configuration:

- `/etc/odbc.ini`
- `/etc/odbcinst.ini`

Configuring the *odbc.ini* file:

On Linux and Solaris, ODBC data sources reside in a file named `odbc.ini`.

1. Using the text editor of your choice, open `odbc.ini`.
2. Create an ODBC Data Sources section and enter the VerticaDSN parameter.

   This parameter establishes the name by which the new data source is referred. There is no special significance to the default name. For example:

   ```
   [ODBC Data Sources]
   VerticaDSN = "vmartdb"
   ```
3. Create a VerticaDSN section in which to establish the parameters for the DSN. The example below this list creates the following parameters:
   - **Description** – Additional information about the data source.
   - **Driver** – The location and designation of the Vertica ODBC driver. 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 using the unixODBC driver manager is:

     `/opt/vertica/lib64/libverticaodbc_unixodbc.so`

     The symbolic link always points to the most up-to-date version of the 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 Vertica is installed. Use localhost if Vertica is installed on the same machine.
   - **UserName** – 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.

- **Password** – The password for the specified user name. This example leaves the password field blank.
- **Port** – The port number on which 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:

```
[VerticaDSN]
Description = VerticaDSN ODBC driver
Driver = /opt/vertica/lib64/libverticaodbc_iodbc.so
Database = vmartdb
Servername = host01
UserName = dbadmin
Password =
Port = 5433
ConnSettings =
SSLKeyFile = /home/dbadmin/client.key
SSLCertFile = /home/dbadmin/client.crt
Locale = en_GB
```

See ***DSN parameters*** (page 38) for a complete list of parameters including Vertica-specific ones.

Configuring the odbcinst.ini File

Create a VerticaDSN section and enter the following parameters:

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

For example:

```
[VerticaDSN]
Description = VMart example database
Driver = /opt/vertica/lib/libverticaodbc_unixodbc.so
```

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
```

Testing the Configuration

unixODBC comes with a variety of tools that allow you to test the connection. These instructions describe how to use the command line tool isql. The isql tool allows you to connect to the DSN to send commands and receive results.

To use isql to test the DSN connection:

1. Run the following command:

   ```
   $ isql –v VerticaDSN
   SQL>
   ```

   A connection message and a SQL prompt display. If does 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 [columname] FROM [tablename];
   ```

   The isql tool returns the results of your SQL statement.

Creating an ODBC DSN for Windows Clients

Creating a DSN for Microsoft Windows clients consists of:

- ***Setting up a DSN*** (page 29)
- ***Testing the DSN using Excel 2003*** (page 32) or ***Excel 2007*** (page 35)
- ***Creating User and System DSN Entries*** (page 37)

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 Vertica ODBC Driver to set up an ODBC DSN. This topic assumes that the driver is already installed, as described in ***Installing ODBC, JDBC, and ADO.NET on Windows*** (page 20).

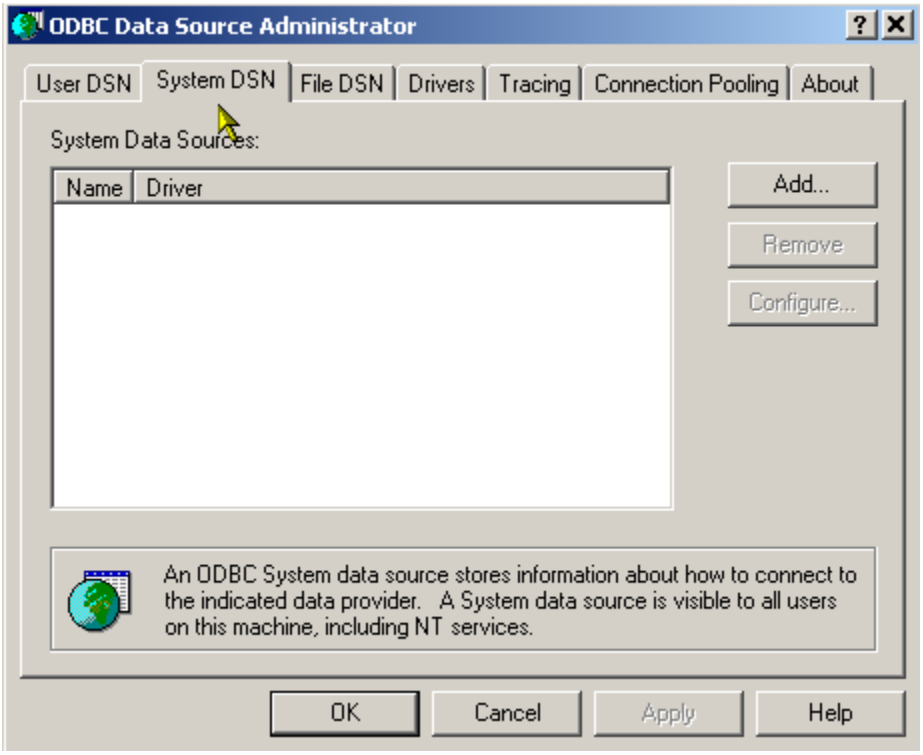
To set up a DSN:

1. From the Windows Control Panel, open the ODBC Administrator.

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

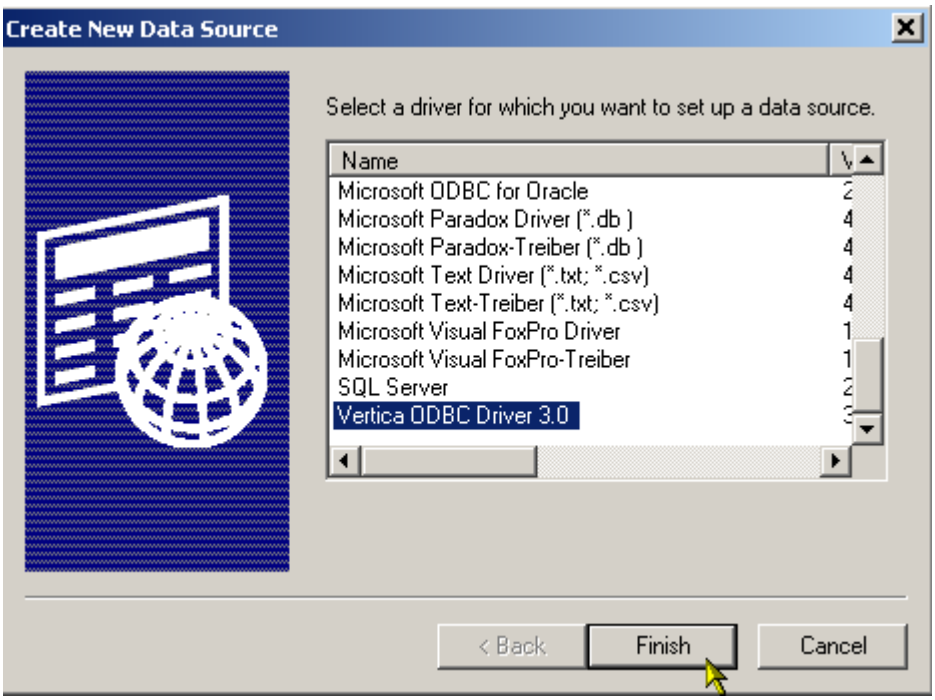
   - Start > Control Panel > Data Sources (ODBC).
   - Start > Control Panel > Administrative Tools > Data Sources (ODBC).
2. In the ODBC Data Source Administrator, click the **System DSN** tab.

This allows all users on the system to use this DSN. If you click the User DSN, only the user creating the DSN Entry can access it.

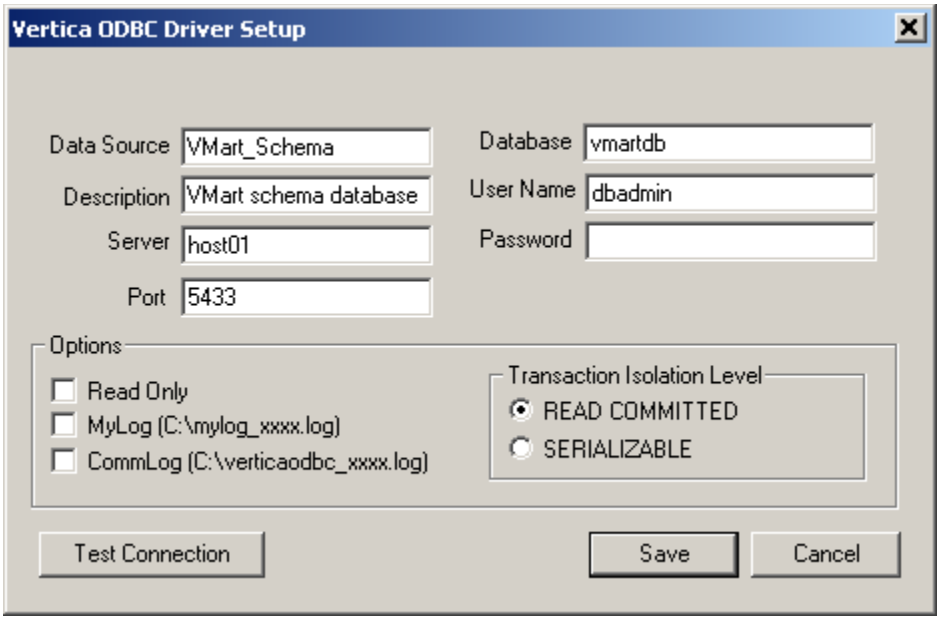


3 Click **Add** to create a system-wide data source name for the Vertica driver.

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



5 Enter your data source information in the Vertica ODBC Driver Setup dialog.

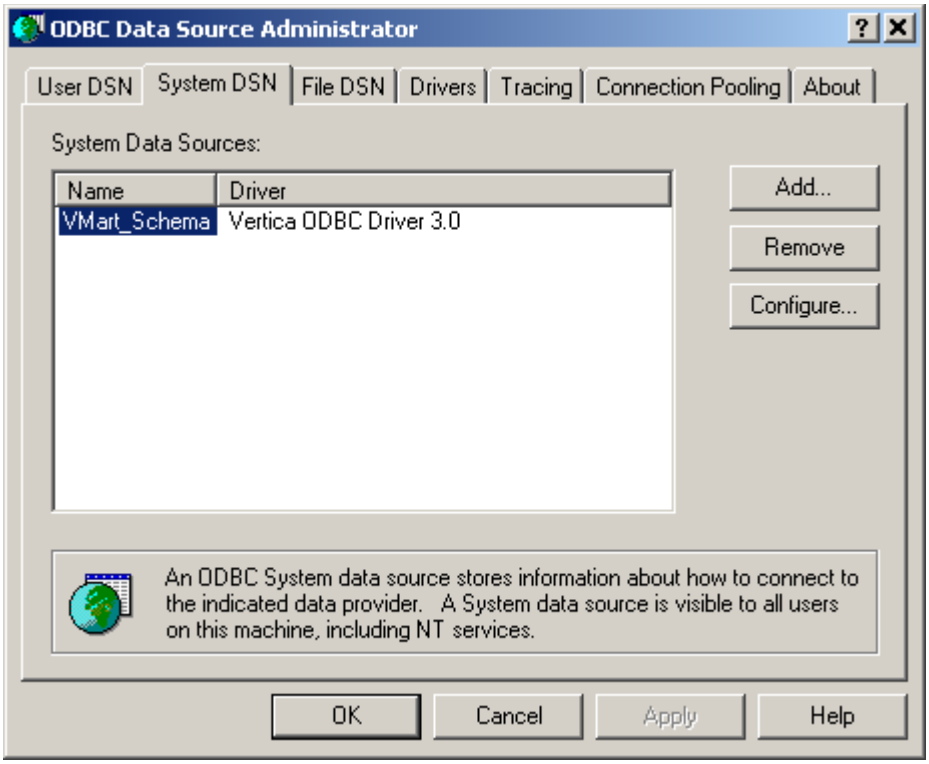


The following list describes all the fields in the Vertica ODBC Driver Setup dialog:

- **Data Source —** The name by which the new data source appears in menus. There is no special significance to the default name.
- **Description —** Additional information about the data source. In this example, the description is "VMart schema database."
- **Server —** The hostname or IP address of any active node within a Vertica database.
- **Port —** The port number on which Vertica listens for ODBC connections. For example, 5433.
- **Database —** The name of the database running on the server. This example uses vmartdb for the Vmart schema.
- **User Name —** 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.
- **Password —** The password for the specified user name. This example leaves the password field blank.
- **Read Only —** Prevents users of this data source from writing to the database. The default is unselected.
- **MyLog —** Logs only debug messages, which is useful for debugging problems with the ODBC driver. The default is unselected.
- **CommLog —** Logs all communications between the application and the server, which is useful for application debugging. The default is unselected.
- **READ COMMITTED —** (Default) Allows concurrent transactions and prevents dirty reads by reading data from the last epoch and committing changes to the current epoch.

- **SERIALIZABLE** — Is the most strict level of SQL transaction isolation. Although this isolation level permits transactions to run concurrently, it creates the effect that transactions are running in serial order. It acquires locks for both read and write operations, which ensures that successive SELECT commands within a single transaction always produce the same results. SERIALIZABLE isolation always uses the current epoch.
- **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.

6 Optionally click **Test Connection** and then click **Save**.



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

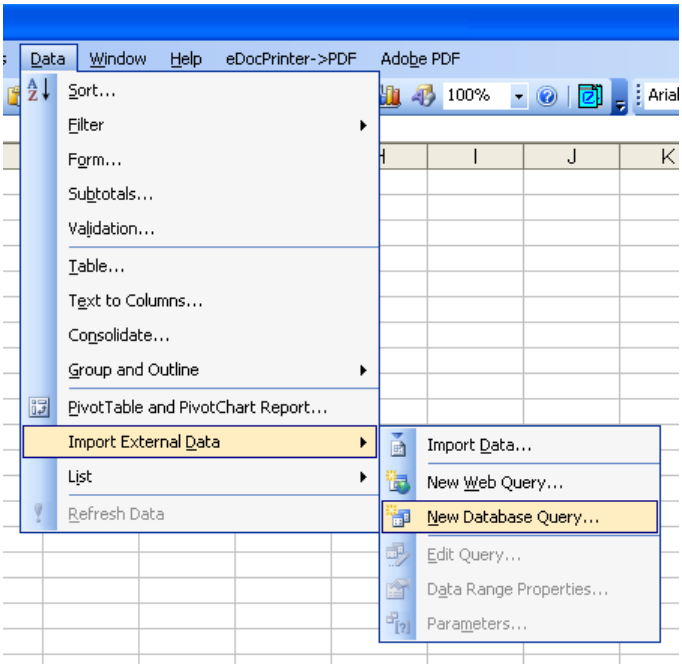
8 ***Verify*** (page 32) that applications can use the DSN to connect to an ODBC data source.

Testing a DSN Using Excel 2003

This section uses Microsoft Excel 2003 to verify that an application can connect to an ODBC data source. You can accomplish the same thing with any ODBC application.

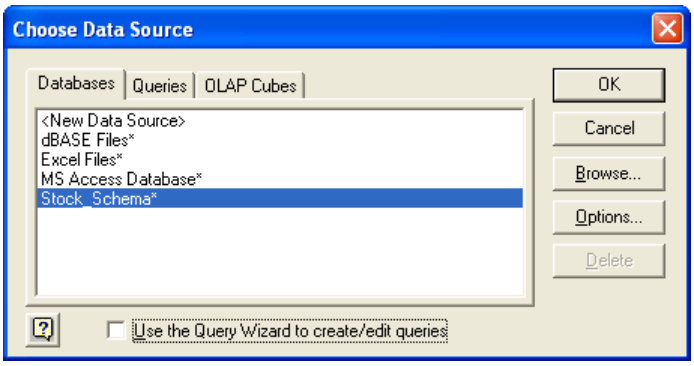
1 Open Excel.

2 From the menu, select **Data** > **Import External Data** > **New Database Query**.

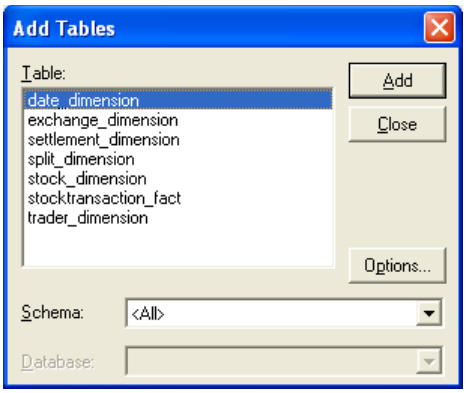


If Microsoft Query is not installed, Excel offers to install it for you.

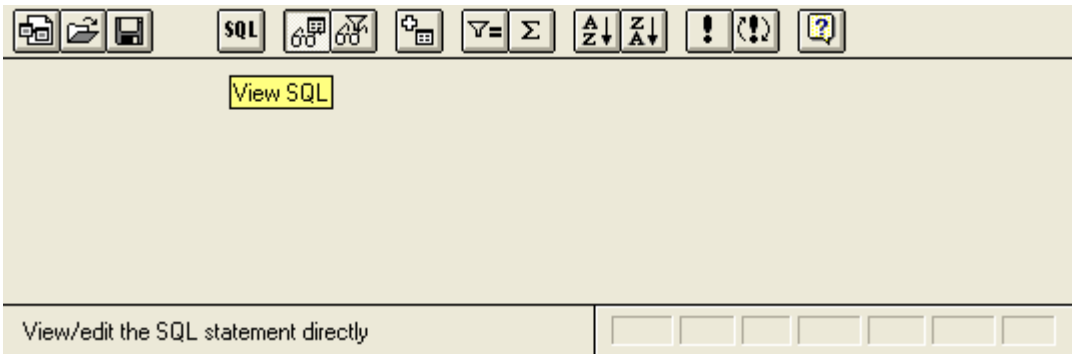
3 Select the data source name (Stock_Schema in this example), make sure the "Use the Query Wizard" check box is deselected and click **OK**.



4 In the Add Tables dialog, click **Close**.

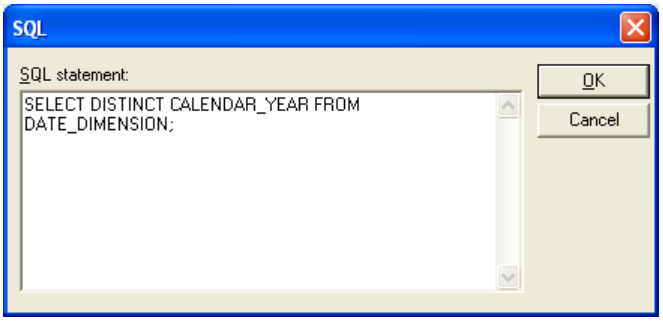


5 Click the **SQL** button.



6 Enter any simple query to test. This example uses the following query:

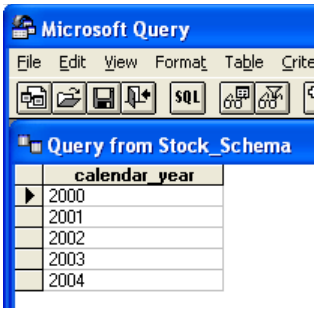
```
SELECT DISTINCT calendar_year FROM date_dimension;
```



7 Click **OK**.

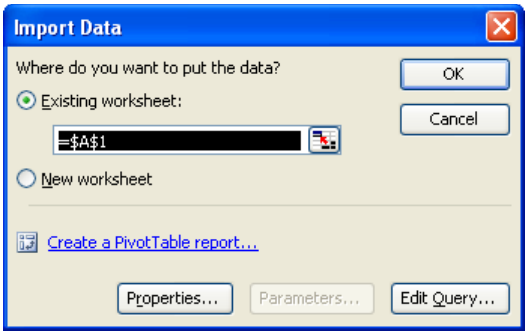
8 If you see the caution, "SQL Query can't be represented graphically. Continue anyway?" click **OK**.

The data values 2000, 2001, 2002, 2003, 2004 indicate that you successfully connected to and ran a query through ODBC.

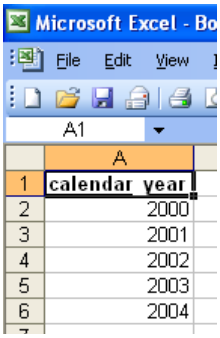


9 Click **File > Return Data to Microsoft Office Excel**.

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



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



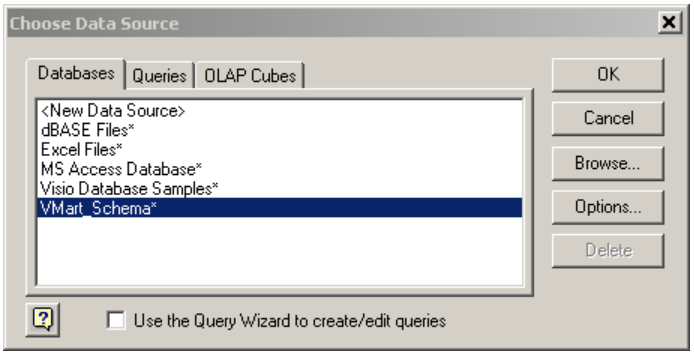
Testing a DSN Using Excel 2007

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.

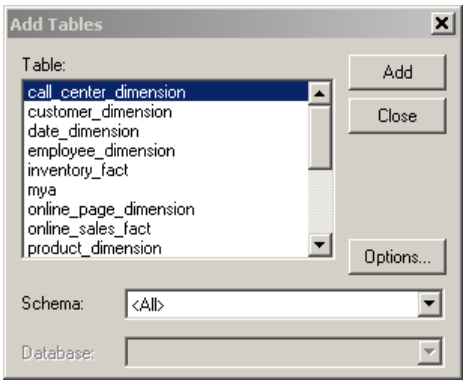
1 Open Excel.

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

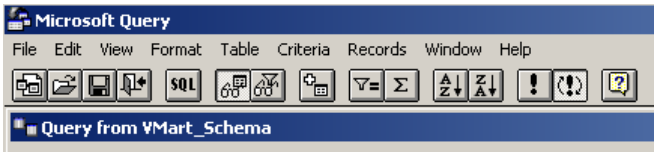
3 Select VMart_Schema*, make sure the "Use the Query Wizard" check box is deselected and click **OK**.



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

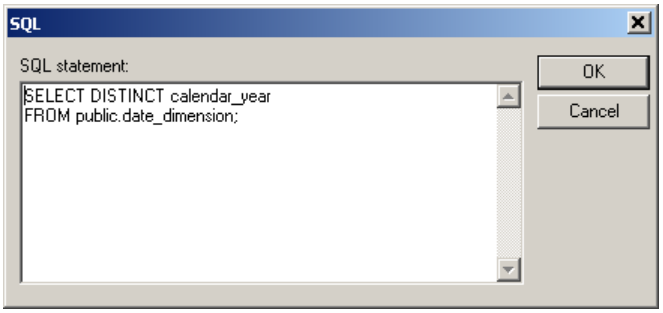


5 The Microsoft Query window opens; click the **SQL** button.



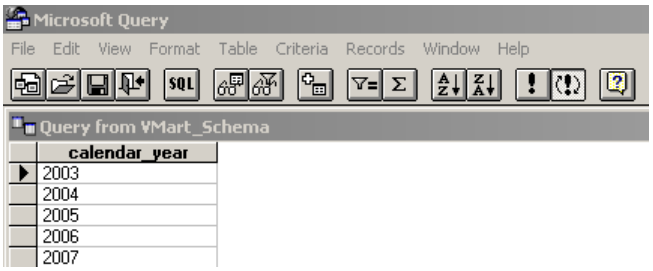
6 In the SQL window write any simple query to test your connection. This example uses the following query:

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



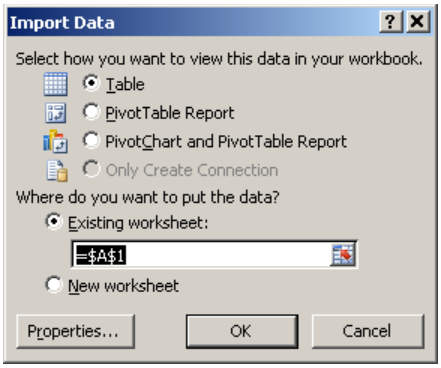
7 If you see the caution, "SQL Query can't be represented graphically. Continue anyway?" click **OK**.

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

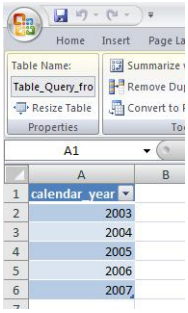


8 Click **File > Return Data to Microsoft Office Excel**.

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



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



Creating User and System DSN Entries

Once you have created and tested a DSN, you need to create user and system DSN entries in the Windows registry. The ***DSN parameters*** (page 38) you set to create these entries are identical, but the paths differ depending on:

- The type of entry (user or system) you want to create.
- Whether the system is a 32 or 64-bit system.
- Whether the driver installed on a 64-bit system is actually a 32 bit driver.

User DSN Paths

- 32 bit - HKEY_CURRENT_USER\Software\ODBC\ODBC.INI\<DSN name>
- 64 bit - HKEY_CURRENT_USER\Software\ODBC\ODBC.INI\<DSN name>
- 32 bit driver on 64 bit system - HKEY_CURRENT_USER\SOFTWARE\WOW6432Node\ODBC\ODBC.INI\<DSN name>

System DSN Paths

- 32 bit - HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\<DSN name>
- 64 bit - HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\<DSN name>
- 32 bit driver on 64 bit system - HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\ODBC\ODBC.INI\<DSN name>

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:

- UNIX and Linux users can edit the odbc.ini file. (See ***Creating an ODBC DSN for Linux and Solaris Clients*** (page 27).) The location of this file is specific to the driver manager.
- Windows users can edit the DSN parameters directly by opening the DSN entry in the Windows registry (for example, at HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\vmartdb). However, the Vertica-preferred method is to follow the steps in ***Creating an ODBC DSN for Windows Clients*** (page 29).
- Parameters can be set while making the connection using SQLDriverConnect().

```
sqlRet = SQLDriverConnect(sql_hDBC, 0,
   (SQLCHAR*)"DSN=VerticaSQL;BinaryDataTransfer=1",
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=VerticaSQL;BinaryDataTransfer=1;ConnSettings=
> set+search_path+to+a,b,c%3 Bset+locale=ch;SSLMode=prefer", SQL_NTS,
> szDNS, 1024,&nSize, SQL_DRIVER_NOPROMPT);
> ```

- Parameters can also be set and retrieved after the connection has been made using SQLConnect(). Parameters can be set and retrieved using SQLSetConnectAttr(),SQLSetStmtAttr(), SQLGetConnectAttr() and SQLGetStmtAttr() API calls.

  For details of the list of Vertica specific parameters see ***Vertica-specific ODBC Header File*** (page 43).

General Parameters

Parameters	Description	Example	Standard/Vertica
Driver	The file path and name of the driver used.	`C:\Program Files\Vertica Systems\Vertica Client Drivers 4.0\lib\vertica_4.0_odbc_3.5.dll`	Standard
ReadOnly	If set to 1, DSN is read only	`1`	Vertica

Description	An optional description for the DSN entry. Insert an empty string to leave the description empty.	""	Standard
Database	The name of the database running on the server.	vmartdb	Standard
Servername	The hostname or IP address of any active node within a Vertica database; for example, host01.	10.10.21.250	Standard
Port	The port number on which Vertica listens for ODBC connections.	5433	Standard
Username	Either the database superuser (same name as the database administrator account) or a user that the superuser has created and granted privileges.	dbadmin	Standard
Password	The password for the specified user name. You may insert an empty string to leave this parameter blank.	""	Standard

Internationalization

Parameters	Description	Example	Standard/Vertica
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.	Locale = en_GB;	Vertica
ColumnsAsChar	By default, when driver is in Unicode mode, character column type is reported as WCHAR. If ColumnsAsChar is set to 1 then driver in unicode mode will return CHAR type for character columns.	ColumnsAsChar=1	Vertica
WideCharSizeIn	Size of the input wide characters specific to platform and	WideCharSizeIn=4	Vertica

	programming environment.		
WideCharSizeOut	Size of the output wide characters specific to platform and programming environment.	`WideCharSizeOut=4`	Vertica

Utilities

Parameters	Description	Example	Standard/Vertica
ConnSettings	This value contains SQL commands to be run immediately after connecting to the server. **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.	`SET SEARCH_PATH = schema1, schema2, public;`	Vertica
TxnReadCommitted	If set to 1, the transaction isolation mode for the connection is READ COMMITTED, otherwise SERIALIZABLE. **Note:** If not specified in the DSN, the ODBC client connection defaults to the transaction level set by the server. See Changing Transaction Isolation Levels in the Administrator's Guide.	1	Vertica
SessionLabel	Allows to uniquely identify a client session on the server.		Vertica

Security

Parameters	Description	Example	Standard/Vertica
SSLMode	The connection setting used for SSL: ▪ always — Requires the server to use SSL. If the server cannot provide an encrypted channel, the connection fails. ▪ prefer (default) — Prefers the server to use SSL. If the server does not offer an encrypted channel, the	prefer	Vertica

	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. For more information about using SSL, see Implementing SSL.		
SSLKeyFile	The file path and name of the client's private key. This file can reside anywhere on the system.	`SSLKeyFile = C:\Program Files\Vertica Systems\home\ dbadmin\client.key`	Vertica
SSLCertFile	The file path and name of the client's public certificate. This file can reside anywhere on the system.	`SSLCertFile = C:\Program Files\Vertica Systems\home\ dbadmin\client.crt`	Vertica

Load

Parameters	Description	Example	Standard/Vertica
BatchInsertEnforceLength	Enforces rejection of strings longer than the column width. If set to 1 then the string is rejected, when set to 0 the string is truncated. Default is false (value of 0).	`0`	Vertica
DirectBatchInsert	Determines whether a batch is inserted directly into the ROS (1) or WOS/ROS (0). By default batches are inserted using AUTO mode.	`0`	Vertica

Note: In Vertica 4.1, the batch-related parameters Use35CopyFormat, BatchAutoComplete, BatchInsertManaged, and ReportParamSuccess have been deprecated. These settings are no longer needed for Vertica 4.1's new batch load behavior (see ***Using Batch Inserts*** (page 51) for details). Setting any of these parameters has no effect. In addition, the Use35CopyParameters parameter has also been deprecated. In addition, the AbortOnError parameter is obsolete, since the Vertica ODBC client driver has better error reporting ability. This parameter still works, but you should avoid using it.

Performance/Query

Parameters	Description	Example	Standard/Vertica
LRSPath	Specifies the location of the temporary file on the client system that is used to store large result sets. Windows Default: %TEMP% Linux Default: /tmp	`/tmp`	Vertica
LRSStreaming	If set to 1 (the default), the ODBC driver pauses the query execution when the memory cache on the client is full and resumes execution of the query after the memory cache rows are retrieved by the ODBC application using SQLFetch. If the value is false (0), the driver dumps large result sets to the temporary file specified by SQL_ATT_VERTICA_LRS_STREAMING. By default, this parameter is set to 1.	`1`	Vertica
BinaryDataTransfer	If set to 1, the driver requests binary data transfer from the server. The following data types can benefit from binary data transfer: ▪ All date/time types (DATE, TIME, TIMESTAMP) ▪ NUMERIC ▪ BIGINT with large values (>99999999) **Note:** The Vertica ODBC client driver does not validate the binary data it sends to the server. Your own application should verify the data it is sending (especially data/time values) are valid before inserting them in a batch to send to Vertica.	`0`	Vertica
MaxMemoryCache	Size of memory buffer for the large result sets in streaming mode.	`67108864`	Vertica

Third-Party Integration

Parameters	Description	Example	Standard/Vertica
BoolsAsChar	If set to 1, the driver reports Boolean type as SQLCHAR, otherwise as SQLBIT.	0	Vertica
SuppressWarnings	If set to 1, the driver converts SQL_SUCCESS_WITH_INFO to SQL_SUCCESS. If set to 0, warnings are not suppressed.	0	Vertica

Troubleshooting

Parameters	Description	Example	Standard/Vertica
Debug	If set to 1, the driver debug information is saved in the C:/mylog_*NNN*.log (on Windows) or /tmp/mylog_*NNN*.log (on Linux and Solaris), where *NNN* is the application process ID.	0	Vertica
Trace	If this flag is 1, tracing is turned on for the driver manager. If the value is 0, tracing is turned off. See also TraceFile and TraceDll.	0	Standard
TraceFile	If tracing is turned on, specify the full path of the file to which ODBC calls are written.	/home/my_dir/odbctrace.out	Standard
TraceDll	If tracing is turned on, specify the name of the trace DLL that performs the tracing.	/usr/local/lib/odbctrac.so	Standard

Vertica-specific ODBC Header File

The Vertica ODBC header file, `verticaodbc.h` contains the following:

```
#define SQL_ATTR_VERTICA_LRSPATH      12000
#define SQL_ATTR_VERTICA_MAX_MEM_CACHE 12001
#define SQL_ATTR_VERTICA_LRS_STREAMING 12002
#define SQL_ATTR_VERTICA_SUPPRESS_WARNINGS 12003
#define SQL_ATTR_VERTICA_DIRECT_BATCH_INSERT 12004
#define SQL_ATTR_VERTICA_BATCH_AUTO_COMPLETE 12008
#define SQL_ATTR_VERTICA_BATCH_INSERT_NULL 12009
#define SQL_ATTR_VERTICA_BATCH_INSERT_RECORD_TERMINATOR 12010
#define SQL_ATTR_VERTICA_LOCALE 12011
```

The following table describes these parameters.

Parameter	Description	Associated Function
SQL_ATTR_VERTICA_ABORT_ON_ERROR	Instructs Vertica to abort on error (1) or not (0). By default Vertica does not abort when it encounters an error.	SQLSetConnectAttr() SQLSetStmtAttr() SQLGetConnectAttr() SQLGetStmtAttr()
SQL_ATTR_VERTICA_BATCH_INSERT_NULL	Sets the batch null value indicator. By default, Vertica uses 'null'. If you have this string in your data, change the null value indicator. See the NULL parameter in the COPY statement for more information about choosing a null indicator.	SQLSetConnectAttr() SQLGetConnectAttr()
SQL_ATTR_VERTICA_BATCH_INSERT_RECORD_TERMINATOR	Sets the batch insert record terminator. By default, Vertica uses "a\v\b". In the unlikely case you have this string in your data, change the record terminator. See the RECORD TERMINATOR parameter for the COPY statement for more information about choosing a record terminator.	SQLSetConnectAttr() SQLGetConnectAttr()
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 into the ROS.	SQLSetConnectAttr() SQLSetStmtAttr() SQLGetConnectAttr() SQLGetStmtAttr()
SQL_ATTR_VERTICA_LRS_PATH	Specifies the location of the flat file on the client system that is used to store large result sets. Windows Default: %TEMP% Linux Default: /tmp	SQLSetConnectAttr() SQLGetConnectAttr()
SQL_ATTR_VERTICA_LRS_STREAMING	Determines whether the driver uses a temporary file to keep the large result set, or use streaming mode to fetch the large result set from the database server. If the value is true (1), the ODBC driver pauses the query execution when the memory cache on the client is full and resumes execution of the query after the memory cache rows are retrieved by the ODBC application using SQLFetch. If the value is false (0), the driver	SQLSetConnectAttr() SQLSetStmtAttr() SQLGetConnectAttr() SQLGetStmtAttr()

	dumps large result sets to the flat file specified by SQL_ATT_VERTICA_LRS_STREAMING. By default, this parameter is set to 1.	
SQL_ATTR_VERTICA_MAX_MEM_CACHE	Sets the size of the buffer in theVertica driver that is used to temporarily store result sets. By default the size is 67108864 (64MB). Tip: To decrease the time it takes the client application to receive the result sets, you could reduce the value of the cache to as little as 256K.	SQLSetConnectAttr() SQLSetStmtAttr() SQLGetConnectAttr() SQLGetStmtAttr()
SQL_ATTR_VERTICA_SUPPRESS_WARNINGS	Determines whether warnings are suppressed (1) or not (0) for SQLExecDirect(), SQLExecDirectW(), and SQLExecute() and if SQL_SUCCESS_WITH_INFO is replaced with SQL_SUCCESS. Warnings are not suppressed by default.	SQLSetConnectAttr() SQLSetStmtAttr() SQLGetConnectAttr() SQLGetStmtAttr()
SQL_ATTR_VERTICA_LOCALE	Changes the locale from en_US@collation=binary to the ICU locale specified.	SQLSetConnectAttr() SQLGetConnectAttr()

Note: The parameters SQL_ATTR_VERTICA_BATCH_AUTO_COMPLETE, SQL_ATTR_VERTICA_NUM_ACCEPTED_ROWS, and SQL_ATTR_VERTICA_NUM_REJECTED_ROWS available in versions of Vertica before 4.1 have been deprecated.

Supported ODBC Functions

The ODBC driver for Vertica supports the following ODBC functions for Microsoft ODBC 3.5. Any deviations from the standard are noted.

Use	Function	Support
Connecting to a data source	SQLAllocHandle	Standard
	SQLConnect	Standard
	SQLDriverConnect	This function differs from the standard in the following ways: ▪ The connection string may contain any Vertica-specific parameters specified in the INI file. ▪ When the client application uses

		SQLDriverConnect, the connection string must supply all the required information for making the connection. For example, the driver only supports displaying a dialog for users to enter missing values under MS Windows. If you are using Linux or UNIX, you must specify all required values through the connection string.
	SQLBrowseConnect	Standard
Obtaining information about a driver and data source	SQLGetInfo	Standard
	SQLGetFunctions	Standard
	SQLGetTypeInfo	Standard
Setting and retrieving driver attributes	SQLSetConnectAttr	This is a standard call, but the driver provides its own attributes.
	SQLGetConnectAttr	This is a standard call, but the driver provides its own attributes.
	SQLSetEnvAttr	Standard
	SQLGetEnvAttr	Standard
	SQLSetStmtAttr	This is a standard call, but the driver provides its own attributes.
	SQLGetStmtAttr	This is a standard call, but the driver provides its own attributes.
Setting and retrieving descriptor fields	SQLGetDescField	Standard
	SQLGetDescRec	Standard
	SQLSetDescField	Standard
	SQLSetDescRec	Standard

Preparing SQL requests	SQLPrepare	For batch inserts, the driver converts the prepare statement from INSERT to COPY. For example, the following would be converted:
		`INSERT INTO <table> [<columns_list>]`
		`  VALUES (?, ?, ?...);`
		Note that not every INSERT can be converted to COPY. If the list of values contains either of the following, it cannot be converted:
		▪ a literal; For example: ('a' , ?)
		▪ a function; For example: (current_time() , ?)

	SQLBindParameter	Standard
	SQLParamOptions	Standard
Submitting requests	SQLExecute	Standard
	SQLExecDirect	Standard
	SQLNativeSql	Standard

	SQLDescribeParam	This function is supported, but there could be cases in which the parameter type returns VARCHAR(64000).
	SQLNumParams	Standard
	SQLParamData	Standard
	SQLPutData	Standard
Retrieving results and information about results	SQLRowCount	Standard Note: In version 3.5, when the BatchAutoComplete parameter was not set, this function always returned zero. In version 4.0, or in earlier versions when BatchAutoComplete was set, this function returned the number of rows inserted by the last insert or batch. From version 4.1, this function acts according to the ODBC specifications, returning the number of rows affected by the last SQLExecute.
	SQLNumResultsCols	Standard
	SQLDescribeCol	Standard
	SQLColAttribute	Standard
	SQLBindCol	Standard
	SQLFetch	Standard
	SQLFetchScroll	Only supports SQL_FETCH_NEXT as the orientation.
	SQLGetData	Standard
	SQLSetPos	Only supports the SQL_POSITION and SQL_REFRESH options.
	SQLMoreResults	Vertica does not support the multi-statement batch (MSB) feature. Calls to this function will always return SQL_NO_DATA. See ***Unsupported ODBC Functions and Parameters*** (page 48) for details.
	SQLGetDiagField	Standard
	SQLGetDiagRec	Standard
Obtaining information about the data source's system tables (catalog functions)	SQLColumns	Standard
	SQForeignKeys	Standard
	SQLPrimaryKeys	Standard
	SQLSpecialColumns	Standard
	SQLTables	Standard
Terminating a statement	SQLFreeStmt	Standard
	SQLCloseCursor	Standard
	SQLCancel	Standard
	SQLEndTran	Standard
Terminating a	SQLDisconnect	Standard

connection	SQLFreeHandle	Standard

Notes

- The Vertica ODBC driver supports one cursor per connection. Attempting to use more than one cursor per connection will result in an error. For example, you will receive an error if you execute a statement while another statement has a result set open.
- The Vertica ODBC driver does not support scrollable cursors.

Unsupported ODBC Functions and Parameters

The ODBC driver for Vertica does not support the following ODBC functions.

Use	Function
Obtaining information about a driver and data source	SQLDataSources SQLDrivers SQLSetCursorName SQLSetScrollOptions
Preparing SQL requests	SQLGetCursorName SQLBulkOperations
Obtaining information about the data source's system tables (catalog functions)	SQLColumnPrivileges SQLProcedureColumns SQLProcedures SQLStatistics SQLTablePrivileges
Terminating a statement	SQLCancelHandle Function

Cursors Per Connection

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

Multi-Statement Batches

Vertica does not support the ODBC multi-statement batch (MSB) feature. While you can submit a batch that contains multiple statements, you only receive the result of the last statement executed. The SQLMoreResults function always returns SQL_NO_DATA.

Scrollable Cursors

Vertica's ODBC driver does not support scrollable cursors.

Unsupported Parameters

The SQL_ATTR_MAX_LENGTH parameter is not supported by the Vertica ODBC client driver. You can assign a value to this parameter without causing an error, however it has no effect.

Setting the Locale for ODBC Sessions

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

- Specify the locale at connection through the *odbc.ini* file. See:
  - ***Creating an ODBC DSN for Linux and Solaris Clients*** (page 27)
  - ***Creating an ODBC DSN for Windows Clients*** (page 29)
  - ***DSN Parameters*** (page 38)
- Use the `SQLSetConnectAttr()` method with the `SQL_ATTR_VERTICA_LOCALE` constant and specify the ICU string as the attribute value. See:
  - ***Vertica-Specific ODBC Header File*** (page 43)
  - ***DSN Parameters*** (page 38)

  For example:

  ```
  SQLSetConnectAttr(dbc, SQL_ATTR_VERTICA_LOCALE,
     (SQLPOINTER)strLocale, SQL_NTS);
  ```
- Use Locale 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)
  ```

Notes

- 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 Vertica server and converts data sent by the 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.

Loading Data Through ODBC

The following methods enable you to load data from a client application to Vertica through ODBC.

- ***Single row insert*** (page 50)
- ***Batch insert*** (page 51)
- ***COPY statement*** (page 61)
- ***LCOPY statement*** (page 62)

Additionally, you can:

- ***Load data into the WOS/ROS*** (page 62)
- ***Load batches in parallel*** (page 60)

Vertica provides two formats to load data using ODBC:

- Text format with delimiters (default LCOPY command)
- Native binary format or native varchar format when required (default for batch inserts in Vertica 4.0)

Notes

- When using NATIVE BINARY mode with ODBC, you application must validate that the data it is inserting is in the correct format, since the ODBC client driver does not perform any validation of its own. This is especially true of date and time data types, which are more complex than the simpler data types, and are easy to format incorrectly.
- Batch inserts will automatically use either the NATIVE BINARY or NATIVE VARCHAR formats. NATIVE BINARY is used if the application data types match the actual table data types exactly (including maximum lengths of CHAR/VARCHAR and precision/scale of numeric data types), which provides best possible load performance. If there is any data type mismatch, NATIVE VARCHAR is used. NATIVE varchar format uses a similar file format to native binary, but all fields are represented as strings in CHAR or VARCHAR. Conversion to the actual table data type is done on the database server; thus, NATIVE VARCHAR does not provide the same efficiency as NATIVE BINARY. However, NATIVE VARCHAR provides the convenience of not having to use delimiters or escape special characters, such as quotes, which can make working with client applications easier.

Using a Single Row Insert

The easiest way to load data into Vertica is to run an INSERT SQL statement. 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 Batch Inserts

Batch load insert is a method for bulk loading data into Vertica by loading one or more consecutive batches. Like a typical batch load, it uses parameterized statements that work with bound variables. The data to be loaded is stored in an array and bound to the parameter.

In Vertica Version 4.1, all sequential batch loads are handled behind the scenes by a single COPY statement. Ending the batch insert transaction, closing the cursor, or executing a non-INSERT statement ends the COPY statement. Using a single COPY statement for multiple batches makes batch loading more efficient by reducing the overhead of inserting individual batches. It also allows the COPY statement to combine individual batches into larger and more efficient ROS containers.

Even though a single COPY command handles multiple batches within a transaction, you can still find which (if any) rows were rejected due to invalid row formats or data type issues after each batch is loaded. When you are within a transaction, getting a report of accepted or rejected rows (for example, by using the SQLRowCount function) will return the results from the last batch. After the transaction is committed, getting these parameters returns the results for the entire transaction.

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 batches share a COPY statement, errors in a batch can cause earlier batches in the same transaction to be rolled back. For example, these rollbacks can occur if you enable the abortOnError connection property, which would cause the entire COPY statement 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. Leaving autocommit enabled means that each batch starts a new transaction which will result in more ROS containers being created and a much higher overhead for Vertica.
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.

```
//Header files:
#include <sql.h>
#include <sqltypes.h>
```

```
#include <sqlext.h>

#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cassert>
#include <cstring>

// The following include file will depend on your
// platform and where you installed Vertica.
#include "/opt/vertica/include/verticaodbc.h"
#include "utils.h"

// Helper function that prints SQL error messages
static void PrintError( SQLSMALLINT siType, SQLHANDLE shHandle )
{
    SQLINTEGER siError;
    SQLSMALLINT siAvail;
    SQLCHAR szError[ 1024 ], szState[ 256 ];

    SQLGetDiagRec( siType, shHandle, 1, szState, &siError,
        szError, sizeof( szError ), &siAvail );
    printf( "ERROR: %s\n", szError );
}


int main(int argc, char* argv[])
{
    // Get the environment
    SQLHENV hdlEnv;
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hdlEnv);
    SQLSetEnvAttr(hdlEnv,SQL_ATTR_ODBC_VERSION,
        (void*)SQL_OV_ODBC3, 0); // or SQL_OV_ODBC30
    // Set up a connection to the database
    SQLHDBC hdlDbc;
    SQLAllocHandle(SQL_HANDLE_DBC, hdlEnv, &hdlDbc);

    std::cout << "Connect to DB" << std::endl;
    SQLRETURN rc;

    // Hard-coded database connection settings. Real applications
    // shouldn't do this!
    const char *dsnName = "ExampleDB";
    const char *userID = "ExampleUser";
    const char *passwd = "password123";
    rc = SQLConnect(hdlDbc, (SQLCHAR*)dsnName,SQL_NTS,
        (SQLCHAR*)userID, SQL_NTS, (SQLCHAR*)passwd, SQL_NTS);
    // If connection did not succeed, exit.  if(rc != SQL_SUCCESS) return 1;
    // Turn off autocommit, so multiple batches can be loaded in a
    // transaction.
    std::cout << "Disable Autocommit." << std::endl;
    rc = SQLSetConnectOption(hdlDbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
    if(rc != SQL_SUCCESS) printf("Failed to disable autocommit!\n");
    // Set up a statement handle
```

```
SQLHSTMT hdlStmt;
SQLAllocHandle(SQL_HANDLE_STMT, hdlDbc, &hdlStmt);

// Create a table into which we can store data
std::cout << "Create table." << std::endl;

rc = SQLExecDirect(hdlStmt, (SQLCHAR*)"CREATE TABLE customers "
    "(CustID int, CustName varchar(100), Phone_Number char(15));",
    SQL_NTS);
if(rc != SQL_SUCCESS)
        PrintError( SQL_HANDLE_STMT, hdlStmt );
// Create the prepared statement. This will insert data into the
// table we created above.
rc = SQLPrepare (hdlStmt, (SQLTCHAR*)"INSERT INTO customers (CustID, "
    "CustName,  Phone_Number) VALUES(?,?,?)", SQL_NTS) ;
if(rc != SQL_SUCCESS)
    PrintError( SQL_HANDLE_STMT, hdlStmt );
// 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
SQLBindParameter(hdlStmt, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
    0, 0, (SQLPOINTER)custIDs, sizeof(*custIDs) , NULL);
SQLBindParameter(hdlStmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
    50, 0, (SQLPOINTER)custNames, sizeof(custNames[0]), NULL);
SQLBindParameter(hdlStmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
    15, 0, (SQLPOINTER)phoneNums, sizeof(phoneNums[0]), NULL);

// Tell the ODBC driver how many rows we have in the
// array.
SQLSetStmtAttr( hdlStmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)4, 0 );
// Variables to hold the number of accepted and rejected rows.
SQLINTEGER acc_rows = 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<=4; batchLoop++) {
    // Execute the prepared statement, loading all of the data
    // in the arrays.
    printf("Batch #%d: ", batchLoop);
    rc = SQLExecute(hdlStmt);
    if(rc != SQL_SUCCESS)
        PrintError( SQL_HANDLE_STMT, hdlStmt );
    // Print the accepted rows from the last batch.
    SQLRowCount(hdlStmt, &acc_rows);
    printf("Rows affected: %d\n", (int)acc_rows);
}

// Done with batches, commit the transaction
std::cout << "Commit Transaction" << std::endl;
rc = SQLEndTran(SQL_HANDLE_DBC, hdlDbc, SQL_COMMIT);
if(rc != SQL_SUCCESS)
```

```
        printf("Failed to commit transaction.\n");
    // Get the accepted rows from the transaction.
    SQLRowCount(hdlStmt, &acc_rows);
    printf("Transaction affected %d rows.\n", (int)acc_rows);

    // Get rid of the table
    rc = SQLExecDirect(hdlStmt, (SQLCHAR*)"DROP TABLE customers;",
        SQL_NTS);
    if(rc != SQL_SUCCESS)
        printf("Failed to drop table.\n");
    // Clean up
    std::cout << "Free handles." << std::endl;
    SQLFreeHandle(SQL_HANDLE_STMT, hdlStmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdlDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hdlEnv);

    return 0;
}
```

The result of running the above code is shown below.

```
Connect to DB
Disable Autocommit.
Create table & projection.
Batch #1: Rows affected: 4
Batch #2: Rows affected: 4
Batch #3: Rows affected: 4
Batch #4: Rows affected: 4
Commit Transaction
Transaction affected 16 rows.
Free handles.
```

Using Batch Insert With Version 4.0 Drivers

Vertica Version 4.1 has changed the way batch inserts are handled using ODBC by combining all batches loaded in a transaction into a single COPY statement. This results in a faster and more efficient data load process.

The new batch insert behavior has deprecated some ODBC parameters that were available in Vertica Version 4.0. If your batch load process relies on these older parameters, you can retain the old ODBC batch behavior by using the 4.0 ODBC drivers with the Vertica 4.1 server.

For details on using the Vertica Version 4.0 ODBC driver, see the Vertica® Analytic Database Version 4.0 documentation.

Note: Future versions of Vertica may not work with the 4.0 ODBC drivers. You should update your client applications to take advantage of the new batch loading behavior to avoid future incompatibility problems.

Using Prepared Statements

Vertica supports using server-side prepared statements with both ODBC and JDBC. Prepared statements enable you to write a statement once, and then run it many times with different parameters. This is accomplished by passing placeholders instead of parameters to the server and binding user input to the parameter.

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.

  The query only needs to be parsed the first time it is passed to the server.
- 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.
- Binding direct variables to return columns.

  By pointing to data structures, the code doesn't have to perform extra transformations.

This section:

- Describes how to ***create and execute prepared statements*** (page 55)
- Provides a ***command reference for prepared statements*** (page 55)

Creating and Executing Prepared Statements

To prepare and execute statements:

1. Call ***SQLPrepare*** (page 55) to prepare the statement.
2. (Optional) Bind each parameter to a program variable by using ***SQLBindParameter*** (page 56). Configure any data-at-execution parameters.
3. For each execution of a prepared statement:
   - If the statement has parameter markers, put the data values into the bound parameter buffer.
   - Call ***SQLExecute*** (page 57) to execute the prepared statement.

If data-at-execution input parameters are used, SQLExecute returns SQL_NEED_DATA. Send the data in chunks by using SQLParamData and SQLPutData.

Command Reference for Prepared Statements

This section describes the ODBC APIs for using prepared statements. You can use prepared statements to supply data to a query at execution time.

SQLPrepare

When you call SQLPrepare() with a string containing a SQL statement, the driver sends the string to the server and stores the statement identifier for later execution. The string is stored on the server and is not sent again when the prepared statement is run more than once.

Syntax

```
SQLRETURN SQLPrepare (
    SQLHSTMT StatementHandle,
    SQLCHAR *StatementText,
    SQLINTEGER TextLength
);
```

Parameters

StatementHandle	[Input] Statement handle
StatementText	[Input] SQL text string
TextLength	[Input] Length of *StatementText in characters

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

SQLBindParameter

When you call SQLBindParameter(),the driver binds the statement parameters but does not communicate with the server.

Syntax

```
SQLRETURN SQLBindParameter (
    SQLHSTMT StatementHandle,
    SQLUSMALLINT ParameterNumber,
    SQLSMALLINT InputOutputType,
    SQLSMALLINT ValueType,
    SQLSMALLINT ParameterType,
    SQLULEN ColumnSize,
    SQLSMALLINT DecimalDigits,
    SQLPOINTER ParameterValuePtr,
    SQLINTEGER BufferLength,
    SQLLEN *StrLen_or_IndPtr
);
```

Parameters

StatementHandle	[Input] Statement handle
ParameterNumber	[Input] Parameter number, ordered sequentially in increasing parameter order, starting at 1
InputOutputType	[Input] The type of the parameter
ValueType	[Input] The C data type of the parameter
ParameterType	[Input] The SQL data type of the parameter

ColumnSize	[Input] The size of the column or expression of the corresponding parameter marker
DecimalDigits	[Input] The decimal digits of the column or expression of the corresponding parameter marker
ParameterValuePtr	[Deferred Input] A pointer to a buffer for the parameter's data
BufferLength	[Input/Output] Length of the ParameterValuePtr buffer in bytes
StrLen_or_IndPtr	[Deferred Input] A pointer to a buffer for the parameter's length

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

SQLExecute

When you call SQLExecute, the driver sends the statement identifier and parameter values to the server and returns the result set or an error. The driver also returns semantic and syntactic errors at this point.

Syntax

```
SQLRETURN SQLExecute (SQLHSTMT StatementHandle);
```

Parameters

StatementHandle	[Input] Statement handle

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE

Notes

This executes a prepared statement, using the current values of the parameter marker variables if any parameter markers exist in the statement.

SQLParamData

SQLParamData is used together with SQLPutData to supply parameter data at statement execution time.

Syntax

```
SQLRETURN SQLParamData (
    SQLHSTMT StatementHandle,
    SQLPOINTER *ValuePtrPtr
);
```

Parameters

StatementHandle	[Input] Statement handle

ValuePtrPtr	[Output] Pointer to a buffer in which to return the address of the ParameterValuePtr buffer specified in SQLBindParameter (for parameter data) or the address of the TargetValuePtr buffer specified in SQLBindCol (for column data), as contained in the SQL_DESC_DATA_PTR descriptor record field

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

SQLPutData

SQLPutData allows an application to send data for a parameter or column to the driver at statement execution time. This function can be used to send character or binary data values in parts to a column with a character, binary, or data source–specific data type (for example, parameters of the SQL_LONGVARBINARY or SQL_LONGVARCHAR types).

Syntax

```
SQLRETURN SQLPutData (
    SQLHSTMT StatementHandle,
    SQLPOINTER DataPtr,
    SQLLEN StrLen_or_Ind
);
```

Parameters

StatementHandle	[Input] Statement handle
DataPtr	[Input] Pointer to a buffer containing the actual data for the parameter or column. The data must be in the C data type specified in the ValueType argument of SQLBindParameter (for parameter data) or the TargetType argument of SQLBindCol (for column data)
StrLen_or_Ind	[Input] Length of *DataPtr. Specifies the amount of data sent in a call to SQLPutData. The amount of data can vary with each call for a given parameter or column

Returns

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Tracking Load Status on the Server with ODBC

The client can track load status on the server for the last completed database load within the current session by:

- ***Identifying the number of rows that were accepted or rejected*** (page 59)
- ***Identifying which rows were accepted or rejected*** (page 59)

Both methods are useful for determining the status of a load in cases in which data is loaded regardless of any load errors encountered. However, identifying the number of accepted or rejected rows has virtually no performance impact on the server while identifying the status of all the rows in the load slightly affects performance. This occurs because the server sends the row number for each rejected row to the client which, in turn, receives this data. Additionally, the data must be loaded into an array that is supplied by the application.

Note: Data regarding loads does not persist and is dropped when a new load is initiated.

Identifying the Number of Accepted Rows (ODBC)

Vertica tracks the number of rows that were accepted during loading, which you can retrieve using the SQLRowCount function. If you are loading data in batches, you can get the total number of rows loaded into the database at two points in the load process:

- After each batch is inserted, you can get the number of accepted rows for the batch.
- After a transaction containing multiple batch loads is complete, you can get the total number of accepted rows for all of the batches in the transaction.

If you are loading a batch with auto-complete (BatchAutoComplete) enabled (the default), you can only retrieve the accepted row counts for that batch, since the transaction used to load the batch is automatically committed after the load is finished. In order to get the total for several batches, you need to disable auto-complete, then load the batches, and finally commit the transaction that was started by the first batch load either explicitly using SQLEndTran, by executing SQLCloseCursor, or by executing any statement other than an INSERT statement.

See Tracking Load Status for Batch Inserts and Updates for detailed examples.

Identifying Accepted and Rejected Rows (ODBC)

You can track the status of each row being loaded in a batch by binding an array to a statement using the SQL_ATTR_PARAMS_PROCESSED_PTR statement attribute. When a row status is sent from the server to the client, the driver loads the status of each row in the database load into the array that you supplied.

The following example creates a pointer to an array, loads the array with the row number and status for each row in the load, and then prints the results to stdout.

```
retcode = SQLSetStmtAttr( hStmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)lRows, 0 );
    retcode = SQLSetStmtAttr( hStmt, SQL_ATTR_PARAM_STATUS_PTR, rowStatus, 0 );
    retcode = SQLSetStmtAttr( hStmt, SQL_ATTR_PARAMS_PROCESSED_PTR,
&ulRowsProcessed, 0 );
    for ( int i = 1; i <= lCols; i ++ )
    {
        retcode = SQLBindParameter( hStmt, i, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, pplBuffer[ i - 1 ], 0, NULL );
    }

    retcode = SQLExecDirect( hStmt, (SQLCHAR*)szInsert, (SQLINTEGER)strlen(
szInsert ) );
    SQLCloseCursor( hStmt );
    if ( ulRowsProcessed != lRows )
    {
        printf( "Rows Processed: %d\nShould have been %d\n", ulRowsProcessed,
lRows );
```

```
    }

    printf("Parameter Set  Status\n");
    printf("-------------  -------------\n");
    for (unsigned int i = 0; i < ulRowsProcessed; i++) {

        switch (rowStatus[i]) {
        case SQL_PARAM_SUCCESS:
            printf("%13d  Success\n", i);
            break;

        case SQL_PARAM_ERROR:
            printf("%13d  Error\n", i);
            break;

                }
    }
```

See Tracking Load Status for Batch Inserts and Updates for detailed examples.

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 on the Server*** (page 92) 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 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 closing the cursor using SQLCloseCursor(), 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, (SQLPOINTER)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 is useful for bulk loading cleansed data from a file on the database server into Vertica. The advantage of this method is that it is the most efficient way to load data into Vertica because the file resides on the database server. In some cases, however, the user may not have access to the database server. In these cases, the user can use LCOPY.

If you intend to use COPY to load data, determine the approximate size of the load. For large loads, load the data into the ROS. For small loads, load it directly into the WOS.

See the COPY statement for more information about its syntax and use.

The following example loads data into the WOS (Write Optimized Store)/ROS (Read Optimized Store.

```
ret = SQLExecDirect (hstmt, (SQLTCHAR*)"COPY \"public\".\"test\"(c1,c2)FROM 'data.csv' NULL 'null'
DELIMITER \", SQL_NTS);
```

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

```
ret = SQLExecDirect (hstmt, (SQLTCHAR*)"COPY \"public\".\"test\"(c1,c2)FROM 'data.csv' NULL 'null'
DELIMITER \",\" DIRECT", SQL_NTS);
```

Using the LCOPY Statement

The LCOPY statement is useful for bulk loading cleansed data from a file on the client machine into Vertica. The advantage of this method is that it does not require the user to have access to the server. However, LCOPY is proprietary to Vertica and can only be used with custom client applications through ODBC. It does not support any other methods of database connectivity, and Traditional ETL tools must be modified to invoke it.

If you intend to use LCOPY to load data, determine the approximate size of the load. For large loads, load the data into the ROS. For small loads, load it into the WOS/ROS.

See the LCOPY statement for more information about its syntax and use.

The following example loads data into the WOS (Write Optimized Store)/ROS (Read Optimized Store)

```
ret = SQLExecDirect (hstmt, (SQLTCHAR*)"LCOPY \"public\".\"test\"(c1,c2) FROM 'data.csv' NULL 'null'
DELIMITER \", SQL_NTS);
```

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

```
ret = SQLExecDirect (hstmt, (SQLTCHAR*)"LCOPY \"public\".\"test\"(c1,c2) FROM 'data.csv' NULL 'null'
DELIMITER \",\" DIRECT", SQL_NTS);
```

Using LCOPY with Named Pipes

To use a named pipe, the producer creates the named pipe and sends data through it to the consumer which, in turn, reads the data. In this case, the consumer uses LCOPY to load the data it retrieves from the pipe into the database. The following example shows how the producer and consumer implement LCOPY with a named pipe.

Producer:

```
mkfifo /tmp/pipe_sample
echo "test_data_line2|test_data_line2" > /tmp/pipe_sample
```

Consumer:

```
CREATE TABLE test_named_pipes(
  c1 VARCHAR
);
SELECT IMPLEMENT_TEMP_DESIGN('test_named_pipes');
LCOPY test_named_pipes FROM '/tmp/pipe_sample' DELIMITER '|' DIRECT;
```

Note: If the producer does not send data through the pipe, the connection remains open and Vertica waits for data. This causes LCOPY to hang.

Loading Data Into the WOS/ROS

If you intend to use COPY or LCOPY to load small loads, load it into the WOS and automatically switch to ROS when the WOS is full. By loading small loads into the WOS, you avoid creating too many ROS containers. ***Using the COPY Statement*** (page 61) and ***Using the LCOPY Statement*** (page 62) illustrate how to do this.

Working with ODBC Transactions

Whether auto-commit is turned on or off determines how you execute and commit statements.

Single Statements

If auto-commit is on, the transaction is implicitly committed after a single transaction is executed. You cannot roll back a SQL statement executed in auto-commit mode.

The following example illustrates auto-commit:

```
ret = SQLExecDirect (hstmt, (SQLTCHAR*)sqlStatement.c_str(),
     sqlStatement.length()) ;
```

If auto-commit is off, you need to manually commit the transaction after executing a statement. The following example illustrates this:

```
rc = SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
     ret = SQLExecDirect (hstmt, (SQLTCHAR*)sqlStatement.c_str(),
          sqlStatement.length()) ;
     ret = SQLEndTran (SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
```

Multiple Statements

To establish a transaction that contains two or more statements, you must turn auto-commit off, execute the statements, and then commit the transaction. The following example illustrates this:

```
rc = SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
     ret = SQLExecDirect (hstmt, (SQLTCHAR*)sqlStatement1.c_str(),
          sqlStatement1.length()) ;
     ret = SQLExecDirect (hstmt, (SQLTCHAR*)sqlStatement2.c_str(),
          sqlStatement2.length()) ;
     ret = SQLEndTran (SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
```

Working With Large Result Sets

The following attributes support large result sets, as defined in the file *verticaodbc.h*.

1. The connection attribute ATTR_VERTICA_LRSPATH specifies the client-side location in which the ODBC driver keeps temporary files for large result sets. (The name of these temporary files is vtlrs*.) For example:

   ```
   CHAR * lrspath="/my_disk/tmp";
   ret = SQLSetConnectAttr(conn.dbc, SQL_ATTR_VERTICA_LRSPATH,
       (PTR)lrspath, strlen(lrspath));
   ```

   Linux/Solaris default values:

   - If the environment variable TMPDIR exists and contains the name of an appropriate directory, that variable is used.
   - Otherwise, if the dir argument is non-NULL and appropriate, it is used.
   - Otherwise, "/tmp" is used.

   Windows default values:

   - If the TMP environment variable is defined and set to a valid directory name, that name is used.

- Otherwise, the dir parameter is used as the path.
- If the dir parameter is NULL or set to the name of a directory that does not exist, the current working directory is used.

2 The statement attribute SQL_ATTR_VERTICA_MAX_MEM_CACHE defines the maximum memory for the client storage of a large result set. If the result set size exceeds this value, the ODBC driver uses a temporary file to keep the large result set or uses streaming mode for fetching data from the database server. For example:

```
SQLUINTEGER mem_cache_size=256*1024*1024; // 256 MB
SQLSetStmtAttr(hstmt, SQL_ATTR_VERTICA_MAX_MEM_CACHE,
   (PTR)mem_cache_size, 0);
```

The default value is 64MB.

3 The statement attribute SQL_ATTR_VERTICA_LRS_STREAMING specifies that the ODBC driver uses a temporary file to keep the large result set, or use streaming mode to fetch the large result set from the database server. If the value is TRUE, the ODBC driver pauses the query execution when the memory cache on the client is full and resumes execution of the query after the memory cache rows are retrieved by the ODBC application using SQLFetch. For example:

```
SQLUINTEGER lrs_streaming=1;
SQLSetStmtAttr(hstmt, SQL_ATTR_VERTICA_LRS_STREAMING,
   (PTR)lrs_streaming, 0);
```

If set to FALSE, all rows are fetched from the server and saved in a temporary file. Default value is TRUE.

Note: When SQL_ATTR_VERTICA_LRS_STREAMING is set to TRUE, only one cursor can be open for fetch at a time using the same connection handle.

Temporary Tables and AUTOCOMMIT

When working with temporary tables through ODBC, you must disable AUTOCOMMIT if the temporary table is set to ON COMMIT DELETE ROWS. Otherwise, you will see unexpected behavior, such as rows that should have been deleted on commit remaining in the table.

Examples

This section contains examples of ODBC concepts that are specific to Vertica.

- ***Using Vertica-Specific Parameters With INSERT*** (page 64)
- Tracking Load Status for Batch Inserts and Updates
- Using BATCH_AUTO_COMPLETE

Using Vertica-Specific Parameters With INSERT

This section illustrates the defaults for the following ***parameters*** (page 43) and then shows how to modify them programmatically as part of the INSERT statement:

- SQL_ATTR_VERTICA_DIRECT_BATCH_INSERT
- SQL_ATTR_VERTICA_BATCH_INSERT_RECORD_TERMINATOR

- SQL_ATTR_VERTICA_BATCH_INSERT_NULL

Default Parameters

This batch insert illustrates how the Vertica driver manager converts these default parameters into a COPY statement.

Defaults:

```
rc = SQLSetStmtAttr(hstmt, SQL_ATTR_VERTICA_DIRECT_BATCH_INSERT, (void *)1, 0);
rc = SQLSetStmtAttr(hstmt, SQL_ATTR_VERTICA_ABORT_ON_ERROR, (void *)0, 0);
rc = SQLSetConnectAttr(test.conn.dbc,
SQL_ATTR_VERTICA_BATCH_INSERT_RECORD_TERMINATOR, (void *)"\a\v\b", 3);
rc = SQLSetConnectAttr(test.conn.dbc, SQL_ATTR_VERTICA_BATCH_INSERT_NULL, (void
*)"null", 4);
```

Converts to:

```
COPY "myDimensionTable"  FROM STDIN DELIMITER '|' NULL 'null' RECORD TERMINATOR
'\a\v\b' DIRECT
```

SQL_ATTR_VERTICA_DIRECT_BATCH_INSERT

This example illustrates how to turn off Direct Batch Insert so that a batch is inserted into the WOS instead of the ROS.

```
rc = SQLSetStmtAttr(hstmt, SQL_ATTR_VERTICA_DIRECT_BATCH_INSERT, 0, 0);
```

Converts to:

```
COPY "myDimensionTable"  FROM STDIN DELIMITER '|' NULL 'null' RECORD TERMINATOR
'^G^K^H'
```

SQL_ATTR_VERTICA_BATCH_INSERT_RECORD_TERMINATOR

This example illustrates how to change the record terminator for the batch insert.

```
rc = SQLSetConnectAttr(test.conn.dbc,
SQL_ATTR_VERTICA_BATCH_INSERT_RECORD_TERMINATOR, (void *)"END", 3);
```

Converts to:

```
COPY "myDimensionTable"  FROM STDIN DELIMITER '|' NULL 'null' RECORD TERMINATOR
'END' ABORT ON ERROR
```

SQL_ATTR_VERTICA_BATCH_INSERT_NULL

This example illustrates how to change the null value indicator for the batch insert.

```
rc = SQLSetConnectAttr(test.conn.dbc, SQL_ATTR_VERTICA_BATCH_INSERT_NULL, (void
*)"-0-", 3);
```

Converts to:

```
COPY "myDimensionTable"  FROM STDIN DELIMITER '|' NULL '-0-' RECORD TERMINATOR
'END' ABORT ON ERROR
```

Using JDBC

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

You must first install the JDBC client driver on all client systems that will be accessing the Vertica database. For installation instructions, see ***Installing the Vertica Client Drivers*** (page 10).

For more information about JDBC:

- ***JDBC Driver JavaDoc*** (***../../JDBC/index.html***) (Vertica extensions)
- ***An Introduction to JDBC, Part 1*** (***http://www.onjava.com/pub/a/onjava/excerpt/javaentnut_2/index1.html***)

Creating and Configuring a Connection

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

Importing SQL Packages and Loading the Driver

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;
```

Finally, you'll need to load the Vertica JDBC driver using the `Class.forName()` method:

```
try {
      Class.forName("com.vertica.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.
}
```

Opening the Connection

With SQL packages imported and the driver loaded, 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 user who has access to the database
- The password of the user

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 specific JDBC driver to use, followed by the location of the database.

The last two parameters, username and password, can be given to the JDBC 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);
  ```

You usually want to use the `Properties` object, since it makes it easier to pass additional connection properties to the `getConnection()` method. See ***Connection Properties*** (page 68) and ***Setting and Getting Connection Property Values*** (page 71) for more information about the additional connection properties.

The `getConnection()` throws a `SQLException` if there is any problem establishing a connection to the database, so you will 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 ConnectionExample {
    public static void main(String[] args) {
        // Load JDBC driver
        try {
            Class.forName("com.vertica.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.
        }

        // Create property object to hold username & password
```

```
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                        "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
        } catch (SQLException e) {
            // Could not connect to database.
            System.err.println("Could not connect to database.");
            e.printStackTrace();
            return;
        }

        // Connection is established, do something with it here or
        // return it to a calling method
    }
}
```

Note: When you disconnect a user session, any transactions in progress are automatically rolled back.

Connection Properties

Most of the `Connection` object's parameters can be set either by specifying them in the connection string or `Properties` object passed to the `DriverManager.getConnection()` method, or by using setter and getter methods on the `Connection` object (or `PGConnection` for Vertica-specific methods). The following tables list the properties you can set in the connection string or `Properties` object you use to create the connection. When these properties also have setters and getters, they are listed as well.

General Parameters

Property	Description	Default Value
BinaryDataTransfer	Determines whether binary data is transferred using binary transfer protocol. Enabling this option can improve transfer speed for binary data types such as floats and timestamps.	false
defaultAutoCommit	Controls whether the connection automatically commits transactions. Set this parameter to false to prevent the connection from automatically committing its transactions (this is what you want to do when performing batch loading). ▪ **Setter:** `Connection.setAutoCommit()` ▪ **Getter:** `Connection.getAutoCommit()`	true
KeepAlive	Controls whether the connection uses keepalive packets to ensure the connection remains open during idle periods. This option sets the underlying network socket's SO_KEEPALIVE property.	false
Label	Sets the client label for the connection.	none

Locale	The default 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 properties that can be used to specify a locale. ▪ **Setter:** `PGConnection.setLocale()` ▪ **Getter:** `PGConnection.getLocale()`	en_US@collation=binary (English as in the United States of America)
loginTimeout	The number of seconds Vertica waits for a connection to be established to the database before throwing a `PSQLException`. When set to 0 (the default) the timeout for the connection attempt will be the default TCP timeout.	0
password	The password to use to log into the database.	none
prepareThreshold	The number of times a prepared statement must be executed before the driver switches to using server-side prepared statements. ▪ **Setter:** `setPrepareThreshold()` ▪ **Getter:** `getPrepareThreshold()`	5
ssl	When set to true, uses SSL to encrypt the connection to the server. 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.	false
user	The database username to use to connection to the database.	none

Load Properties

Property	Description	Default Value
batchInsertEnforceLength	Enforces rejection of strings longer than the column width. When set to false, strings that are too long are truncated to the maximum length allowed in the column. When set to true, rows containing strings too long for their columns are rejected. ▪ **Setter:** `PGConnection.setBatchInsertEnforceLength()` ▪ **Getter:** `PGConnection.getBatchInsertEnforceLength()`	false
binaryBatchInsert	When set to true, the JDBC driver sends non-string data to the server as binary, rather than string. ▪ **Setter:** `PGConnection.setBinaryBatchInsert()` ▪ **Getter:** `PGConnection.getBinaryBatchInsert()`	false

directBatchInsert	Determines whether a batch is inserted directly into the ROS (true) or using AUTO mode (false). ▪ **Setter:** `PGConnection.setDirectBatchInsert()` ▪ **Getter:** `PGConnection.getDirectBatchInsert()`	false

Note: The properties use35CopyParameters, use35CopyFormat, and managedBatchInsert available in versions of Vertica earlier than version 4.1 have been deprecated. Setting them has no effect. The abortBatchInsertOnError parameter still works, but is obsolete.

Version 3.5 Data Format Properties

Property	Description	Default Value
batchInsertRecordTerminator	Sets the record terminator string that marks the end of a row of data. ▪ **Setter:** `PGConnection.setBatchInsertRecordTerminator()` ▪ **Getter:** `PGConnection.getBatchInsertRecordTerminator()`	\b\t\f

Large Result Set Properties

Property	Description	Default Value
maxLRSMemory	Sets the size of the buffer in the Vertica driver that is used to temporarily store result sets. **Tip:** To decrease the time it takes the client application to receive the result sets, you could reduce the value of the cache to as little as 256K. ▪ **Setter:** `PGConnection.setMaxLRSMemory()` ▪ **Getter:** `PGConnection.getMaxLRSMemory()`	8388608 (8MB)
streamingLRS	Determines whether the JDBC driver uses a temporary file to keep the large result set, or use streaming mode to fetch the large result set from the database server. If the value is true (the default), the JDBC driver pauses the query execution when the memory cache on the client is full and resumes execution of the query after the memory cache rows are retrieved by the JDBC application. If the value is false, all the data is fetched from the server in one large chunk and is cached on the client side. ▪ **Setter:** `PGConnection.setStreamingLRS()` ▪ **Getter:** `PGConnection.getStreamingLRS()`	true

Additional Properties

The properties listed below can only be set using getters and setters—they cannot be set in the connection string or in the `Properties` object used to create the connection.

Property	Description	Default Value
Transaction Isolation	Sets the isolation level of the transactions that use the connection. See ***Changing the Transaction Isolation Level*** (page 73) for details. ▪ **Setter:** `Connection.setTransactionIsolation()` ▪ **Getter:** `Connection.getTransactionIsolation()`	The current transactio isolation level set by the server.
Read Only	Controls whether the connection is read-only. Any queries attempting to update the database using a read-only connection recieve a `PSQLException`. ▪ **Setter:** `Connection.setReadOnly()` ▪ **Getter:** `Connection.isReadOnly()`	false

For information about manipulating these attributes, see ***Setting and Getting Connection Property Values*** (page 71).

Setting and Getting Connection Property Values

You can set most connection properties when you instantiate the `Connection` object. After you create the the Connection object, you can use getters and setters to access many of the connection properties.

Setting Properties when Connecting

There are two ways you can set connection properties when creating a connection to 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://server:port/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");
myProp.put("binaryBatchInsert", "true");
Connection conn;
try {
```

```
        conn = DriverManager.getConnection(
            "jdbc:vertica://VerticaHost:5433/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 is integer or Boolean.

Getting and Setting Properties after Connecting

Most properties have setters and getters on the `Connection` object that let you get and change the current value of the property after establishing the connection to Vertica. Some setters and getters are defined by the `PGConnection` interface, so you need to cast the `Connection` object to this interface to access them. You need to either use the full qualified name of the interface (`com.vertica.PGConnection`) or import it in order to cast to this interface. The following example demonstrates getting and setting the value of several properties.

```
import java.sql.*;
import java.util.Properties;
import com.vertica.PGConnection;

public class SetConnectionProperties {
    public static void main(String[] args) {
        try {
            Class.forName("com.vertica.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Could not find the JDBC driver class.");
            e.printStackTrace();
            return;
        }
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        // Make batch inserts enforce string lengths rather than
        // truncate.
        myProp.put("batchInsertEnforceLength", "true");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
            // get the state of the auto commit parameter
            System.out.println("Autocommit state: " + conn.getAutoCommit());
            // Change the auto commit state to false
            conn.setAutoCommit(false);
            // Check the state again
            System.out.println("Autocommit state: " + conn.getAutoCommit());
            // Get the batch insert enforce length setting.
            // Need to cast to PGConnection
            System.out.println("BatchInsertEnforceLength state: " +
                    ((PGConnection) conn).getBatchInsertEnforceLength());
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

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

```
Autocommit state: true
Autocommit state: false
BatchInsertEnforceLength state: true
```

Setting the Locale for JDBC Sessions

You set the locale for a session by using the Locale connection property while opening the connection (see ***Creating and Configuring a Connection*** (page 66)), or by calling the `setLocale` setter on the `Connection` object (see ***Setting and Getting Connection Property Values*** (page 71)). For example:

```
((com.vertica.PGConnection)conn).setLocale(ICU_locale_identifier);
```

You can get the locale by calling `getLocale()` on the `Connection` object, which returns the ICU locale identifier as a string:

```
((com.vertica.PGConnection)conn).getLocale();
```

Notes:

- JDBC 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 JDBC driver converts UTF-16 data to UTF-8 when passing to the Vertica server and converts data sent by Vertica server from UTF-8 to UTF-16 .
- JDBC applications should set the correct server session locale by executing the SET LOCALE TO statement in order to get expected collation and string functions behavior on the server. See the SET LOCALE statement in the SQL Reference Manual.

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 Vertica server. The 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. Transactions can only read committed data.

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 has changed mid-transaction. Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations are run at `SERIALIZABLE` isolation 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` class 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` class also defines several other transaction isolation constants (`READ UNCOMMITTED` and `REPEATABLE READ`). Since 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) {
        try {
            Class.forName("com.vertica.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Could not find the JDBC driver class.");
            e.printStackTrace();
            return;
        }
        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: 2
Transaction Isolation Level: 8
```

Creating a Pooling Datasource

A pooling datasource uses pool of connections in order to reduce the overhead of network connections between the client and server. Opening a new connection for each request is more costly to 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.

If you are using a J2EE-based application server in conjunction with Vertica, it should already have a built-in data pooling feature. All that is required is that the application server work with the `ConnectionPoolDataSource` interface implemented by Vertica, which is defined by the JDBC 3.0 standard. 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 Vertica, you can use JDBC's basic support for connection pools through the `PoolingDataSource` class. You use an instance of this class to create your connections to Vertica. As you close connections, they are returned to the pool maintained by the JDBC driver, so that they can be reused by later connection requests.

The following example demonstrates how you can create a pooled connection to a Vertica database using JDBC.

```
import java.sql.*;
import com.vertica.ds.common.BaseDataSource;
import com.vertica.jdbc2.optional.PoolingDataSource;

public class PoolingDSExample {
    public static void main(String[] args) {
        // Create a pooling data source via JDBC
        BaseDataSource pds;
        pds = new PoolingDataSource();
        pds.setServerName("VerticaHost");
        pds.setPortNumber(5433);
        pds.setDatabaseName("ExampleDB");
        pds.setUser("ExampleUser");
        pds.setPassword("password123");
        String firstConnName;  // Save the name of the connection until later

        // Create and initial connection, have it add a table
        // to the DB we can query later.
```

```
        try {
            Connection conn1=pds.getConnection();
            firstConnName=conn1.toString(); // Save the name of the connection
            System.out.println("First connection name: " + firstConnName);
            Statement stmt = conn1.createStatement();
            // Perform some work, to show this is a real connection.
            stmt.executeUpdate("CREATE TABLE pdstest (c1 INTEGER, c2 VARCHAR(20))
                               ");
            stmt.executeUpdate("CREATE PROJECTION pdstest_p (c1, c2) " +
                               "AS SELECT c1, c2 FROM pdstest");
            stmt.executeUpdate("INSERT INTO pdstest VALUES (1, 'Test Row 1')");
            stmt.close();
            conn1.close(); // The connection is closed, and is returned to
                           // the pool
        } catch (SQLException e) {
            e.printStackTrace();
            return;
        }

        // Create another connection and check to see if its name
        // matches the previously used connection.
        try {
            Connection conn2 = pds.getConnection();
            System.out.println("Second connection name: " + conn2.toString());
            System.out.println("Are the connections the same?: "
                    + firstConnName.equalsIgnoreCase(conn2.toString()));
            // If the connections are pooled, the new connection should have
            // reused the old connection. The connection object names should
            // be the same.
            // Drop the previously created table Statement stmt2 =
conn2.createStatement(); stmt2.execute("DROP TABLE pdstest CASCADE");
conn2.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return;
    }
}
```

This example prints the following to the standard output when run:

```
First connection name: Pooled connection wrapping physical connection
com.vertica.jdbc3g.Jdbc3gConnection@2c091cee
Second connection name: Pooled connection wrapping physical connection
com.vertica.jdbc3g.Jdbc3gConnection@2c091cee
Are the connections the same?: true
```

JDBC Data Types

Vertica server supports data type aliases for integer, float and numeric types. However, it processes and reports them as its basic types (INT8, FLOAT8, and NUMERIC), as follows:

Vertica Server Types and Aliases	Vertica JDBC Type
INTEGER INT INT8 BIGINT SMALLINT TINYINT	Int8
DOUBLE PRECISION FLOAT5 FLOAT8 REAL	Float8
DECIMAL NUMERIC NUMBER MONEY	Numeric

If a client application retrieves the values into smaller data types, Vertica JDBC driver does not check for overflows The following code example illustrates this.

```
Statement statement = conn.createStatement();
try {
          statement.executeUpdate("drop table test_all_types cascade");
} catch (Exception e) {
}
statement.executeUpdate("create table test_all_types (" +
            "c0 integer, " +
            "c1 bigint, " +
            "c2 smallint, " +
            "c3 tinyint, " +
            "c4 decimal, " +
            "c5 numeric, " +
            "c6 number, " +
            "c7 money, " +
            "c8 double precision, " +
            "c9 float, " +
            "c10 real" +
            ")");
statement.executeUpdate("create projection test_all_types_p (c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10) " +
            "as select * from test_all_types");
statement.executeUpdate("insert into test_all_types values(111111111111, 222222222222, 3333, 444, "
+
             "55555555555.5555, 66666.66, 65656565.65, 77777777.77, " +
             "88888888888888888.88, 999999.9, 10101010.10101010101010" +
             ")");
ResultSet rs=statement.executeQuery("select * from test_all_types");
ResultSetMetaData md = rs.getMetaData();
while (rs.next()){
             resultStream.println("INTEGER\tgetColumnType()\t"+md.getColumnType(1));
             resultStream.println("INTEGER\tgetColumnTypeName()\t"+md.getColumnTypeName(1));
             resultStream.println("INTEGER\tgetLong()\t"+rs.getLong(1));
             resultStream.println("INTEGER\tgetInt()\t"+rs.getInt(1));
             resultStream.println("INTEGER\tgetShort()\t"+rs.getShort(1));
             resultStream.println("INTEGER\tgetByte()\t"+rs.getByte(1));
             resultStream.println("TINYINT\tgetColumnType()\t"+md.getColumnType(4));
```

```
resultStream.println("TINYINT\tgetColumnTypeName()\t"+md.getColumnTypeName(4));
resultStream.println("TINYINT\tgetLong()\t"+rs.getLong(4));
resultStream.println("TINYINT\tgetInt()\t"+rs.getInt(4));
resultStream.println("TINYINT\tgetShort()\t"+rs.getShort(4));
resultStream.println("TINYINT\tgetByte()\t"+rs.getByte(4));
            resultStream.println("DECIMAL\tgetColumnType()\t"+md.getColumnType(5));
resultStream.println("DECIMAL\tgetColumnTypeName()\t"+md.getColumnTypeName(5));
resultStream.println("DECIMAL\tgetLong()\t"+rs.getLong(5));
resultStream.println("DECIMAL\tgetBigDecimal()\t"+rs.getBigDecimal(5));
resultStream.println("DECIMAL\tgetDouble()\t"+rs.getDouble(5));
            resultStream.println("MONEY\tgetColumnType()\t"+md.getColumnType(8));
resultStream.println("MONEY\tgetColumnTypeName()\t"+md.getColumnTypeName(8));
resultStream.println("MONEY\tgetLong()\t"+rs.getLong(8));
resultStream.println("MONEY\tgetBigDecimal()\t"+rs.getBigDecimal(8));
resultStream.println("MONEY\tgetDouble()\t"+rs.getDouble(8));
            resultStream.println("DOUBLE PRECISION\tgetColumnType()\t"+md.getColumnType(9));
            resultStream.println("DOUBLE
PRECISION\tgetColumnTypeName()\t"+md.getColumnTypeName(9));
            resultStream.println("DOUBLE PRECISION\tgetLong()\t"+rs.getLong(9));
            resultStream.println("DOUBLE PRECISION\tgetBigDecimal()\t"+rs.getBigDecimal(9));
            resultStream.println("DOUBLE PRECISION\tgetDouble()\t"+rs.getDouble(9));
            resultStream.println("DOUBLE PRECISION\tgetFloat()\t"+rs.getFloat(9));
            resultStream.println("REAL\tgetColumnType()\t"+md.getColumnType(11));
resultStream.println("REAL\tgetColumnTypeName()\t"+md.getColumnTypeName(11));
resultStream.println("REAL\tgetLong()\t"+rs.getLong(11));
resultStream.println("REAL\tgetBigDecimal()\t"+rs.getBigDecimal(11));
resultStream.println("REAL\tgetDouble()\t"+rs.getDouble(11));
resultStream.println("REAL\tgetFloat()\t"+rs.getFloat(11));
}
rs.close();
statement.executeUpdate("drop table test_all_types cascade");
statement.close();
```

Output:

```
INTEGER             getColumnType()         -5
INTEGER             getColumnTypeName()     int8
INTEGER             getLong()               111111111111
INTEGER             getInt()                -558038585
INTEGER             getShort()              455
INTEGER             getByte()               -57
TINYINT             getColumnType()         -5
TINYINT             getColumnTypeName()     int8
TINYINT             getLong()               444
TINYINT             getInt()                444
TINYINT             getShort()              444
TINYINT             getByte()               -68
DECIMAL             getColumnType()         2
DECIMAL             getColumnTypeName()     numeric
DECIMAL             getLong()               55555555555
DECIMAL             getBigDecimal()         55555555555.555500000000000
DECIMAL             getDouble()             5.55555555555555E10
MONEY               getColumnType()         2
MONEY               getColumnTypeName()     numeric
MONEY               getLong()               77777777
MONEY               getBigDecimal()         77777777.7700
MONEY               getDouble()             7.777777777E7
DOUBLE PRECISION    getColumnType()         8
DOUBLE PRECISION    getColumnTypeName()     float8
DOUBLE PRECISION    getLong()               88888888888888900
```

```
DOUBLE PRECISION    getBigDecimal()         8.88888888888889E+16
DOUBLE PRECISION    getDouble()             8.8888888888888896E16
DOUBLE PRECISION    getFloat()              8.8888892E16
REAL                getColumnType()         8
REAL                getColumnTypeName()     float8
REAL                getLong()               10101010
REAL                getBigDecimal()         10101010.1010101
REAL                getDouble()             1.01010101010101E7
REAL                getFloat()              1.010101E7
```

Executing Queries Through JDBC

To run a query through JDBC:

1. Connect with the Vertica database. See ***Creating and Configuring a Connection*** (page 66).
2. Run the query.
   The method you use depends on the type of query you want to run:

Executing DDL (Data Definition Language) Queries

To run DDL queries, such as CREATE TABLE and COPY, use the execute method of the `Statement` class. 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'");
```

Note: If your database size exceeds your licensed data allowance, all successful queries from ODBC and JDBC clients return with a status of SUCCESS_WITH_INFO instead of the usual SUCCESS. The message sent with the results contains a warning about the database size. Your ODBC and JDBC clients should be prepared to handle these messages instead of assuming that successful requests always return SUCCESS. See Managing Your License Key in the Administrator's Guide for details.

Executing Queries that Return Result Sets

Use the `Statement` class's `executeQuery` method to execute queries that return a result set of records, such as SELECT. To get the results from the result set, use methods such as `getInt`, `getString`, and `getDouble` depending upon the data types of the results to be returned.

```
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 Vertica JDBC driver does not support scrollable cursors.

Executing DML (Data Manipulation Language) Queries Using executeUpdate

Use the `executeUpdate` method for DML SQL queries such as INSERT, UPDATE and DELETE that do not return a result set of records.

```
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 Vertica JDBC driver does not support multiple SQL statements in the SQL string you pass to the `execute`, `executeUpdate`, or `executeQuery` methods. Attempting to include multiple statements in the SQL string results in an exception.

Loading Data Through JDBC

There are three methods you can use 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.

A primary concern when loading data into Vertica is the data's destination: the Write Optimized Store (WOS) or the Read Optimized Store (ROS). By default, most methods of loading data into Vertica will insert data into the WOS until it fills up, then additional data is inserted directly into ROS containers. This is the best strategy to follow when frequently loading small amounts of data (often referred to as trickle loading). When performing less frequent large data loads (any loads over roughly 100MB of data at once, such as initially loading the database or loading a day's or week's worth of transactions), you want to change this behavior to directly insert data 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 would 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')");
```

There are several drawbacks to this method: you need convert your data to string, and you need to escape your data for special characters. A better way to insert data is to use prepared statements. See ***Batch Inserts Using JDBC Prepared Statements*** (page 81).

Note: The Vertica JDBC driver does not support multiple SQL statements in the SQL string you pass to the `execute`, `executeUpdate`, or `executeQuery` methods. Attempting to include multiple statements in the SQL string results in an exception.

Batch Inserts Using JDBC Prepared Statements

You can load batches of data into 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 contain 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()` (see ***Command Reference for Prepared Statements in JDBC*** (page 83) for a list of these methods). 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 statements.

Behind the scenes, the batch insert is converted into a COPY statement. When the defaultAutoCommit connection parameter is disabled, Vertica uses the same COPY command to load batches until either the transaction is committed, the cursor is closed, or a non-insert statement is executed. If you are loading multiple batches, you should disable the defaultAutoCommit property of the database to make the load more efficient.

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) {
        try {
            Class.forName("com.vertica.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Could not find the JDBC driver class.");
            e.printStackTrace();
            return;
        }
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        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();
            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[] {"123-456-789","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();
            }

            // Batch is ready, execute it to insert the data
            pstmt.executeBatch();
            // Print the resulting table.
            ResultSet rs = null;
            rs = stmt.executeQuery("SELECT CustID, First_Name, " +
                    "Last_Name FROM customers");
            while(rs.next()){
                System.out.println(rs.getInt(1) + " - " + rs.getString(2).trim()
                        + " " + rs.getString(3).trim());
            }

            // Cleanup
            stmt.execute("DROP TABLE customers CASCADE");
            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
```

Command Reference for Prepared Statements in JDBC

This section describes the JDBC API for using prepared statements. You can use prepared statements to supply data to a query at execution time.

Commands

addBatch

Adds the given SQL command to the current list of commands for this Statement object.

Syntax

```
public void addBatch (String sql) throws SQLException
```

Parameters

SQL	Typically this is a static SQL INSERT or UPDATE statement.

Note

You can call the method executeBatch to execute the commands in this list as a batch.

Throws

- SQLException if a database access error occurs or the driver does not support batch updates.
- If an addBatch has been issued against one statement, you will get an error if you try to prepare and addBatch for a second statement without executing the first one.

execute

Executes the given SQL statement.

Syntax

```
boolean execute() throws SQLException
```

Notes

Some prepared statements return multiple results; the execute method handles these complex statements as well as the simpler form of statements handled by the methods executeQuery and executeUpdate.

Returns

The execute method returns a boolean to indicate the form of the first result, as follows:

- True if the first result is a ResultSet object
- False if the first result is an update count or there is no result

To retrieve the result, call either the method getResultSet or getUpdateCount. To move to any subsequent results, call getMoreResults.

Throws

SQLException if a database access error occurs or an argument is supplied to this method

executeBatch

Submits a batch of commands to the database for execution and, if all commands execute successfully, returns an array of update counts.

Syntax

```
public int[] executeBatch() throws SQLException
```

Note

The int elements of the array that is returned are ordered to correspond to the commands in the batch, which are ordered according to the order in which they were added to the batch. The elements in the array returned by the method executeBatch are one of the following:

- A number greater than or equal to zero

  This indicates that the command was processed successfully and provides the number of rows in the database that were affected by the command's execution

- A value of SUCCESS_NO_INFO

  This indicates that the command was processed successfully, but that the number of rows affected is unknown.

Returns

An array of update counts that contains one element for each command in the batch. The elements of the array are ordered in the same order in which commands were added to the batch.

Throws

- SQLException if a database access error occurs or the driver does not support batch updates.
- BatchUpdateException (a subclass of SQLException) if one of the commands sent to the database fails to execute properly or attempts to return a result set.

executeQuery

Executes the given SQL statement, which returns a single ResultSet object.

Syntax

```
public ResultSet executeQuery (String sql) throws SQLException
```

Parameters

SQL	The SQL statement that is sent to the database, typically a static SQL SELECT statement.
sqlType	The SQL type code defined in java.sql.Types.

Returns

- A ResultSet object that contains the data produced by the given query; never null.
- Any semantic or syntactic errors

Throws

SQLException if a database access error occurs or the given SQL statement produces anything other than a single ResultSet object

executeUpdate

Executes the given SQL statement.

Syntax

```
public int executeUpdate (String sql) throws SQLException
```

Parameters

SQL	A SQL INSERT, UPDATE or DELETE statement or a SQL statement that returns nothing

Note

The statement can be an INSERT, UPDATE, or DELETE statement; it can even be a SQL statement that returns nothing, such as a SQL DDL statement.

Returns

- One of the following:
  - The row count for INSERT, UPDATE or DELETE statements
  - A *0* for SQL statements that return nothing
- Any semantic or syntactic errors

Throws

SQLException if a database access error occurs or the given SQL statement produces a ResultSet object

PreparedStatement

The object of PreparedStatement interface represents a pre-compiled SQL statement. A SQL statement is pre-compiled and stored on the server in a PreparedStatement object. This object can then be used to repeatedly execute the statement in an efficient manner.

Note: The setXXX methods for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter. For instance, if the IN parameter has SQL type Integer, then the method setInt should be used.

```
public interface PreparedStatement
extends Statement
```

setBoolean

Sets the designated parameter to a Java boolean value. The driver converts this to a SQL BIT value when it sends it to the database.

Syntax

```
public void setBoolean (int parameterIndex, boolean x) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2, and so on.
x	The parameter value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setDate

Sets the designated parameter to a value. The driver converts this to a SQL DATE value when it sends it to the database.

Syntax

```
public void setDate (int parameterIndex, Date x) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2, and so on.
x	The parameter value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setDouble

Sets the designated parameter to a Java double value. The driver converts this to a SQL DOUBLE value when it sends it to the database.

Syntax

```
public void setDouble (int parameterIndex, double x) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2, and so on.
x	The parameter value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setFloat

Sets the designated parameter to a Java float value. The driver converts this to a SQL INTEGER value when it sends it to the database.

Syntax

```
public final void setFloat (int n, float x) throws SQLException
```

Parameters

n	An int that indicates the parameter number
x	The float value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setInt

Sets the designated parameter to a Java int value. The driver converts this to a SQL INTEGER value when it sends it to the database.

Syntax

```
public void setInt (int parameterIndex, int x) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2, and so on.
x	The parameter value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setLong

Sets the designated parameter to a Java long value. The driver converts this to a SQL BIGINT value when it sends it to the database.

Syntax

```
public void setLong (int parameterIndex, long x) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2, and so on.
x	The parameter value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setNull

Sets the designated parameter to SQL NULL.

Syntax

```
public void setNull (int parameterIndex, int sqlType) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2, and so on.
sqlType	The SQL type code defined in java.sql.Types.

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setString

Sets the designated parameter to a Java String value. The driver converts this to a SQL VARCHAR or LONGVARCHAR value (depending on the argument's size relative to the driver's limits on VARCHAR values) when it sends it to the database.

Syntax

```
public void setString (int parameterIndex, String x) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2, and so on.
x	The parameter value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setTime

Sets the designated parameter to a java.sql.Time value. The driver converts this to a SQL TIME value when it sends it to the database.

Syntax

```
public void setTime (int parameterIndex, Time x) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2...
x	The parameter value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

setTimestamp

Sets the designated parameter to a java.sql.Timestamp value. The driver converts this to a SQL TIMESTAMP value when it sends it to the database.

Syntax

```
public void setTimestamp (int parameterIndex, Timestamp x) throws SQLException
```

Parameters

parameterIndex	The first parameter is 1, the second is 2, and so on.
x	The parameter value

Note

The driver binds the statement parameter, but does not communicate with the server.

Throws

SQLException if a database access error occurs

Statement

The object of Statement interface is used for executing a static SQL statement and returning the results it produces. By default, only one ResultSet object per Statement object can be open at the same time. Therefore, if the reading of one ResultSet object is interleaved with the reading of another, each must have been generated by different Statement objects. All execution methods in the Statement interface implicitly close a statement's current ResultSet object if an open one exists.

```
public interface Statement
```

Directly Loading Batches 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. The Tuple Mover has to perform a moveout on the data in the WOS, while subsequent data is directly written into ROS containers.

To directly load batches into ROS, set the directBatchInsert connection property to true. See ***Setting and Getting Connection Property Values*** (page 71) 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 want to set directBatchInsert 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 `PGConnection.setDirectBatchInsert` method:

```
((PGConnection)conn).setDirectBatchInsert(true);
```

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 on the Server with JDBC*** (page 92) 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 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 closing the statement using close(), 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. You should avoid ending the COPY statement in this manner because any errors from the COPY statement appear the response for the non-insert statement. This 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.

Using Delimiters and Record Terminators for Batch Insert

Delimiters

By default, JDBC uses the delimiter '|' for JDBC batch insert. The driver escapes '|' and '\' in the data, so your application should not escape them.

Record Terminators

Vertica uses '\b\t\f' as the default record terminator. Your application may try to escape this sequence, or you can set another string as record terminator.

To set the batch insert record terminator string for the:

- Connection, use:
  `((PGConnection)dbConn).setBatchInsertRecordTerminator("`*record_terminator*`");`
- Statement, use:
  `((PGStatement)pstmt).setBatchInsertRecordTerminator("`*record_terminator*`");`

  Where *record_terminator* represents your specific record terminator.

Tracking Load Status on the Server

The client can track load status on the server for the last completed database load within the current session by:

- ***Identifying the number of rows that were accepted or rejected*** (page 92).
- ***Identifying which rows were accepted or rejected*** (page 95).

Both methods are useful for determining the status of a load in cases in which data is loaded regardless of any load errors encountered. However, identifying the number of accepted or rejected rows has virtually no performance impact on the server while identifying the status of all the rows in the load slightly affects performance. This occurs because the server sends the row number for each rejected row to the client which, in turn, receives this data. Additionally, the data must be loaded into an array that is supplied by the driver.

Note: Data regarding loads does not persist, and is dropped when a new load is initiated.

Identifying the Number of Accepted and Rejected Rows

In any data load task, one of the basic pieces of information you need is how many rows were successfully loaded into the database and how many were rejected. The standard way of determining how many rows were loaded and rejected is to call the Statement class's `getUpdateCount()` method. This method returns the number of rows that the last executed statement affected which, in the case of an insert insert command, is the number of rows that were inserted.

To find the number of rejected rows, subtract the number of rows that were actually loaded from the number of rows that you attempted to load.

The following example shows how to use `getUpdateCount()` to find the number of rows loaded and rejected by a batch load. In order to trigger a row to be rejected, the example code sets the batchInsertEnforceLength connection parameter to true. Setting this parameter to true forces the last row in the batch to be rejected since its phone number value is too wide to be stored in the database column.

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

import com.vertica.PGConnection;
public class BatchInsertExamplegetUpdateCount {
    public static void main(String[] args) {
        try {
            Class.forName("com.vertica.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Could not find the JDBC driver class.");
```

```
            e.printStackTrace();
            return;
        }
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        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();
            stmt.execute("CREATE TABLE customers (CustID int, Last_Name" +
                    " char(50), First_Name char(50),Email char(50), " +
                    "Phone_Number char(12))");
            // Turn on enforce length. This rejects rows that have a value // too
wide to fit into a column, rather than truncate it.
((PGConnection)conn).setBatchInsertEnforceLength(true);
            // Some dummy data to insert. The final row won't insert because //
the phone number is too long for the phone column, and // batchInsertEnforceLength
is true.
            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",
                    "555-867-5309","555-555-1212",
                    "23123123123123123123123123343"};

            // 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.
int numRowsToLoad = firstNames.length;
            for (int i=0; i < numRowsToLoad; 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();
            }

            // Batch is ready, execute it to insert the data
            pstmt.executeBatch();
```

```
            // Get the number of rows that were affected by the last // command
(which will be the number of rows inserted in this case) int rowCount =
pstmt.getUpdateCount();
            System.out.println("Number of accepted rows = " +
                    rowCount);
            // Number of rejected rows = row we tried to load - rows loaded
            System.out.println("Number of rejected rows = " +
                    (numRowsToLoad - rowCount));
            // Print the resulting table.
            ResultSet rs = null;
            rs = stmt.executeQuery("SELECT CustID, First_Name, " +
                    "Last_Name FROM customers");
            while(rs.next()){
                System.out.println(rs.getInt(1) + " - " + rs.getString(2).trim()
                        + " " + rs.getString(3).trim());
            }

            // Cleanup
            stmt.execute("DROP TABLE customers CASCADE");
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

The output from running the previous example is:

```
Number of accepted rows = 4
Number of rejected rows = 1
1 - Anna Allen
2 - Bill Brown
3 - Cindy Chu
4 - Don Dodd
```

Handling Large Numbers of Accepted and Rejected Rows

Since Vertica loads can contain billions of rows (which is enough to overflow a standard `int`), `PreparedStatement` has a set of methods that return the count of accepted and rejected rows as long integers:

- `PreparedStatement.getLongNumAcceptedRows()` returns a `long` containing the number of rows that Vertica successfully loaded.
- `PreparedStatement.getLongNumRejectedRows()` returns a `long` containing the number of rows that Vertica rejected.

When loading batches, the values returned by these methods depend on when in the load process you call them. Immediately after loading a batch, whether or not defaultAutoCommit is enabled, these methods always report the number of accepted and rejected rows from the latest batch. If you are loading multiple batches with defaultAutoCommit disabled, after the transaction is committed (either explicitly or through closing the cursor or executing a non-insert statement) these methods return the total count of accepted and rejected rows for the entire transaction.

Note: If the statement or copy fails or is canceled after adding a stream of data (`addStreamToCopyIn()`), the results of the methods listed above are not guaranteed. Use these methods only after a successful copy statement.

See the ***Tracking Load Status*** (page 106) example.

Identifying Accepted and Rejected Rows (JDBC)

When row status is sent from the server to the client, the status of each row in the load must be loaded into an array that is supplied by the driver. The following example uses a prepared statement that creates an array and runs a batch method to load the array with the row number and integer 1 (accepted) or -3 (rejected) for each row in the load.

```
ps1 = dbConn.prepareStatement("INSERT INTO test_batch_table(a) VALUES (?)");
for (int i = 1; i <= 10; ++i) {
       ps1.setLong(1, i);
       ps1.addBatch();
}
 int[] counts=ps1.executeBatch();
 int irows;
 for(irows=0;irows<counts.length;++irows)
       resultStream.println("Row "+irows+": status "+counts[irows]);
```

See the ***Tracking Load Status*** (page 106) example.

Bulk Loading Using the COPY Statement

The easiest way to load large amounts of data into Vertica at once (bulk loading) is to use the COPY statement. This statement loads data from a file stored on the host (or a data stream) into a table in the database. COPY has many parameters you can set to specify 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 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 would 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 the database superuser can use the COPY statement to copy a file, so you will need to log in using database administrator's account. If you want to have a non-superuser user bulk load data, you can use COPY to load from a stream rather than a file (see ***Copying Streams*** (page 96)). You can also perform a standard ***batch insert using a prepared statement*** (page 81), 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 the WOS and load directly into ROS.

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

public class CopyExample {
```

```
    public static void main(String[] args) {
        try {
            Class.forName("com.vertica.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Could not find the JDBC driver class.");
            e.printStackTrace();
            return;
        }
        Properties myProp = new Properties();
        myProp.put("user", "dbadmin"); // Must be superuser
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
            Statement stmt = conn.createStatement();
            // Create a table and a projection for the table.
            stmt.execute("CREATE TABLE customers (Last_Name char(50) " +
                    "default '', First_Name char(50),Email char(50), " +
                    "Phone_Number char(50))");
            stmt.execute("CREATE PROJECTION customers_p (Last_Name, " +
                    "First_Name, Email, Phone_Number) AS SELECT Last_Name, " +
                    "First_Name, Email, Phone_Number FROM customers");
            // Use the COPY command to load data. Load directly into ROS, since
            // this load could be over 100MB. Data file is on the node
            // to which we've connected (VerticaHost).
            stmt.execute("COPY customers FROM '/data/customers.txt' " +
                    "DELIMITER '|' DIRECT");
            // Get the number of rows in customers now
            ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM customers");
            while (rs.next()){
                System.out.println("Number of rows in customers = "
                        + rs.getInt(1));
            }

            // Get rid of the table
            stmt.execute("DROP TABLE customers CASCADE");
        } catch(Exception e) {
                System.out.print("Loading 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 rows in customers = 2000000
```

Copying Streams

Vertica supports copying ***individual streams*** (page 97) or ***multiple streams*** (page 98) into the database.

Copying Individual Streams

To copy an individual stream into the database, use the `executeCopyIn` method.

executeCopyIn

Executes a COPY TO query.

Syntax

```
boolean executeCopyIn ( java.lang.String sql,
                        java.io.InputStream inStream)
                        throws java.sql.SQLException
```

Parameters

`sql`	A string that represents the COPY TO query to execute
`inStream`	The input stream that contains the data

Notes

- Use the FROM STDIN clause in the COPY command
- Cast statement to PGStatement

Returns

False.

Throws

java.sql.SQLException if a query execution fails.

Example

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

import com.vertica.PGStatement;
public class CopyStreamExample {
    public static void main(String[] args) {
        try {
            Class.forName("com.vertica.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Could not find the JDBC driver class.");
            e.printStackTrace();
            return;
        }

        // Path to the | delimited data file.
        String inputFileName = "C:\\data\\customers.tbl";
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
```

```
            conn = DriverManager.getConnection(
                "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
            Statement stmt = conn.createStatement();
            // Create a table and a projection for the table.
            stmt.execute("CREATE TABLE customers (Last_Name char(50) " +
                    "default '', First_Name char(50),Email char(50), " +
                    "Phone_Number char(50))");
            stmt.execute("CREATE PROJECTION customers_p (Last_Name, " +
                    "First_Name, Email, Phone_Number) AS SELECT Last_Name, " +
                    "First_Name, Email, Phone_Number FROM customers");
            // Prepare the input file stream
            File inputFile = new File(inputFileName);
            FileInputStream inputStream = new FileInputStream(inputFile);

            // Prepare the query to insert from a stream. Unlike copying from
            // a file on the host, you do not need superuser privileges to
            // copy a stream. All your user account needs in INSERT privileges.
            String copyQuery = "COPY customers FROM STDIN " +
                "DELIMITER '|' NULL '\\\\n' DIRECT;";
            // Execute the CopyIn.
            ((com.vertica.PGStatement)stmt).executeCopyIn(copyQuery ,
                    inputStream);
            System.out.println("Number of accepted rows = " +
                    ((PGStatement)stmt).getNumAcceptedRows());
            System.out.println("Number of rejected rows = " +
                    ((PGStatement)stmt).getNumRejectedRows());
            // Get rid of the table stmt.execute("DROP TABLE customers CASCADE");
        } catch(Exception e) {
                System.out.print("Loading error: ");
                System.out.println(e.toString());
        }
    }
}
```

The result of running the example is shown below (assuming that customers.tbl has 10,000 rows):

```
Number of accepted rows = 10000
Number of rejected rows = 0
```

Copying Multiple Streams

Vertica supports pushing more than one buffer into a single COPY DIRECT by attaching multiple streams one after the other without closing the statement. This is useful for loading several files on a client side into one storage container.

This section:

- ***Provides the API*** (page 98) for copying multiple streams
- ***Provides an example*** (page 100) that demonstrates how to copy multiple streams into a Vertica database.

Command Reference for Multiple Streams

This section describes the JDBC API for copying multiple streams.

Commands

- ***startCopyIn*** (page 99)
- ***addStreamToCopyIn*** (page 99)
- ***finishCopyIn*** (page 99)

startCopyIn

Starts Multiple Streams Copy.

Syntax

```
void startCopyIn ( String sql , InputStream inputStream) throws SQLException
```

Parameters

`sql`	A string that represents the copy statement
`inputStream`	The input stream that contains the data

Throws

SQLException if a SQL exception occurs.

Notes

Start streaming before using the addStreamToCopyIn() and finishCopyIn() methods.

addSreamToCopyIn

Adds a new stream of data to the copy statement.

Syntax

```
void addStreamToCopyIn(InputStream inputStream) throws SQLException
```

Parameters

`inputStream`	The input stream that contains the data

Throws

SQLException if a SQL exception occurs.

Notes

Call this method after streaming has been started using the startCopyIn() method.

finishCopyIn

Finishes the streaming.

Syntax

```
void finishCopyIn() throws SQLException
```

Throws

SQLException if a SQL exception occurs.

Notes

Call this method after all the streams have been added or before streaming is started the next time.

Copy Multiple Streams Example

This example loads multiple streams of data into the Vertica database. In this example, `Date_Dimension.tbl` is a file from which data is to be copied, and the `FileInputStream` object (`fis`) is created by reading this file.

The `startCopyIn()` method call starts streaming. After starting the streaming, addStreamToCopyIn is called 5 times to add new streams (in this case, just new `FileInputStream` instances which contain the same input file). After adding five streams, streaming is finished with `finishCopyIn()`. The number of rows inserted into the database is then printed to the system console. This should be five times the number of rows in the input file (assuming that none of the rows were rejected).

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

import com.vertica.PGStatement;
public class CopyMultipleStreamsExample {
    public static void main(String[] args) {
        try {
            Class.forName("com.vertica.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Could not find the JDBC driver class.");
            e.printStackTrace();
            return;
        }

        // Path to the | delimited data file.
        String inputFileName = "C:\\data\\customers.tbl";
        Properties myProp = new Properties();
        myProp.put("user", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
            Statement stmt = conn.createStatement();
            // Create a table and a projection for the table.
            stmt.execute("CREATE TABLE customers (Last_Name char(50) " +
                    "default '', First_Name char(50),Email char(50), " +
                    "Phone_Number char(50))");
            stmt.execute("CREATE PROJECTION customers_p (Last_Name, " +
                    "First_Name, Email, Phone_Number) AS SELECT Last_Name, " +
                    "First_Name, Email, Phone_Number FROM customers");
            // Prepare the input file stream
            File inputFile = new File(inputFileName);
            FileInputStream inputStream = new FileInputStream(inputFile);
```

```
            // Prepare the query to insert from a stream. Unlike copying from
            // a file on the host, you do not need superuser priviledges to
            // copy a stream. All your user account needs in INSERT privilidges.
            String copyQuery = "COPY customers FROM STDIN " +
                "DELIMITER '|' NULL '\\\\n' DIRECT;";
            // Start the CopyIn process.
            ((com.vertica.PGStatement)stmt).startCopyIn(copyQuery ,
inputStream);
            // Loop 5 times, just adding a new copy of the filestream to
            // the Copyin stream. In your application, you would add
            // different stream sources to the CopyIn.
            for (int x=1; x<5; x++) {
                inputStream = new FileInputStream(inputFile);

((com.vertica.PGStatement)stmt).addStreamToCopyIn(inputStream);
            }
            // Complete the CopyIn process
            ((com.vertica.PGStatement)stmt).finishCopyIn();
            System.out.println("Number of accepted rows = " +
                    ((PGStatement)stmt).getNumAcceptedRows());
            System.out.println("Number of rejected rows = " +
                    ((PGStatement)stmt).getNumRejectedRows());
            // Get rid of the table stmt.execute("DROP TABLE customers CASCADE");
        } catch(Exception e) {
                System.out.print("Loading error: ");
                System.out.println(e.toString());
        }
    }
}
```

The result of running the above code (assuming that `customers.tbl` has 10,000 rows) appears below:

```
Number of accepted rows = 50000
Number of rejected rows = 0
```

Handling Large Result Sets

Large result sets can be fetched either in streaming mode or non-streaming mode. In streaming mode, the data is retrieved in small chunks. The JDBC driver pauses the query execution when the memory cache on the client is full and resumes execution of the query after the memory cache rows are retrieved by the JDBC application.

In non-streaming mode, all the data is fetched from the server in one large chunk and is cached on the client side in the default temporary directory specified by the system property java.io.tmpdir. On UNIX systems, the default location is /tmp or /var/tmp; on Microsoft Windows systems, the default location is typically C:\WINDOWS\TEMP.

By default, large results sets are fetched in streaming mode. To change the mode used for large result sets, modify the streamingLRS connection attribute. If you are using non-streaming mode and you want to change the default buffer size of 8,388,608 (8MB), use the maxLRSMemory connection attribute. See ***Connection Properties*** (page 68) and ***Setting and Getting Connection Property Values*** (page 71).

Note: If large result sets are configured to be fetched in streaming mode and a query is currently running, wait for it to complete before issuing another query. Otherwise, Vertica will throw an error indicating that it is not possible to execute a query while retrieving a large result set. To avoid this issue, use non-streaming mode.

This section:

- ***Provides the API*** (page 102) for handling large result sets.
- ***Provides an example*** (page 103) that illustrates how to use the JDBC API for handling large result sets in Vertica.
- Describes the ***automatic creation of temp files*** (page 104) during processing.

Command Reference for Handling Large Result Sets

This section describes the JDBC API for handling large result sets.

Commands

- ***setStreamingLRS*** (page 102)
- ***getStreamingLRS*** (page 102)
- ***setMaxLRSMemory*** (page 103)
- ***getMaxLRSMemory*** (page 103)

setStreamingLRS

Sets the Streaming Mode to true or false. Enabling the Streaming mode causes data to be retrieved in small chunks that the client application can consume. When the client application needs more data, it is fetched from the server.

By default, streaming mode is enabled. If this mode is disabled, all the data is fetched from the server in one chunk and cached on the client side.

Syntax

```
public void setStreamingLRS(boolean streaming)
```

Parameters

`boolean`	Where *true* enables streaming mode (the default) and *false* disables it.

Notes

If large result sets are configured to be fetched in streaming mode and a query is currently running, wait for it to complete before issuing another query. Otherwise, Vertica returns an error indicating that it is not possible to run a query while retrieving a large result set. To avoid this, use non-streaming mode.

getStreamingLRS

Retrieves the status of streaming mode: true or false.

Syntax

```
public Boolean getStreamingLRS()
```

Returns

Returns the status of streaming mode: true (on) or false (off).

setMaxLRSMemory

Sets the maximum memory size that can be used to store the result set fetched from the database. If result set is greater than the maximum size, it is either stored in a temp file on disk (streaming mode off) or fetched from the server in small chunks (streaming mode on).

Syntax

```
public void setMaxLRSMemory(int bytesmemory)
```

Parameters

bytesmemory	The maximum memory size which can be allocated for Large Result Set. Provide the parameter value in bytes.

getMaxLRSMemory

Retrieves the maximum memory size allocated for the large result set.

Syntax

```
public int getMaxLRSMemory()
```

Returns

Returns the maximum memory size that is set for the large result set. The result is returned in bytes.

Large Result Sets Example

```
public void largeSetExample(Connection conn) throws SQLException{
    String sql = "copy lrs from'lrs.dat' delimiter '|' null '\\n' DIRECT";
    Statement stmt = conn.createStatement();
    stmt.execute(sql);
    Statement stmt1 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                           ResultSet.CONCUR_UPDATABLE);
    // Disables streaming mode.
    ((com.vertica.PGConnection)conn).setStreamingLRS(false);
    // Returns and prints the status of streaming mode.
    Boolean b1 = ((com.vertica.PGConnection)conn).getStreamingLRS();
    System.out.println("Streaming output :" + b1 );
    // Sets the maximum size, in bytes, that can be used to
    // store the result set to 1024*1024*200.
    ((com.vertica.PGConnection)conn).setMaxLRSMemory(1024*1024*200);
    // Returns and prints the maximum size, in bytes, that can be
    // used to store the result set.
    int mem1 = ((com.vertica.PGConnection)conn).getMaxLRSMemory();
```

```
    System.out.println("Max LRS Memory :" + mem1 );
    String sql1 = "select * from lrs limit 1000000";
    stmt1.executeQuery(sql1);
}
```

Temp Files Created During Processing

The JDBC driver creates vtRS*.dmp files in the /tmp directory on the client machine when large result sets are processed, and they are removed when the application using the JDBC driver exits. If the JDBC driver is used by an application that doesn't exit, like Tomcat, these files are left in /tmp, at which point they can accumulate and consume disk space. To relocate them, pass an alternative value to the jvm for the java.io.tmpdir property:

```
-Djava.io.tmpdir=/some/other/directory
```

Re-executing Failed Statements

In mission-critical systems, failed statements are typically executed again.

To re-execute a statement that has failed:

1. Catch the exception.
2. Print the error message for the exception (optional).
3. Establish a new connection.
4. Re-execute the statement.

The following example illustrates how to re-execute a query:

```
try{
    rs = stmt.executeQuery("SELECT COUNT(*) FROM pdstest");
    }catch(Exception e){
        resultStream.println(e.getMessage());
        resultStream.println("conn3 caught exception, reconnecting");
        conn=pds.getConnection();
        stmt = conn.createStatement();
        rs = stmt.executeQuery("SELECT COUNT(*) FROM pdstest");
        }
```

Temporary Tables and AUTOCOMMIT

When working with temporary tables through JDBC, you must disable AUTOCOMMIT if the temporary table is set to ON COMMIT DELETE ROWS. Otherwise, you will see unexpected behavior, such as rows that should have been deleted on commit remaining in the table.

JDBC Examples

This section contains examples of using JDBC with Vertica.

- ***Executing Queries*** (page 105)
- ***Tracking Load Status*** (page 106)

- ***Sample JDBC Application*** (page 107)

Executing Queries

The following sample code demonstrates how to:

- Connect to a Vertica Database using the JDBC driver
- Execute various DDL queries (for example, creating a table and projection)
- Execute various DML queries (for example, Select and Delete)

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

public class ExecutingQueriesExample {
    public static void main(String[] args) {
        try {
            Class.forName("com.vertica.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", "ExampleUser");
        myProp.put("password", "password123");
        Connection conn;
        try {
            conn = DriverManager.getConnection(
                    "jdbc:vertica://VerticaHost:5433/ExampleDB", myProp);
            // Create a statement object for the connection.
            Statement stmt = conn.createStatement();
            // Drop any existing table
            try {
                stmt.execute("DROP TABLE address_book");
            } catch (Exception e) {} // Ignore any table not found error.

            /*
             * Use execute for DDL (Data Definition Language) queries such as
             * Create and Copy. Can also be used for DML queries, in which case,
             * may want to check to see if a ResultSet was created. Returns true
             * if the first result is a ResultSet. Use getResultSet() to
             * retrieve the result set if it exists.
             */
            stmt.execute("CREATE TABLE address_book (Last_Name char(50) " +
                    "default '', First_Name char(50),Email char(50), " +
                    "Phone_Number char(50))");

            /*
             * Use executeUpdate for DML(Data Manipulation Language) queries
             * which do not return a result set, such as INSERT, UPDATE, and
             * DELETE Can also be used for DDL queries Returns an int, the row
             * count after INSERT, UPDATE or DELETE or 0 if its a DDL query
```

```
             */
            stmt.executeUpdate("INSERT INTO address_book "
                    + "VALUES ('Allen', 'Alice', 'tamarrow@example.com',"
                    + " '555-380-6466')");
            stmt.executeUpdate("INSERT INTO address_book (First_Name, Email) "
                    + "VALUES ('Bob','bob@example.com')");

            /*
             * Use executeQuery for DML queries which return result sets such as
             * SELECT. This example lists all of the data inserted earlier.
             */
            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++;
            }

            // Remove the table
            stmt.execute("DROP TABLE address_book");

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Tracking Load Status

This example illustrates how to do the following for both batch inserts and batch updates:

- Identify accepted and rejected rows
- Determine the number of accepted and rejected rows

For an overview, see ***Tracking Load Status on the Server*** (page 92).

Note: If Vertica encounters an error during a batch insert, all the statements except for the error statement are run. If it encounters an error during a batch update, only the statements before the error statement are run.

```
// prepare statement
String sql = "insert into test_batch values(?,?)";
PreparedStatement pstmt = dbConn.prepareStatement(sql);
Int[] counts;
try {
          // add batch
          pstmt.setString(1, "1");
          pstmt.setInt(2, 1);
          pstmt.addBatch();
          // add another batch
          pstmt.setString(1, "3");
```

```
          pstmt.setInt(2, 2);
          pstmt.addBatch();
          // execute batch
          counts = pstmt.executeBatch();
          // print per-row status
          for (int i = 0; i < counts.length; ++i)
                    resultStream.println("Row " + (i + 1) + ": status "
                              + counts[i]);
          // print numbers of accepted and rejected rows
          resultStream.println("Accepted rows: "
                              + ((PGStatement) pstmt).getNumAcceptedRows());
          resultStream.println("Rejected rows: "
                              + ((PGStatement) pstmt).getNumRejectedRows());
          pstmt.close();
} catch (SQLException e) {
          while (e != null) {
                              System.out.println(e.getMessage());
                              e = e.getNextException();
          }
          pstmt.close();
```

Sample JDBC Application

This sample application assumes that it is running on the same machine as the Vertica instance and that your username is `devel`.

```
import java.sql.*;
// Create a table, create a projection, insert a row, // query the table, and drop
the table (including the projection)
class jdbc_test
{
    // Static SQL statements
    static String create_table =
        "CREATE TABLE VTEST (COLUMN_1 CHAR(50));";
    static String create_projection =
        "CREATE PROJECTION VTEST_PROJ (COLUMN_1) AS SELECT COLUMN_1 FROM VTEST;";
    static String insert_row =
        "INSERT INTO VTEST VALUES ('Testing vertica');";
    static String select_row =
        "SELECT * FROM VTEST;";
    static String drop_table =
        "DROP TABLE VTEST CASCADE;";

    public static void main(String args[]) throws Exception
    {
        //try to load the class
        Class.forName("com.vertica.Driver");
        //get a connection to the database
        Connection db = DriverManager.getConnection
            ("jdbc:vertica://VerticaHost:5433/testdb", "devel", "");
        //create a statement object
        Statement st = db.createStatement();
        //execute SQL statements
        st.executeUpdate(create_table);
        st.executeUpdate(create_projection);
```

```
        st.executeUpdate(insert_row);

        // print out the result set
        ResultSet rs = st.executeQuery(select_row);
        while(rs.next())
        {
            System.out.println(rs.getObject(1));
        }

        //clean up
        st.executeUpdate(drop_table);
        rs.close();
        st.close();
        db.close();

    }
}
```

Using ADO.NET

The Vertica driver for ADO.NET allows applications written in C# or other .NET languages to read data from, update, and load data into Vertica databases. It provides a 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 (***VerticaDataReader*** (page 123)) for reading data and ***autocommit*** (page 123) functionality for committing transactions automatically.

For more information about ADO.NET, see:

- ***Overview of ADO.NET*** (***http://msdn.microsoft.com/en-us/library/h43ks021(vs.85).aspx***)
- ***.NET Framework Developer's Guide***

Creating an ADO.NET DSN Entry (optional)

If you want to connect to Vertica through a Data Source Name (DSN), you need to add an entry to the machine configuration file (`machine.config`).

To add an entry to `machine.config`:

1. Back up the file before you modify it. `machine.config` is a .NET Framework core file, so it is imperative that you have a functional copy.

   The default location for *machine.config* varies depending upon whether it is 32 or 64 bits:

   - 32 bit — `C:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG\machine.config`
   - 64 bit — `C:\Windows\Microsoft.NET\Framework64\v2.0.50727\CONFIG\machine.config`
2. Open `machine.config` in a text or XML editor.
3. Locate the section called <connectionStrings>.
4. Use the following format to add an entry for Vertica in this section:

   ```
   <add name="DSNEntryName"
      connectionString="DATABASE=NameOfDatabase;SERVER=ServerAddress;PORT=PortNumber;USER=UserName"
   providerName="vertica">
   </add>
   ```

   Where:

   - name is a unique name to specify the entry. Use alphanumeric characters.
   - connection string is the connection string to the Vertica database. See ***Connecting to the Database*** (page 110).
   - providerName is always "vertica".

For example:

```
<add name="VerticaSql"
   connectionString="DATABASE=ADOREGRESS01;SERVER=10.10.21.245;PORT=5
   433;USER=dba"
providerName="vertica">
</add>
```

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 Vertica server and converts data sent by 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

Creating and Closing Database Connections

This section describes:

- How to use a connection string to ***connect to the database*** (page 110)
- ***Connection string keywords*** (page 111)
- How to ***close a database connection*** (page 114)

Connecting to the Database

To connect to the database:

1. Create a connection using a connection string. See ***Connection String Keywords*** (page 111) for a list of available keywords.

   For example:

   ```
   String connectString =
      "DATABASE=vmartdb;SERVER=cluster_host;PORT=5433";
   ```

   If you are using a DSN, specify the name of the entry you added to Machine.config. (See ***Creating an ADO.NET DSN Entry (optional)*** (page 109).)

   ```
   String connectString = "DSN=DSNEntryName";
   ```

2. Build a Vertica connection object that specifies the connection string you created in step 1.

   C# Example:

   ```
   VerticaConnection _conn = new VerticaConnection(connectString)
   ```

3. Open the connection.

C# Example:

```
_conn.Open();
```

At this point, you can pass the connection to a command object and use the connection to read data from, update, or load data into the database.

Note: If your database is not in compliance with your 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.

See Also

Using SSL: Installing certificates on Windows (page 114)

Connection String Keywords

Connection string keywords control the behavior of a connection.

Keyword	Description	Default Value
Host/Server	Address or name of the server to connect to.	string.Empty
Port	Port where Vertica is running.	5433
Database	Name of the Vertica database to connect to.	string.Empty
UserName	Name of the user or client connecting to the database.	string.Empty
Password	Password of the user or client connecting to the database.	string.Empty
SSL	Specifies whether to use Secure Socket Layer (true) or not (false). See Implementing Security.	false
Timeout	Specifies the number of seconds to wait for a connection.	15
Pooling	Specifies whether to use connection pooling (true) or not (false). 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.	true
MinPoolSize	Minimum number of connections allowed in the connection pool. Only has an effect if Pooling is set to true.	1
MaxPoolSize	Maximum number of connections allowed in the connection pool. Only has an effect if Pooling is set to true.	20
CommandTimeout	The wait time, in seconds, before terminating the attempt to execute a command and generating an error.	20
PreloadReader	Specifies whether to use deprecated cached row reader (true) or not (false). Vertica recommends that you do not use the cached row reader.	false
RowBufferSize	Size in MB for the in-memory buffer for the	100

	BufferedDataReader (page 123).	
CacheDirectory	Directory where ***BufferedDataReader*** (page 123) puts temporary files. If no directory is set, Vertica defaults to the Windows temp directory.	string.Empty user temp
IsolationLevel	Sets the transaction isolation level for 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*** (page 112) 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. Vertica's default isolation level is IsolationLevel.ReadCommitted.	IsolationLevel.Unspecified
DSN	Reads a named connection string from machine.config. See ***Creating an ADO.NET DSN Entry (optional)*** (page 109).	string.Empty

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 Vertica. To set the default transaction isolation level for a connection, use the IsolationLevel keyword in the connection string (see ***Connection String Keywords*** (page 111) 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.

The following example demonstrates:

- getting the connection's transaction isolation level.
- setting the connection's isolation level using the connection string.
- setting the transaction isolation level for a new transaction.

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using vertica;
using verticaTypes;

namespace IsolationLevelExample
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            // Create a connection with the default level
            string connectionString = "DATABASE=ExampleDB;SERVER=VerticaHost;"
                + "PORT=5433;USER=ExampleUser;PASSWORD=password123";
            VerticaConnection conn = new VerticaConnection(connectionString);
            try
            {
                conn.Open();
                // Print current isolation level. Should be "Unspecified" which
means
                // use Vertica's default (ReadCommitted).
                Console.WriteLine("Connection isolation Level: " +
                   conn.IsolationLevel.ToString());
                // Close the connection to finish
                conn.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }

            // Create another connection, setting the isolation level in the
connection
            // string to Serializable.
            string connectionString2 = connectionString
                + ";IsolationLevel=Serializable";
            VerticaConnection conn2 = new VerticaConnection(connectionString2);
            try
            {
                conn2.Open();
                // Print current isolation level. Should be Serializable.
                Console.WriteLine("Connection #2 Isolation Level: "
                    + conn2.IsolationLevel.ToString());

                // Create a transaction with a different isolation level
                VerticaTransaction trans = conn2.BeginTransaction(
                    IsolationLevel.ReadCommitted);
                // Print the transaction's isolation level
                Console.WriteLine("Transaction isolation level: "
                    + trans.IsolationLevel.ToString()); trans.Rollback();

                // Close the connection to finish
                conn2.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
    }
}
```

When run, the example code prints the following to the system console:

```
Authenticate: System.IO.BufferedStream
```

```
Connection isolation Level: Unspecified
Authenticate: System.IO.BufferedStream
Connection #2 Isolation Level: Serializable
Transaction isolation level: ReadCommitted
```

Using SSL: Installing Certificates on Windows

The Vertica ADO.NET driver uses the default Windows key store when looking for its certificates. This is the same key store that Internet Explorer uses, for example.

To import the server and client certificates into the Windows key store:

1. Double-click the certificate.
2. Let Windows determine the key type, and click **Install**.

Since it is necessary to establish a chain of trust, you might need to import the public certificate for your CA (especially if it is a self-signed certificate):

1. 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**.

Closing a Database 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();
```

Querying the Database Programmatically

This section describes how to create queries to do the following programmatically:

- ***Read data from the database*** (page 114)
- ***Insert data into the database*** (page 115)
- ***Load data into the database*** (page 116)
- ***Perform a bulk copy into the database*** (page 117)

Reading Data

To read data:

1. ***Create a connection to the database*** (page 110).
2. Create a command object using the connection.
   ```
   VerticaCommand command = _conn.CreateCommand();
   ```
3. Create a query.

```
command.CommandText =
     "SELECT fat_content " +
     "FROM (SELECT DISTINCT fat_content " +
     "      FROM product_dimension " +
     "      WHERE department_description " +
     "      IN ('Dairy') " +
     "      ORDER BY fat_content) AS food " +
     "LIMIT 5;";
command.Connection = _conn;
```

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.

The following examples call the ForwardOnlyDataReader, which is the default implementations:

```
VerticaDataReader dr = command.ExecuteReader(CommandBehavior.Default);
VerticaDataReader dr = command.ExecuteReader();
VerticaDataReader dr =
   command.ExecuteReader(CommandBehavior.Default,false);
```

The following example specifies the BufferedDataReader:

```
VerticaDataReader dr =
   command.ExecuteReader(CommandBehavior.Default,true);
```

The following example will not work because the behavior parameter is not present:

```
VerticaDataReader dr = command.ExecuteReader(,true);
```

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(" fat content");
Console.WriteLine(" -----------");
int rows = 0;
while (dr.Read())
{
   Int64 content = dr.GetValue(0);
   Console.WriteLine("          " + content);
   ++rows;
}
Console.WriteLine(" (" + rows + " rows)");
```

6 When you're finished, close the data reader to free up resources.

```
dr.Close();
```

Inserting Data

To insert data:

1 ***Create a connection to the database*** (page 110).

2 Create a command object using the connection.

```
VerticaCommand command = _conn.CreateCommand();
```

3 Insert data.

The following is an example of a simple insert. Note that is does not contain a COMMIT statement because ADO.NET provider operates in ***autocommit mode*** (page 123) by default.

```
command.CommandText =
     "INSERT into tabled(field_float8) values (7.4)";
     Int32 rowsAdded = command.ExecuteNonQuery();
```

The following is an example of a simple insert using a parameter.

```
command.CommandText =
     "INSERT into tabled(field_float8) values (:a)";
     command.Parameters.Add(new VerticaParameter(":a",
  verticaDbType.Double));
     command.Parameters[0].Value = 7.4D;
     Int32 rowsAdded = command.ExecuteNonQuery();
```

Loading Data

Loading Data Stored on a Node

To load data that is stored on a database node, you will just use a VerticaCommand object to create a COPY command:

1. ***Create a connection to the database*** (page 110) via 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. Note that is does not contain a COMMIT statement because ADO.NET provider operates in autocommit mode by default.

   ```
   command.CommandText =
        "COPY public.product_dimension FROM
     '/dbadmin/proj/sql/product_data'";
        Int32 rowsAdded = command.ExecuteNonQuery();
   ```

Streaming from the Client via VerticaCopyIn

The VerticaCopyIn class lets you stream data from the client to the database. The syntax for instantiating a VerticaCopyIn object is:

```
new VerticaCopyIn(queryCommand, connection [, fromStream])
```

The following table explains the parameters in the above command.

Parameter

queryCommand	either a VerticaCommand object or a string containing the COPY command to be issued on the database to load the data.
connection	a VerticaConnection object that is connected to the database into which you want to load the data.
fromStream	an optional Stream interface object that will supply the data to be loaded into the database.

Once instantiated, you call the VerticaCopyIn object's `Start()` method to start streaming data. If you supplied the object with a stream using the *fromStream* parameter, all of the data in the stream will be sent to the database automatically. Otherwise, after calling `Start()`, you can write data to the VerticaCopyIn object's `CopyStream` property, which is a VerticaCopyInStream.

Once your data has been streamed to the database, call the `End()` method to successfully end the bulk load. If you want to abandon the bulk load rather than committing it (for example, you encounter an error while you are writing data via the `CopyStream` property), you can call the `Cancel()` method instead of `End()`.

The following example show how to create a procedure that will load data from a file into a database. While it demonstrates using a FileStream object, remember that you can use any class that implements the Stream interface to feed data to VerticaCopyIn.

```
public void BulkLoad(string connectString, string fileName,
                     string tableName, string rejectPath,
                     char recDelimiter)
    {
        FileStream fs = new FileStream(fileName, FileMode.Open);
        StringBuilder loadsql = new StringBuilder();
        // You may want to add RECORD
        // TERMINATOR as a parameter to this function.
        // It should probably be a string, since you
        // can have a multi-character record terminator, as in
        // a file created on Windows, e.g. new lines
        // are \r\n instead of Unix's \n
        loadsql.Append("COPY " + tableName + " FROM STDIN DIRECT DELIMITER '"
                       + recDelimiter + "' REJECTED DATA '" + rejectPath + "'
RECORD TERMINATOR '\r\n';");

        VerticaConnection conn = new VerticaConnection(connectString);
        conn.Open();
        VerticaCommand vc = conn.CreateCommand();
        vc.CommandText = loadsql.ToString();
        vc.CommandType = CommandType.Text;
        VerticaCopyIn v = new VerticaCopyIn(vc, conn, fs);
        v.Start();
        v.End();
    }
```

Performing a Bulk Copy

This example performs a bulk copy from a Vertica database to a SQL Server database.

```
// connection string for local SQL Server database
string connectionString = "Server=(local);Database=vertdb;User ID=dbo;
Integrated Security=True;";
// Get data from the source table as a VerticaDataReader.
VerticaCommand commandSourceData = new VerticaCommand(
    "SELECT product_key, product_version, product_description, sku_number
    FROM product_dimension where product_key < 1000", _conn);
// use a buffered data reader
```

```
VerticaDataReader reader =
    commandSourceData.ExecuteReader(CommandBehavior.Default, true);
// Open the destination connection.
using (SqlConnection destinationConnection =
new SqlConnection(connectionString))
{
    destinationConnection.Open();
    // Set up the bulk copy object.
    // Note that the column positions in the source
    // data reader match the column positions in
    // the destination table so there is no need to
    // map columns.
    using (SqlBulkCopy bulkCopy =
    new SqlBulkCopy(destinationConnection))
    {
        bulkCopy.DestinationTableName = "products";
        try
        {
            // Write from the source to the destination.
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Bulkcopy exception: " + ex.Message);
        }
    }

    // Close the Vertica DataReader.
    reader.Close();
```

Working with Transactions

When you connect to a database using the Vertica ADO.NET Driver, the connection is initially in auto-commit mode. To collect multiple statements into a single transaction, execute the beginTransaction function for the connection. The following code uses an explicit transaction to insert one row each to a dimension table and fact table of the VMart schema.

1 ***Create a connection to the database*** (page 110).

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();
```

Handling 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 a SQL injection issues that can occur when simply creating a sql query from a number of variables.

For example, the following typical query uses the string AZ as a filter.

```
SELECT customer_name, customer_address, customer_city, customer_state
FROM customer_dimension WHERE customer_state = 'AZ';
```

Instead, the query should be written to use placeholders. In the following example, the string AZ is replaced by the parameter placeholder :P1.

```
SELECT customer_name, customer_address, customer_city, customer_state
FROM customer_dimension WHERE customer_state = :P1;
```

To create a parameter placeholder, place either the colon (:) or commercial at (@) 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: If you omit the placeholder indicator (: or @), the server will return an error indicating that <parameter-name> is not a valid column.

For example, the ADO.net code for the prior example would be written as:

```
VerticaCommand c = new VerticaCommand( "SELECT customer_name, customer_address,
customer_city, customer_state
FROM customer_dimension WHERE customer_state = :P1", _conn );
VerticaParameter p = new VerticaParameter( "P1",  DbType.Varchar );
    p.Value = 'AZ';
```

Parameters require that a valid DbType, VerticaDbType, or System type be assigned to the parameter. See SQL Data Types for a mapping of System, Vertica, and DbTypes.

The following example illustrates how to use an insert command with parameters for values.

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using vertica;

namespace HandlingParameterExample
{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = "DATABASE=ExampleDB;SERVER=VerticaHost;"
                + "PORT=5433;USER=ExampleUser;PASSWORD=password123";
            VerticaConnection conn = new VerticaConnection(connectionString);
```

```
            try
            {
                conn.Open();
                VerticaTransaction trans = conn.BeginTransaction();
                VerticaCommand command = new VerticaCommand(
                    "INSERT INTO customers values (:id, :name, :address)", conn);
                // Add objects to parameter collection for the parameters in the
                // command
                command.Parameters.Add(new VerticaParameter("id",
DbType.Int32));
                command.Parameters.Add(new VerticaParameter("name",
DbType.String));
                command.Parameters.Add(new VerticaParameter("address",
DbType.String));
                // Set the direction of the parameters (input or output)
command.Parameters["id"].Direction = ParameterDirection.Input;
                command.Parameters["name"].Direction =
ParameterDirection.Input;
                command.Parameters["address"].Direction =
ParameterDirection.Input;
                // Bind some values to the parameters
                command.Parameters["id"].Value = 1;
                command.Parameters["name"].Value = "Allen, Alice";
                command.Parameters["address"].Value = "10 Main St.";
                // Execute the command to insert bound values
command.ExecuteNonQuery();
                // Bind more values to the parameters
                command.Parameters["id"].Value = 2;
                command.Parameters["name"].Value = "Billings, Bob";
                command.Parameters["address"].Value = "1817 Monroe St.";
                command.ExecuteNonQuery();
                // Commit the transaction to store data trans.Commit();
                // Close the connection to finish
                conn.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
    }
}
```

Note: To see an example of this insert that uses literals instead of parameterized values, see ***AutoCommit Functionality*** (page 123).

Data Types

.NET Data Type	ADO.NET Database Type	Vertica Data Type	API Get Method
Boolean	Boolean	Boolean	GetBoolean()

Byte[]	Binary	Binary VarBinary	GetValue()
Comment: There are no specific get methods for this type. Use GetValue() and cast to a Byte[].			
Byte[]	Byte	Binary VarBinary	GetValue()
Comment: There are no specific get methods for this type. Use GetValue() and cast to a Byte[].			
Datetime	DateTime	Timestamp	GetDateTime()
DateTime	Date	Date	GetDate()
DateTime	Time	Time	GetTime()
DateTimeOffset	DateTimeOffset	TimestampTZ TimeTZ	GetValue()
Comment: There are no specific get methods for this type. Use GetValue() and cast to a DateTimeOffset.			
Decimal	Decimal	Numeric	GetDecimal()
Double	Double	Double	GetDouble()
Guid	Guid	Not Supported	GetGuid()
Int64	Int64	Integer	GetInt16() GetInt32() GetInt64()
Object	Object	N/A	GetValue()
Comment: Any value can be returned as an object type.			
String	AnsiString	Varchar	GetString()
String	AnsiStringFixedLength	Char	GetString()
String	String	Varchar	GetString()
String	StringFixedLengt	Char	GetString()
TimeSpan	Object	Interval	GetInterval()

Using the Vertica Data Adapter

The Vertica data adapter enables a client to exchange data between a data set and a Vertica database. Specifically it can read data from a database into a data set, and then writing changed data from the data set back to the database.

The following example shows how to use a data adapter to read from and insert into a dimension table of the VMart schema.

1 ***Create a connection to the database*** (page 110).

2 Create a data adapter object using the connection and a select statement that retrieves all the table's contents.

```
VerticaDataAdapter da = new VerticaDataAdapter("select * from
   product_dimension where product_key < 10", _conn);
```

3 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, :version, :desc )", _conn);
da.InsertCommand.Parameters.Add(new VerticaParameter("key",
   DbType.Int32));
da.InsertCommand.Parameters.Add(new VerticaParameter("version",
   DbType.Int32));
da.InsertCommand.Parameters.Add(new VerticaParameter("desc",
   DbType.String));
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");
```

4 Create and fill a Data set for this dimension table, and get the resulting DataTable.

```
Data set ds = new Data set();
da.Fill(ds,0,0,"product_dimension");
DataTable dt = ds.Tables[0];
```

5 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);
```

6 Extract the changes for the added rows. The program could print these.

```
      Data set ds2 = ds.GetChanges();
```

7 Send the modifications to the server.

```
Int updateCount = da.Update(ds2, "product_dimension");
```

8 Merge the changes into the original Data set, and mark it up to date.

```
      ds.Merge(ds2);
      ds.AcceptChanges();
```

Vertica Extensions for .NET

The Vertica ADO.NET driver provides the following extensions to .NET:

- ***AutoCommit Functionality*** (page 123)
- ***IDataReader Implementations*** (page 123)

AutoCommit Functionality

By default, the ADO.NET provider operates in autocommit mode. This means that that the commit is executed before any other steps are taken. To disable autocommit for a transaction, use the System.Data.ITransaction object.

The following example shows how to use a transaction to override autocommit.

```
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' )";
// execute the second insert
command.ExecuteNonQuery();

// roll back both inserts
txn.Rollback();
```

IDataReader Implementations

VerticaDataReader is the Vertica implementation of IDataReader, and it provides the lowest common denominator of IDataReader functionality. VerticaDataReader provides two implementations: ForwardOnlyDataReader and BufferedDataReader.

ForwardOnlyDataReader

This implementation reads data as it becomes available from the socket in a forward-only, sequential manner. When waiting for data, ForwardOnlyDataReader waits in blocking mode. This means that if the client is not using the entire data set, it must either Close() the data reader or Cancel() the command object that created it before continuing.

The advantage of this implementation is that it is a highly-efficient means of traversing through the data set. The disadvantage is that it locks up the database for the duration of the read. This means that long-running queries can cause resource constraints.

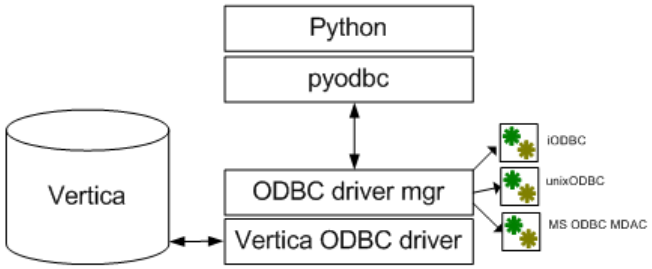
BufferedDataReader

This VerticaDataReader implementation uses a ring-buffer and threading model to keep a large data set in memory. It also allows the buffer to spill to disk if the in-memory portion becomes full. This implementation is useful for moving large volumes of data quickly off the server where it can be run through analytic applications. Use the following keywords in the connection string to modify how the BufferedDataReader behaves:

Keyword	Description	Default Value
RowBufferSize	Size in MB for the in-memory buffer for the BufferedDataReader.	100
CacheDirectory	Directory where BufferedDataReader puts temporary files. If no directory is set, Vertica defaults to the Windows temp directory.	string.Empty user temp

Using Python

Vertica provides an ODBC driver so applications can connect to the Vertica database.



In order to use Python with Vertica, you must install the pyodbc module and a Vertica ODBC driver on the machine where Python is installed. See ***Python Prerequisites*** (page 14).

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 http://www.python.org/download/*** and click an individual release for details. See also ***Python documentation http://www.python.org/doc/***.

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 http://www.activestate.com/activepython/***.

If you need installation instructions for Windows, see ***Using Python on Windows http://docs.python.org/using/windows.html*** at python.org. ***Python on Windows http://diveintopython.org/installing_python/windows.html*** 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 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.

Vertica supports pyodbc version 2.1.6, which requires Python 2.4 or greater, up to 2.6. Vertica does not support Python version 3.x. See ***Python Prerequisites*** (page 14) for additional details.

Download the source distribution from the ***pyodbc Web site http://code.google.com/p/pyodbc/***, unpack it and build it. See the ***pyodbc wiki http://code.google.com/p/pyodbc/w/list*** for instructions.

Note: Links to external Web sites could change between Vertica releases.

External Resources

Python Database API Specification v2.0 http://www.python.org/dev/peps/pep-0249/

Python documentation http://www.python.org/doc/

Python Unicode Support for Wide Characters

The unixODBC and iODBC driver managers differ in how they support wide characters. The SQLWCHAR data type is defined as wchar_t type on Windows (`typedef wchar_t SQLWCHAR;`). On Windows, wchar_t is 16 bits wide, and on Linux, wchar_t is 32 bits wide. The unixODBC driver follows the Windows ODBC API precisely and defines SQLWCHAR as 2-byte characters. However, the iODBC driver defines SQLWCHAR as wchar_t, which expects and returns 4-byte characters.

If an application does not follow the rules set by the driver manager, it can result in incorrect sizing for the SQLWCHAR data type. To handle different sizes of SQLWCHAR using unixODBC or iODBC, Vertica provides two ODBC configuration parameters: `WideCharSizeIn` and `WideCharSizeOut`.

If your system uses UCS-2:

- WideCharSizeIn = 2

WideCharSizeOut = 4 If your system uses using UCS-4:

- WideCharSizeIn = 4
- WideCharSizeOut = 4

To change the Vertica ODBC configuration parameter, specify the setting in the odbc.ini file or at a connection string.

The following code fragment illustrates a connection string that connects to the database and specifies the type of unicode to use; for example, if UCS is less than 4, set WideCharSizeIn to 2. If the system uses UCS-4, set WideCharSizeIn to 4:

```
if sys.maxunicode<65536:
    WideCharSizeIn="WideCharSizeIn=2"
else:
    WideCharSizeIn="WideCharSizeIn=4"
    connection_string = "DSN="+args[0]+";WideCharSizeOut=4;"+WideCharSizeIn
```

```
    cnxn = pyodbc.connect(unicode(connection_string.encode('utf-8'),'utf-8'),
ansi=(not options.unicode), unicode_results=(options.unicode))
```

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 http://www.unixodbc.org/

iODBC Web site http://www.iodbc.org/dataspace/iodbc/wiki/iODBC/

Querying the Database Using Python

The example session below uses pyodbc with the Vertica ODBC driver to connect Python to the Vertica database.

iODBC Users:

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`, and create temporary projections:
   ```
   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)")
   cursor.execute("SELECT IMPLEMENT_TEMP_DESIGN('TEST')")
   ```
2. Insert records into table `TEST`:
   ```
   cursor.execute("INSERT into test
   values(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','1901-01-01
09:00:09','t')
cursor.execute("INSERT into test values(?,?,?,?,?,?,?)",
               values[0], values[1], values[2], values[3], values[4],
  values[5], values[6])
```

4 Create a load file called `load.dat`:

```
load_file = open('/tmp/load.dat', 'w')
load_file.write('3,3.34,abcdefg1234567890,1901-01-01,23:12:34,1901-0
  1-01
09:00:09,t')
load_file.close()
```

5 Insert records using the LCOPY command (bulk insert from file):

```
cursor.execute("LCOPY TEST FROM '/tmp/load.dat' DELIMITER ',' ")
```

6 Select data from the `TEST` table:

```
cursor.execute("SELECT * FROM TEST")
rows = cursor.fetchall()
for row in rows:
    print row
```

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, 2.2799999999999998, 'abcdefg1234567890',
  datetime.date(1901, 1, 1), datetime.time(23, 12, 34),
  datetime.datetime(1901, 1, 1, 9, 0, 9), '1') (3L, 3.3399999999999999,
  'abcdefg1234567890', datetime.date(1901, 1, 1), datetime.time(23,
  12, 34), datetime.datetime(1901, 1, 1, 9, 0, 9), '1')
```

7 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"
```

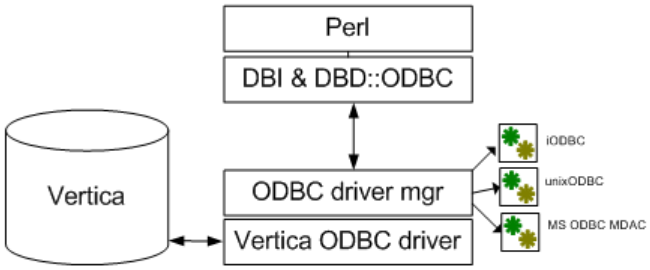
```
Vertica recommends that you name your constraints.
```

See Also

Loading Data Through ODBC (page 50)

Using Perl

Vertica provides an ODBC driver so applications can connect to the Vertica database.



In order to use Perl with Vertica, you must install the Perl driver modules (DBI and DBD::ODBC) and a Vertica ODBC driver on the machine where Perl is installed. See ***Perl Prerequisites*** (page 15).

Perl on Linux

Most Linux distributions come with Perl 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 ***Perl Web site http://www.perl.org/get.html*** for downloads. See also ***Perl documentation http://www.perl.org/docs.html***.

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

Although Perl is typically used on UNIX operating systems, it runs on Windows, as well. Perl is not preinstalled on Windows operating systems. ActiveState distributes a free Windows installer for Perl called ***ActivePerl http://www.activestate.com/activeperl/***. Download the installer and follow the steps in the Install Wizard.

A ***Perl tutorial http://perl.about.com/od/gettingstartedwithperl/ss/installperlwin_2.htm*** on the About.com Web site walks you through using the ActivePerl install package.

The Perl Driver Modules (DBI and DBD::ODBC)

The native perl driver is not supported.

Before you can connect to Vertica using Perl, you need the Perl driver modules. These modules communicate with iODBC/unixODBC driver on UNIX operating systems or the ODBC Driver Manager for Windows operating systems.

DBI (Database Interface) is the standard database interface module for Perl and requires a DBD::* driver module as a translator to talk to the database. Both modules are required to run Perl.

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`)

Download Perl drivers from the ***CPAN modules downloads http://www.cpan.org/modules/by-category/07_Database_Interfaces/DBD/*** site, unpack, and build them. See their accompanying readme files for instructions.

Note: Links to external Web sites could change between Vertica releases.

Perl Unicode Support

Perl does not implement the Unicode standard or all of the accompanying technical reports; however, Perl supports many Unicode features. If you want to understand how Perl implements Unicode support, see the ***Perl Unicode tutorial, perlunitut http://perldoc.perl.org/perlunitut.html***.

Note: DBD::ODBC does not compile with iODBC in Unicode mode, so if you use iODBC, your system uses ANSI. If you want to use Unicode, you must use unixODBC.

Querying the Database Using Perl

The example session below uses DBI with the Vertica ODBC driver to connect Perl to the Vertica database.

1 Call Perl and instruct the program to warn on uninitialized variables, restrict unsafe constructs, and to use the DBI and Data::Dumper modules:

```
#!/bin/perl -w
use strict;
use DBI;
use Data::Dumper;
```

2 Open a database connection:

```
my $db = DBI->connect("dbi:ODBC:VerticaSQL",undef, undef, {AutoCommit
   => 1, });
```

3 Create a table called `TEST` and create temporary projections:

```
$db->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)");
$db->do("SELECT IMPLEMENT_TEMP_DESIGN('TEST')");
```

4 Insert records into `TEST`:

```
$db->do("INSERT into test
values(1,1.1,'abcdefg1234567890','1901-01-01','23:12:34','1901-01-01
09:00:09','t')");
```

5 Insert records using bind values:

```
my @values =
(2,2.28,'abcdefg1234567890','1901-01-01','23:12:34','1901-01-01
09:00:09','t');
my $sth = $db->prepare_cached("INSERT into test
   values(?,?,?,?,?,?,?)"); $sth->execute(@values);
```

6 Create a load file called `load.dat`:

```
open(FH, ">", '/tmp/load.dat');
print FH '3,3.34,abcdefg1234567890,1901-01-01,23:12:34,1901-01-01
09:00:09,t';
close(FH);
```

7 Insert records using bind LCOPY command (bulk insert from file):

```
$db->do("LCOPY TEST FROM '/tmp/load.dat' DELIMITER ',' ");
```

8 Select data from table `TEST`:

```
$sth = $db->prepare_cached("SELECT * FROM TEST"); my $res =
   $sth->execute();
print Dumper( $sth->fetchall_arrayref() );
```

The following is the example output:

```
$VAR1 = [
          [
            '1',
            '1.1',
            'abcdefg1234567890',
            '1901-01-01',
            '23:12:34',
            '1901-01-01 09:00:09',
            '1'
          ],
          [
            '2',
            '2.28',
            'abcdefg1234567890',
            '1901-01-01',
            '23:12:34',
            '1901-01-01 09:00:09',
            '1'
          ],
          [
            '3',
            '3.34',
            'abcdefg1234567890',
            '1901-01-01',
            '23:12:34',
            '1901-01-01 09:00:09',
            '1'
          ]
        ];
```

9 Drop the `TEST` table and its associated projections, and close the database connection:

```
$db->do("DROP TABLE TEST CASCADE");
$sth->finish;
$db->disconnect;
```

See Also

Loading Data Through ODBC (page 50)

Using vsql

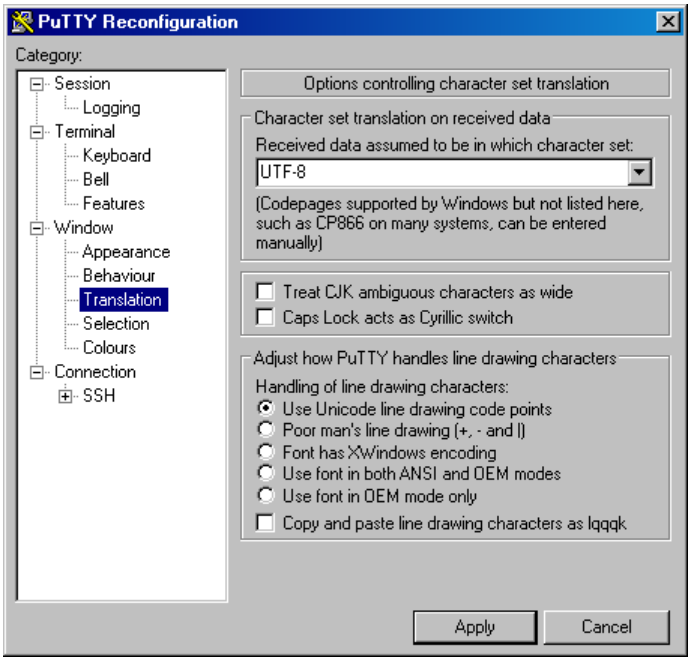
vsql is a character-based, interactive, front-end utility that lets you type SQL statements and see the results. It also provides a number of meta-commands and various shell-like features that facilitate writing scripts and automating a variety of tasks.

You can connect to vsql from the:

- ***Administration Tools*** (page 135)
- ***Linux command line*** (page 136)

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:



  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.

  1. Before connecting to the database, specify that you want to use `less` for query output:

  ```
  $ export PAGER=less
  ```

  2. Connect to the database.

3. Query a wide table:

```
=> select * from wide_table;
```

4. 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:

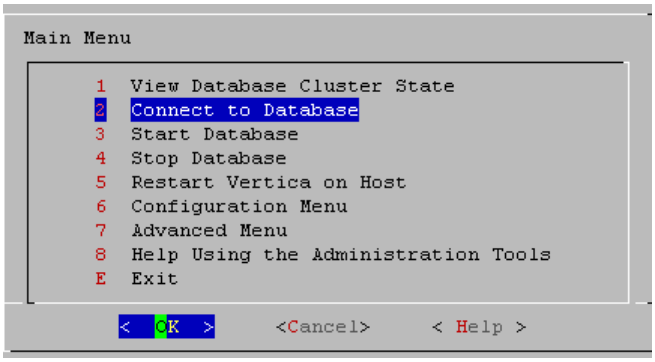
```
# ps -ef | grep vertica
    ⋮
fred   2401     1  0 06:02 pts/1    00:00:00 /opt/vertica/bin/vsql -p
   5433 -h test01_site01 quick_start_single
    ⋮
# kill -9 2401
```

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. (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.



4. Supply the database password if asked:

   ```
   Password:
   ```
5. The Administration Tools connect to the database and transfer control to vsql.

   ```
   Welcome to the vsql, Vertica_Database v5.0.x interactive terminal.
   Type:  \h for help with SQL commands
          \? for help with vsql commands
          \g or terminate with semicolon to execute query
          \q to quit
   vmartdb=>
   ```

Note: See ***Meta-Commands*** (page 141) 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*** (on page 136)
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*** (see "***w password***" on page 140) 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.

? --help

`-? --help` displays help about vsql command line arguments and exits.

a --echo-all

`-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` (page 160) to `all`.

A --no-align

`-A --no-align` switches to unaligned output mode. (The default output mode is aligned.)

c command --command command

`-c command --command command` runs one command and exits. This is useful in shell scripts. The command must be either a command string that is completely parsable 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.

d dbname --dbname dbname

`-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

`-e --echo-queries` copies all SQL commands sent to the server to standard output as well. This is equivalent to setting the variable `ECHO` (page 160) to `queries`.

E

`-E` displays queries generated by internal commands.

f filename --file filename

`-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` (see "`i FILE`" on page 152).

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|FEDEX
111|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

`-F separator --field-separator separator` specifies the field separator for unaligned output (default: "|") (-P fieldsep=). (See `-A --no-align` (page 137).) This is equivalent to `\pset (page 154) fieldsep` or `\f` (see "`f [ string ]`" on page 152).

h hostname --host hostname

`-h hostname --host hostname` specifies the host name of the machine on which the server is running.

Notes:

- If you are using client authentication with a connection method of either "gss" or" "krb5" (Kerberos), you are required to specify -h hostname.
- If you are using client authentication with a "local" connection type specified, avoid using `-h hostname` if you want to match the client authentication entry.

H --html

`-H --html` turns on HTML tabular output. This is equivalent to `\pset (page 154) format html` or the `\H` (see "H" on page 152) command.

l --list

`-l --list` returns all available databases, then exits. Other non-connection options are ignored. This command is similar to the internal command `\list`.

n

`-n` disables command line editing.

o filename --output filename

`-o filename --output filename` writes all query output into file *filename*. This is equivalent to the command `\o` (page 154).

p port --port port

`-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

`-P assignment --pset assignment` lets you specify printing options in the style of `\pset` (page 154) 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

`-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` (page 137) option. Within vsql you can also set the `QUIET` (page 161) variable to achieve the same effect.

R separator --record-separator separator

`-R separator --record-separator separator` uses *separator* as the record separator. This is equivalent to the `\pset (page 154) recordsep` command.

s --single-step

`-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

`-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

`-t --tuples-only` disables printing of column names, result row count footers, and so on. This is equivalent to the `\t` (see "`t`" on page 157) command.

T table_options --table-attr table_options

`-T table_options --table-attr table_options` allows you to specify options to be placed within the HTML `table` tag. See `\pset` (page 154) for details.

U username --username username

`-U username --username username` connects to the database as the user *username* instead of the default.

v assignment --set assignment --variable assignment

`-v assignment --set assignment --variable assignment` performs a variable assignment, like the `\set` (see "`set [ NAME [ VALUE [ ... ] ] ]`" on page 156) 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

`-V --version` prints the vsql version and exits.

w password

`-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

`-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` (see "`c (or \connect) [ dbname [ username ] ]`" on page 144).

x --expanded

`-x --expanded` enables extended table formatting mode. This is equivalent to the command `\ x` (see "`x`" on page 158).

X, --no-vsqlrc

`-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 Vertica vsql executable image on a non-cluster Linux host to connect to a 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 5 64-bit, and SUSE 64-bit*** (page 17) 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 Vertica server RPM in order to get vsql. Download the appropriate rpm package from the Vertica ***Download Website http://www.vertica.com/v-zone/download_vertica*** 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_Vertica` script on the non-cluster host in order to use vsql.

Notes

- Use the same ***command line options*** (on page 136) 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.

```
=> \?

General
   \c[onnect] [DBNAME|- [USER]]
                  connect to new database (currently "vmartdb")
   \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*** (page 154) for a more general solution.

`\a` is similar to the command line option ***-A --no-align*** (page 137), 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 (page 154) 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 ***\!*** (see "***! [COMMAND]***" on page 142)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 [PATTERN] meta-command lists all tables in the database and returns their schema, table name, kind (e.g., table), and owner. For example, the following is the result of \d in the vmart schema.

```
vmartdb=> \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)
```

If you provide the table name as an argument, the result shows the schema name, table name, column name, column data type, data type size, default value, whether it is Nullable or has a NOT NULL constraint, and whether there is a primary key or foreign key constraint.

```
vmartdb=> \d inventory_fact
                                                    List of Fields by Tables
 Schema |     Table      |     Column      | Type | Size | Default | Not Null | Primary Key |
Foreign Key
--------+----------------+-----------------+------+------+---------+----------+-------------+----
-------------------------------------
 public | inventory_fact | date_key        | int  |    8 |         | t        | f           |
public.date_dimension(date_key)
 public | inventory_fact | product_key     | int  |    8 |         | t        | f           |
public.product_dimension(product_key)
 public | inventory_fact | product_version | int  |    8 |         | t        | f           |
```

```
public.product_dimension(product_version)
 public | inventory_fact | warehouse_key   | int  |    8 |         | t        | f          |
public.warehouse_dimension(warehouse_key)
 public | inventory_fact | qty_in_stock    | int  |    8 |         | f        | f          |
(5 rows)
```

You can also use the question mark [?] argument to replace a single character. For example, the ? argument replaces the last character in the 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)
```

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** (see "*z*" on page 158).

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 V_CATALOG_SEQUENCES`, as described in the SQL Reference Manual.

dS [PATTERN]

The `\dS [ PATTERN ]` meta-command (uppercase S) returns all system table (monitoring API) names. You can get identical results issuing `SELECT * FROM system_tables`;

```
vmartdb=> \ds
                                   List of tables
  Schema   |       Name          |  Kind  |            Description
-----------+---------------------+--------+-----------------------------------
-------------
 v_catalog | columns             | system | Table column information
 v_catalog | dual                | system | Oracle(TM) compatibility DUAL table
 v_catalog | foreign_keys        | system | Foreign key information
 v_catalog | grants              | system | Grant information
 v_catalog | passwords           | system | User password history and password reuse
policy
 v_catalog | primary_keys        | system | Primary key information
```

```
 v_catalog | profile_parameters | system | Profile Parameters information
 v_catalog | profiles           | system | Profile information
 v_catalog | projection_columns | system | Projection columns information
 v_catalog | projections        | system | Projection information
...
 v_monitor | host_resources     | system | Per host profiling information
 v_monitor | load_streams       | system | Load metrics for each load stream on
each node
 v_monitor | locks              | system | Lock grants and requests for all nodes
 v_monitor | node_resources     | system | Per node profiling information
...
```

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
 Numeric
 Time
 TimeTz
 Timestamp
```

```
 TimestampTz
 Varbinary
 Varchar
(14 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       | my_seqview              | view  | 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
(16 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` (see "`i FILE`" on page 152) 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*** (page 154) command to redirect your query output you might want to use ***\qecho*** (page 156) 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` (page 154) for a generic way of setting output options.

g

The `\g` meta-command sends the query in the input buffer (see ***\p*** (see "***p***" on page 154)) 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 meta-command (see "***o***" on page 154)

H

`\H` toggles HTML query output format. This command is for compatibility and convenience, but see `\pset` (page 154) about setting other output options.

h \help [command]

`\h \help [ command ]` gives syntax help on the specified SQL command. If *command* is not specified, vsql lists all the commands for which syntax help is available. If *command* is an asterisk (`*`), syntax help on all SQL commands is shown.

Note: To simplify typing, commands that consists of several words do not have to be quoted. For example:
`\help alter table.`

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*** (page 160) 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_GB
INFO:  Locale: 'en_GB'
INFO:    English (United Kingdom)
INFO:  Short form: 'LEN'
```

You can also use the short form of an ICU locale identifier:

```
\locale LEN
INFO:  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 Vertica 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*** (page 156).

See Also

\g meta-command (page 152)

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. They are prompted for their old password, their new password, and then their new password again to confirm.

The superuser can change the password of another user by supplying the username. The 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*** (page 169) section.

Tip: There are various shortcut commands for \pset. See **\a** (see "***a***" on page 143), **\C** (see "***C [STRING]***" on page 144), **\H** (see "***H***" on page 152), **\t** (see "***t***" on page 157), **\T** (see "***T [STRING]***" on page 157), and **\ *x*** (see "***x***" on page 158).

q

`\q` quits the vsql program.

qecho [STRING]

`\qecho [ STRING ]` is identical to `\echo` (see "`echo [ STRING ]`" on page 151) except that the output is written to the query output stream, as set by **\o** (see "***o***" on page 154).

r

`\r` resets (clears) the query buffer.

For example, run the **\p** (see "***p***" on page 154) 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.

It no argument is provided, \set lists all internal variables; for example:

```
vmartdb=> \set
```

```
VERSION = 'Vertica Analytic Database v4.1.6-0'
AUTOCOMMIT = 'off'
VERBOSITY = 'default'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
ROWS_AT_A_TIME = '1000'
DBNAME = 'vmartdb'
USER = 'dbadmin'
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*** (page 158).
- 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*** (page 157) for examples.
- To unset a variable, use the ***\unset*** (page 158) command.

t

`\t` toggles the display of output column name headings and row count footer. This command is equivalent to `\pset (page 154) 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 (page 154) tableattr` *table_options*.

timing

`\timing` toggles 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.

```
=> \unset ROWS_AT_A_TIME
=> select * from fact limit 100000;
Time: First fetch (100000 rows): 220.286 ms. All rows formatted: 231.778 ms
```

In this case, the database retrieved all 100000 rows in 220 ms and spent 11 ms formatting them.

Note: Use ***\unset*** (page 158) with the `ROWS AT A TIME` (page 156) parameter to get results comparable to Vertica 2.5.

See Also

set (page 156)

unset [NAME]

`\unset [ NAME ]` unsets (deletes) the internal variable *name* that was set using the ***set*** (page 156) 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 (page 154) 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*** (see "***dp [PATTERN]***" on page 148)

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` (see "`set [ NAME [ VALUE [ ... ] ] ]`" on page 156):

```
testdb=> \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:

```
testdb=> \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` (see "`unset [ NAME ]`" on page 158).

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 (`:`).

```
testdb=> \set fact 'my_table'
testdb=> SELECT * FROM :fact;
```

would then query the table `my_table`. The value of the variable is copied literally, 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.

AUTOCOMMIT

When AUTOCOMMIT is set 'on', each SQL command is automatically committed upon successful completion; for example:

`\set` (see "***set [NAME [VALUE [...]]]***" on page 156) 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` (see "`a --echo-all`" on page 137). If set to `queries`, vsql merely prints all queries as they are sent to the server. The switch for this is `-e` (see "`e --echo-queries`" on page 137).

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 Vertica internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch `-E` (see "`E`" on page 137).)

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 non-interactive scripts encounter an error, such as a malformed SQL command or internal meta-command, processing continues. This has been the traditional behavior of vsql but it is sometimes not desirable. If this variable is set, script processing immediately terminates. If the script was called from another script it terminates in the same manner. If the outermost script was not called from an interactive vsql session but rather using the `-f` (see "`f filename --file filename`" on page 137) option, vsql returns error code 3, to distinguish this case from fatal error conditions (error code 1).

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*** (page 162) below.

QUIET

This variable is equivalent to the command line option `-q` (see "`q`" on page 156). It is probably not too useful in interactive mode.

SINGLELINE

This variable is equivalent to the command line option `-S` (see "`S --single-line`" on page 140).

SINGLESTEP

This variable is equivalent to the command line option `-s` (page 139).

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:** This feature was adapted from tcsh.

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.

Environment

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*** (see "***pset NAME [VALUE]***" on page 154) command.

PGDATABASE

Default connection database

PGHOST
PGPORT
PGUSER

Default connection parameters

VSQL_EDITOR
EDITOR
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 ***\!*** (see "***! [COMMAND]***" on page 142) command.

TMPDIR

Directory for storing temporary files. The default is `/tmp`.

Locales

The default terminal emulator under Linux is gnome-terminal, although xterm can also be used.

Vertica 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.

  Vertica recommends the following (or whatever other .UTF-8 locale setting you find appropriate):

  ```
  export LANG=en_US.UTF-8
  ```
- The vsql ***\locale*** (see "***locale***" on page 153) 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** (page 143)	***-A*** (page 137) or --no-align
Show only tuples, disabling column headings and row counts.	**\t** (page 157)	***-t*** (page 140) or --tuples-only
Set the field separator character.	**\pset** (page 154) fieldsep	***-F*** (page 138) or --field-separator
Send output to a file.	**\o** (page 154)	***-o*** (page 139) or --output
Specify a SQL statement to execute.	N/A	***-c*** (page 137) 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 explicitely 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 a 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 Vertica databases. This technique is similar to the technique explained in ***Exporting Data via vsql*** (page 166), 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.

The easiest way to copy using vsql is to log into a node of the target database, then issue a vsql command that connects to the source 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_fact"  \
| vsql -U username -w passwd -d vmart -c "COPY store.store_sales_fact FROM STDIN DELIMITER '|';"
```

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 Vertica databases. One way of monitoring the progress of the copy operation is to use a utility such as ***Pipe Viewer*** (***http://www.ivarch.com/programs/pv.shtml***) 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*** (***http://www.ivarch.com/programs/pv.shtml***) page for download packages and instructions. Vertica 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_fact"  \
| 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 DELIMITER '|';"
```

While running, the above command displays a progress bar that looks like this:

```
0:00:39 [12.6M/s] [============================>                              ] 50% ETA 00:00:40
```

Notes for Windows Users

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 German; replace it with your value.

  Note: If you use Cygwin, you can put this command in `/etc/profile`.
- Set the console font to "Lucida Console", because the raster font does not work with the ANSI code page.

Output Formatting Examples

The first example shows how to spread a command over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(> first integer not null default 0,
testdb(> second text) testdb-> ;
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)
testdb=> \pset border 0
Border style is 0.
testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)
testdb=>
\pset border 1
Border style is 1.
testdb=> \pset format unaligned
Output format is unaligned.
testdb=> \pset fieldsep ","
Field separator is ",".
testdb=> \pset tuples_only
Showing only tuples.
testdb=> SELECT second, first FROM my_table; one,1
```

```
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
testdb=> \a \t \ x
Output format is aligned.
Tuples only is off.
Expanded display is on.
testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first | 1
second | one
-[ RECORD 2 ]-
first | 2
second | two
-[ RECORD 3 ]-
first | 3
second | three
-[ RECORD 4 ]-
first | 4
second | four
```

Writing Queries

Queries are database operations that retrieve data from one or more tables or views. In Vertica, the top-level `SELECT` statement is the query, and a query nested within another SQL statement is called a subquery.

Vertica is designed to run the same SQL standard queries that run on other databases. However, there are some differences between Vertica queries and queries used in other relational database management systems.

The 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 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

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 in the Concepts Guide

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 Vertica, with additional details in later sections.

Note: Many of the examples in this chapter use the VMart schema. For information about other 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*** (page 198). See also Multicolumn Subqueries.

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*** (page 204).

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*** (page 175).

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*** (page 215) 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 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*** (page 187). You can even use them to ***update or delete*** (page 189) 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*** (page 197).

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)

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

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 t1))
```

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 (page 197)

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);
```

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 (page 197)

Comparison Operators

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 (page 197)

LIKE Pattern Matching

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)
```

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 (page 197)

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 (page 197)

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, ...);
```

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

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, 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 (page 197)

IN-predicate in the SQL Reference Manual

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 runtime and 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 (page 197)

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, state`
`FROM 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 (page 197)

Flattening FROM Clause Subqueries and Views

A subquery in the `FROM` clause must be evaluated before the containing query can be evaluated, such as in the following statement, where all the records in table `fact` must be evaluated before records in table `T`:

```
=> SELECT * FROM (SELECT a, MAX(a) AS max FROM (SELECT * FROM fact) AS T GROUP BY
a);
```

If such queries could be internally rewritten so its subqueries were combined with outer query block, queries would often run more quickly. This internal optimization is called subquery flattening, where Vertica flattens some `FROM` clause subqueries into the containing query, offering significant performance improvements.

For example, the previous query is flattened as follows:

```
=> SELECT * FROM (SELECT a, MAX(a) FROM fact GROUP BY a) AS T;
```

Both queries return the same results, but the flattened query runs more quickly.

Note: When views are mentioned in the `FROM` clause of a SQL query, Vertica first replaces the view names with the view definition queries, creating further opportunities for subquery flattening. This process is called view flattening, and the process described for subquery flattening also applies to view flattening. See Implementing Views in the Administrator's Guide for additional details about views.

Vertica flattens subqueries or views into the containing query, as long as the subquery or view does not contain:

- Aggregates
- Analytics
- An outer join

- A GROUP BY, ORDER BY, or HAVING clause
- DISTINCT keyword
- A LIMIT or OFFSET clause
- A UNION
- An EXISTS subquery

TIP: To see if a FROM clause subquery has been flattened, inspect the query plan. Typically, the number of value expression nodes (`ValExpNode`) decreases after flattening. See EXPLAIN in the SQL Reference Manual.

Examples

If you have a predicate that applies to a view or subquery, flattening allows optimizations by evaluating the predicates before the flattening takes place. In this example, without flattening, Vertica must first evaluate the subquery, and only then can the predicate `WHERE x > 10` be applied. In the flattened subquery, 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;
```

Vertica internally transforms this query as follows:

```
=> SELECT * FROM (SELECT * FROM A) AS fact 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;
```

Vertica transforms `FROM` clause subqueries within a `WHERE` clause `IN` subquery as shown below:

Original query: `SELECT * FROM a WHERE b IN (SELECT b FROM (SELECT * FROM dim) AS D WHERE x=1;`

Flattened query: `SELECT * FROM a WHERE b IN (SELECT b FROM dim) AS D WHERE x=1;`

See Also

Subquery Restrictions (page 197)

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 [schema-name.]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.

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, Vertica returns an error specifying which row had more than one match.

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*** (page 187) 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 [schema_name.]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.

Examples

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 (page 197)

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:

```
=> SELECT *
   FROM online_sales.online_sales_fact
   INNER JOIN public.customer_dimension
   USING (customer_key)
   WHERE customer_state = 'MA';
```

Subquery:

```
=> SELECT *
   FROM online_sales.online_sales_fact
   WHERE customer_key IN
     (SELECT customer_key
      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            |  61192.786013986
 Matt                | Gauthier           |        988807 | South           | 57337.8638902996
 Lauren              | Nguyen             |        968625 | West            | 56848.4274914089
 Jack                | Campbell           |        963914 | West            | 56848.4274914089
 William             | Martin             |        943477 | NorthWest       | 58928.2276119403
 Luigi               | Campbell           |        939255 | MidWest         | 59614.9170454545
 Sarah               | Brown              |        901619 | South           | 57337.8638902996
 Craig               | Goldberg           |        895836 | East            |  61192.786013986
 Sam                 | Vu                 |        889841 | MidWest         | 59614.9170454545
 Luigi               | Sanchez            |        885078 | MidWest         | 59614.9170454545
 Michael             | Weaver             |        882685 | South           | 57337.8638902996
 Doug                | Pavlov             |        881443 | SouthWest       | 57187.2510548523
 Ruth                | McNulty            |        874897 | East            |  61192.786013986
 Luigi               | Dobisz             |        868213 | West            | 56848.4274914089
 Laura               | Lang               |        865829 | East            |  61192.786013986
 ...
```

You can also use the UNION [ALL] keyword 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 lowfat 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 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 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, Vertica returns 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
```

- 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*** (page 175).

Vertica supports the following join types:

- ***Inner*** (page 200) (including ***natural*** (page 202), ***cross*** (page 204)) joins
- Left, right, and full ***outer*** (page 205) joins
- Optimizations for equality and ***range*** (page 206) joins predicates
- Hash, merge and sort-merge join algorithms.

There are three basic algorithms that perform a join operation: nested loops, merge joins, and hash joins. This chapter does not describe how join algorithms work outside mentioning them in the following list:

- If both inputs are pre-sorted, merge joins do not have to do any pre-processing. Vertica uses the term sort-merge join to refer to the case when one of the inputs must be sorted prior to the merge join. Vertica sorts the inner input side but only if the outer input side is already sorted on the join keys.
- Hash joins are used only for equi-joins where hashed values are compared for equality, not for other relationships.
- Vertica does not support nested loops 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*** (***http://savage.net.au/SQL/sql-99.bnf.html***) for additional details.

Although some users continue to use the older join syntax, Vertica 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. (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: 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.

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.
- You can also use inner join syntax to specify joins for pre-join projections. See ***Pre-join Projections and Join Predicates*** (page 208).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 Vertica 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*** (page 205).

See Also

Joins Notes and Restrictions (page 210)

Equi-joins and Non Equi-Joins

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)
```

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 dimension 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*** (page 201) because join condition are equal between common columns. (If the data types are incompatible, Vertica returns an error.)

```
=> 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;
```

In another illustration, the following three queries return the same result expressed as a basic query, an inner join, and a natural join. Note that the table expressions are equivalent only if the common attribute in Table 1 (`store_sales_fact`) and Table 2 (`store_dimension`) 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, it is easy to imagine how cross join results can be extremely large and difficult to manage. Cross joins can be useful, however, such as when returning a single-row result set.

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*** (page 208).

Vertica 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.:

```
=> 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 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 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

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

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
  ```

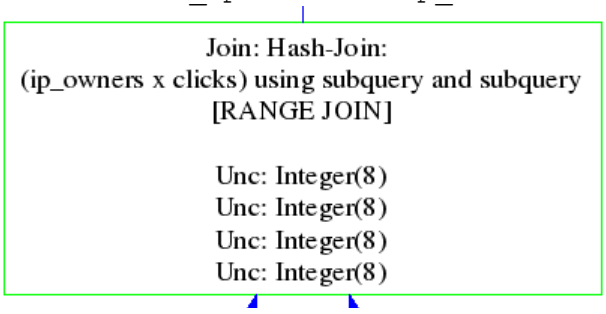
```
    ON fact.point BETWEEN NVL(dim.start, -1) AND NVL(dim.end,
   1000000000000);
```

- 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

Vertica can use pre-join projections when queries contain ***equi-joins*** (page 201) 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. 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: 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 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.
- You can also use inner join syntax to specify joins for pre-join projections. See ***Pre-join Projections and Join Predicates*** (page 208).
- 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;
  ```

- 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.
- Vertica 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.
- 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.
- 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.
-

Using SQL Analytics

The ANSI SQL 99 standard introduced a set of functionality, called SQL analytic functions, that handle complex analysis and reporting, for example, a moving average of retail volume over a specified time frame or a running total.

Analytic aggregate functions differ from standard aggregate functions in that, rather than return a single summary value, they return the same number of rows as the input. Moreover, unlike standard aggregate functions, the groups of rows on which the analytic aggregate function operates are not defined by a GROUP BY clause, but by window partitioning and frame clauses. You can sort these 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.

The windowing components (partitioning, ordering, and framing) are specified in the analytic `OVER()` clause. For example, window framing defines the unique construct of a moving window, whose size is based on either logical intervals (such as time) or on a physical number of rows. For each row, a window is computed in relation to the current row. As the current row advances, the window moves along with it.

Analytic functions take the following form:

`analytic_function ( arguments ) OVER( analytic_clause )`

Analytic functions conform to the following phases of execution:

1. Take the input rows after `WHERE`, `GROUP BY`, `HAVING` clause operations and joins are performed.
2. Group the rows according to the `PARTITION BY` clause.

   Note: The analytic `PARTITION BY` clause (called the `window_partition_clause (page 214))` is different from table partition expressions. See Table Partitioning in the Administrator's Guide for details.

   Unlike normal `GROUP BY` aggregation, analytic functions output the same number of rows as the input.
3. Order the rows within partitions according to analytic `ORDER BY` clause.

   Note: The analytic `ORDER BY` clause (called the `window_order_clause (page 215))` 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*** (page 271) for additional information about sort computation.
4. Compute some analytic function for each row.

Notes

Analytic functions:

- Require the `OVER()` clause. However, depending on the analytic function, the `window_frame_clause` and `window_order_clause` might not apply.

Note: When used with analytic aggregate functions, `OVER()` may be used 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.
- Cannot be nested; for example, the following is not allowed:
  `=> SELECT MEDIAN(RANK() OVER(ORDER BY sal) OVER().`

Tip:

Remember that analytic functions are evaluated *after* all other clauses except the query's final `ORDER BY` clause. So if you were to write a query like the following, which gets all rows with sales larger than the median, the system would return an error:

```
=> SELECT name, sales, MEDIAN(sales) OVER () AS m from allsales WHERE sales < m;
ERROR:  column "m" does not exist
```

Rewrite the query to use a subquery and mirror the analytic evaluation order:

```
=> SELECT * FROM (SELECT name, sales, MEDIAN(sales)
   OVER() AS m FROM allstates) sq WHERE sales < m;

 name | sales | m
------+-------------
 G    |    10 | 20
 C    |    15 | 20
(2 rows)
```

See Also

Analytic Functions in the SQL Reference Manual

Using Time Series Analytics (page 233)

The Window OVER() Clause

A *window* specifies 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 window `OVER()` clause specifies that the analytic function operates on a query result set (the rows that are returned after the `FROM`, `WHERE`, `GROUP BY`, and `HAVING` clauses have been evaluated). You use then use the `OVER()` clause to define a moving window of data for every row in a partition with certain analytic functions.

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 )
```

For details, see:

- ***Window Partitioning*** (page 214)
- ***Window Ordering*** (page 215)
- ***Window Framing*** (page 216)

Named Windows

You can avoid typing long `OVER()` clause syntax by naming a window using the `WINDOW` clause, which takes the following form:

```
WINDOW window_name AS ( window_definition_clause );
```

In the following example, `RANK()` and `DENSE_RANK()` use the partitioning and ordering specifications in the window definition for `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:

- 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 AS
     WINDOW w1 AS (PARTITION BY deptno), w2 AS (w1 ORDER BY sal);
  ```

Window Partitioning

Window partitioning divides the rows in the input by a given list of columns or expressions. If the optional `window_partition_clause` is omitted, all input rows are treated as a single partition. Window partitioning is similar to `GROUP BY`, except the function returns one result row per input row.

The analytic function is computed per partition and starts over again (resets) at the beginning of each subsequent partition.

Syntax

```
OVER( window_partition_clause
      window_order_clause
      window_frame_clause )
```

The examples in this topic `use` the following 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;
```

```
=> 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)
```

Examples

The first example uses the analytic function `MEDIAN()` to partition the results by state and then calculate the median of sales:

```
=> SELECT state, name, sales, MEDIAN(sales)
     OVER (PARTITION BY state) AS MEDIAN from allsales;
 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)
```

In the above results, notice the two partitions for MA and NY under the MEDIAN column.

The next example calculates the median of total sales among states. Note that 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)
```

Window Ordering

Window ordering sorts the rows specified by the OVER() clause and supplies the ordered set of rows to the `window_frame_clause` (if present), to the analytic function, or to both. Using ORDER BY in an OVER() clause changes the default window to RANGE UNBOUNDED PRECEDING AND CURRENT ROW.

Syntax

```
OVER ( window_partition_clause  window_order_clause  window_frame_clause )
```

The `window_order_clause` specifies whether data is sorted in ascending or descending order and specifies where null values appear in the sorted result as either first or last; for example:

```
ORDER BY expr_list [ ASC | DESC ]
                   [ NULLS { FIRST | LAST | AUTO ]
```

The following list shows the default ordering, with bold clauses to indicate what is implicit:

- 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**

The placement of the ORDER BY clause might not guarantee the final result order. For example, the `window_order_clause` is different from the final ORDER BY in that the `window_order_clause` specifies the order within each partition and affects the result of the analytic calculation; it does not guarantee the order of the SQL result. Use the SQL ORDER BY clause to guarantee the ordering of the final result set. See also ***Null Placement*** (page 271).

Example 1

In this example, the query orders the sales inside each sales partition:

```
SELECT state, sales, name, RANK()
OVER (PARTITION BY state
  ORDER BY sales) AS RANK
FROM allsales;
```

Example 2

In this example, the final ORDER BY clause sorts the results by name:

```
SELECT state, sales, name, RANK()
OVER (PARTITION by state
  ORDER BY sales) AS RANK
FROM allsales ORDER BY name;
```

```
 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)
```

```
 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 (the `window_frame_clause`) represents a unique construct called a **moving window**, which is defined in the analytic OVER() clause. The window frame clause defines a window relative to the current row, in terms of either logical intervals (such as time) or on a physical number of records before and/or after the (current) row. Logical windows are expressed using the RANGE keyword and physical windows using the ROWS keyword. As the current row advances, the window boundaries are recomputed along with it, to determine what 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.

Syntax

`OVER ( window_partition_clause  window_order_clause  **window_frame_clause** )`

Each analytic function is computed based on data within the window frame boundaries. The window size is based on either logical intervals (such as time) or on a physical number of records. For each row, Vertica computes a window based on the current row. As the current row advances, the window recomputes along with it, and rows are excluded or included based on the position (ROWS) or value (RANGE) relative to the current row.

Window Aggregates

Analytic functions that take the `window_frame_clause` are called window aggregates, and they return information such as moving averages and cumulative results. To use the following functions as window (analytic) aggregates, instead of basic aggregates, specify both an ORDER BY clause (`window_order_clause`) and a moving window (`window_frame_clause`) in the OVER() clause. If you omit the `window_frame_clause` but you specify the `window_order_clause`, the system provides the default window of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

- 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, there is no moving window, and the function is used as a reporting function, where the entire input is treated as one partition.

Note: The value returned by an analytic function with a *logical* offset is always deterministic. However, 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.

Framing Windows with ROWS

ROWS specifies the window as a physical offset.

Legend

In the examples on this page:

- The blue 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:

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.

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 now spans 2 preceding and current row:

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

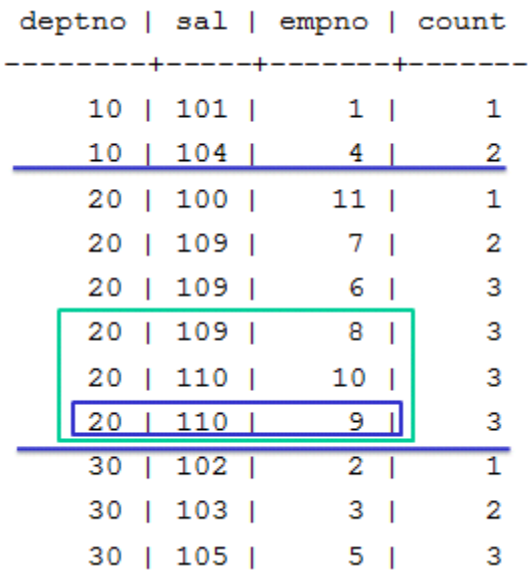
When the current row moves, the window slides to maintain 2 preceding and current row. The count of 3 is repeated because it represents the number of rows in the window:

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

Here, the current row advances yet again, and the window spans from 2 rows preceding to the current row:

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 again and the window span is defined by the window frame once again. 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

Framing Windows with RANGE

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 it is one of following:

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
- RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

Legend

In the examples on this page:

- The blue 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 RANGE-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 range between the current row and two preceding.

```
SELECT deptno, sal, empno, COUNT(*)
OVER (PARTITION BY deptno order by sal
      RANGE 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:

```
 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

In the next example, 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

In this example, 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
```

Notes

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.

Reporting Aggregates

Reporting aggregate functions let you compare a partition's aggregate values with detail rows, taking the place of correlated subqueries or joins. In this context, these functions do not have a `window_order_clause` or a `window_frame_clause`; otherwise they would be treated as window aggregates.

- AVG
- COUNT
- MAX and MIN
- SUM
- STDDEV, STDDEV_POP, and STDDEV_SAMP
- VARIANCE, VAR_POP, and VAR_SAMP

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`, 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 FROM 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)
```

Sample Analytics Queries

The following two queries operate on a stocks table defined as:

```
=> CREATE TABLE ticks (ts TIMESTAMP, stock VARCHAR(10), bid FLOAT);
```

Price differential for two stocks

The subquery selects out two stocks of interest. The outer query uses the last_value and over components of analytics, with ignore nulls.

```
=> 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 = 'sym1' THEN bid ELSE NULL END AS price1,
       CASE WHEN stock = 'sym2' THEN bid ELSE NULL END AS price2
       FROM ticks
       WHERE stock IN ('sym1','sym2')
      ) v1
   ORDER BY ts;
```

Moving average

This example calculates a 40-second moving average of bids for one stock:

```
SELECT ts, bid, avg(bid) OVER(order by ts
  RANGE BETWEEN INTERVAL '40 SECONDS' PRECEDING AND CURRENT ROW)
FROM ticks
WHERE stock = 'sym1'
GROUP BY bid, ts
ORDER BY ts;
```

Latest Bid and Ask

The following query operates on a table defined as:

```
CREATE TABLE OrderBookLevel1(
    vendorinstrumentid VARCHAR(100),
    utcdatetime TIMESTAMP,
    sequenceno INT,
    askprice FLOAT,
    asksize INT,
    bidprice FLOAT,
    bidsize INT);
```

The following statement fills in missing (null) values to create a full Order Book showing the latest bid, ask price, and size, by instrument ID. Original rows have values for (typically) one price and one size, so use the LAST_VALUE() analytic function 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.

```
SELECT
    sequenceno SEQ,
    utcdatetime "TIME",
    vendorinstrumentid RIC,
    LAST_VALUE ( bidprice IGNORE NULLS )
     OVER (PARTITION BY vendorinstrumentid ORDER BY sequenceno
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
    AS "BID PRICE",
    LAST_VALUE ( bidsize IGNORE NULLS )
     OVER (PARTITION BY vendorinstrumentid ORDER BY sequenceno
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
    AS "BID SIZE",
    LAST_VALUE ( askprice IGNORE NULLS )
     OVER (PARTITION BY vendorinstrumentid ORDER BY sequenceno
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
    AS "ASK PRICE",
    LAST_VALUE ( asksize IGNORE NULLS )
     OVER (PARTITION BY vendorinstrumentid ORDER BY sequenceno
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
    AS  "ASK SIZE"
 FROM orderbooklevel1 ORDER BY sequenceno;
```

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 Vertica. These functions are a 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_EVENT and CONDITIONAL_TRUE_EVENT functions do not allow ***window framing*** (page 216).

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;
```

CONDITIONAL_CHANGE_EVENT

The analytical function CONDITIONAL_CHANGE_EVENT returns a sequence of integers indicating window numbers, starting from 0. The window number is incremented, when the result of evaluating expression on the current row differs from that on the previous one.

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, and because each row is different from the previous row, the function increments the window ID from 0 to 3:

```
SELECT ts, symbol, bid
FROM 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
```

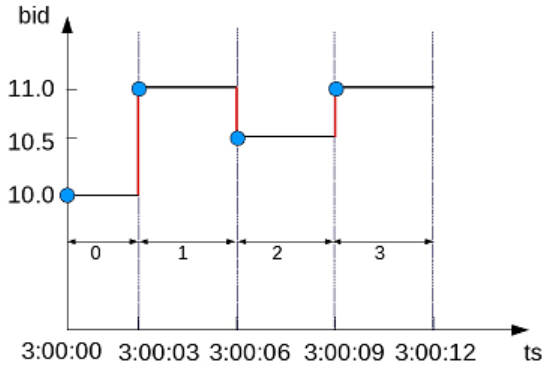
==>

```
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 03:00:09 | XYZ    |   11
(4 rows)
```

```
 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 by 1 each for each time slice:



So the window ID starts at 0 and increments by 1 at every change in value.

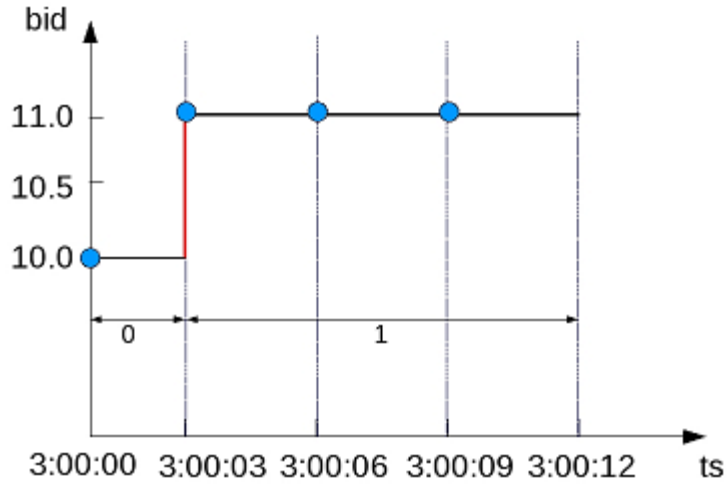
In this example, the bid price changes from $10 to $11 in the second row. So the CONDITIONAL_CHANGE_EVENT function returns the same window ID for the other rows, which also returned a bid value of $11.:

```
SELECT ts, symbol, bid
FROM 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    |   11
 2009-01-01 03:00:09 | XYZ    |   11
```

==>

```
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    |   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:



CONDITIONAL_TRUE_EVENT

Like CONDITIONAL_CHANGE_EVENT, the analytic function CONDITIONAL_TRUE_EVENT returns a sequence of integers indicating window numbers, starting from 0. The difference between the two functions is that CONDITIONAL_TRUE_EVENT increments the window ID every time the expression evaluates to true, so even if the value remains the same, such as in the previous example ($11.0), the window ID increments by 1 for each value where the expression is true.

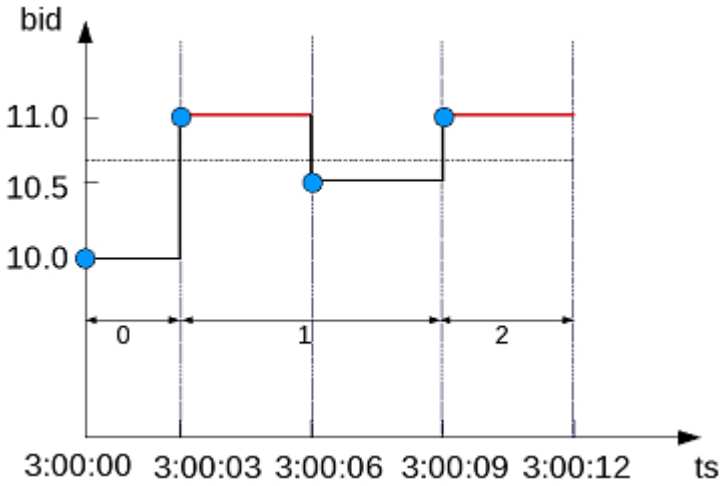
In the following example, the first query returns all records from the TickStore3 table. The second query uses the CONDITIONAL_TRUE_EVENT function to increment the window ID each time the bid value is greater than $10.6. The first window ID to be returned is on row 2, where the value is $11. The window ID stays the same for the next row (because the value is not greater than $10.6), and increments by 1 for the final row:

```
SELECT ts, symbol, bid
FROM 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 > 10.6)
  OVER(ORDER BY ts)
FROM Tickstore3;
         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. 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 window ID to be returned is on row 2, where the value is $11. The window ID then increments each time after that. Even though the value stays the same ($11), it is greater than $10.6 for each time slice:

```
SELECT ts, symbol, bid
```

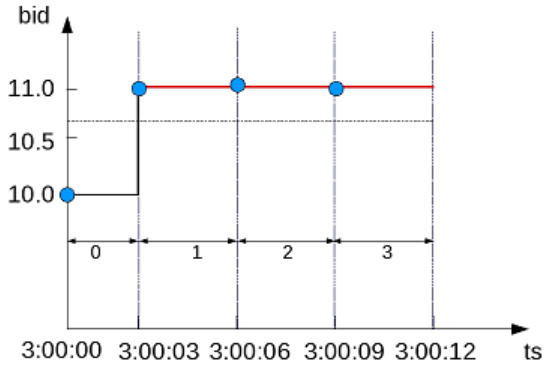
```
SELECT CONDITION_TRUE_EVENT(bid > 10.6)
```

```
FROM 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    |   11
 2009-01-01 03:00:09 | XYZ    |   11
```

==>

```
  OVER(ORDER BY ts)
FROM Tickstore3;
         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 slices (3:00:03) and remains there for the remaining two time slices; however, the window ID increments each time because each value is greater than $10.6:



Advanced use of event-based windows

In event-based window functions, the condition expression accesses values from the current row only, but if you want to look at 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, LAG(x, n) retrieves the value of column X in the *n*th to last input record. The semantics in this case are the same as the analytic function LAG, and the OVER() clause can be used.

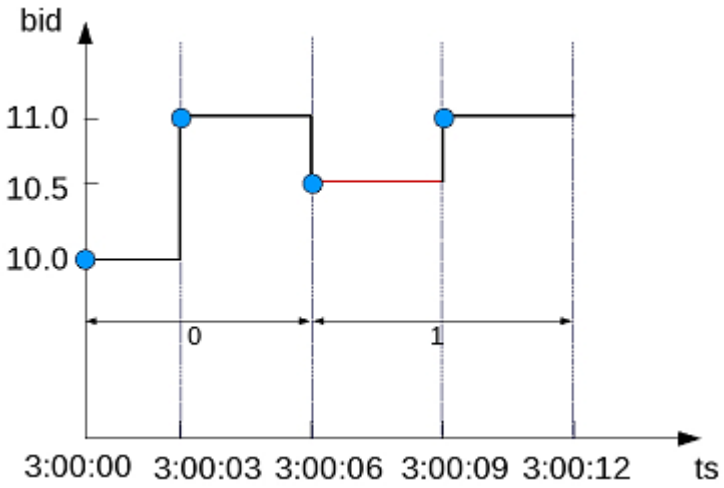
In the following example, the first query returns all records from the TickStore3 table. The second query uses the CONDITIONAL_TRUE_EVENT function with LAG() to increment the window ID each time the bid value is less than the previous value. The first window ID starts on the third time slice because $10.5 is less than $11, and it remains at 1 because the final value is greater than in the previous row:

Anything with the LAG expression shares the OVER clause specifications

```
SELECT ts, symbol, bid
FROM 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 CONDITION_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 (page 230)

Using Time Series Analytics (page 233)

CONDITIONAL_CHANGE_EVENT(), CONDITIONAL_TRUE_EVENT() and LAG() in the SQL Reference Manual

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 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 are treated as though they are from two different browsing sessions.

Example Schema

The examples in this topic use the following 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 following example illustrates the standard semantics of sessionization. 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 (page 225)

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. 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. 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*** (page 225) 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*** (page 230), 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.

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 (page 211), particularly ***Event-based Windows*** (page 225) and ***Sessionization*** (page 230)

Gap Filling and Interpolation (GFI)

Example Schema

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 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, as input records usually occur at non-uniform intervals and could contain gaps. Consider, for example, the following table, which contains two input rows 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 the above inputs, how would you determine the bid price 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, Vertica provides gap-filling functionality, which fills in missing data points. Vertica then provides an interpolation scheme, which is a method of constructing new data points within the range of a discrete set of known data points. 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. This is accomplished with time series aggregate functions and a the SQL TIMESERIES clause, which are discussed later in this topic.

But first, we'll illustrate the components that make up gap filling and interpolation in Vertica, starting with ***Constant Interpolation*** (page 235).

Note: 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 the input timestamps at 03:00:00 and 03:00:05 in the sample TickStore schema, what if you wanted to 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**, where Vertica computes a value based on the other input records; for example, 03:00:00 and 03:00:05.

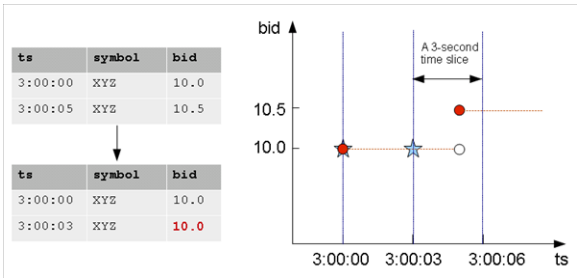
Note: Constant is the default interpolation scheme. Another interpolation scheme, ***linear*** (page 237), is discussed in an upcoming topic.

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 (illustrated in the following figure), 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.

Figure 1:



Before you write a query that makes the input rows from a table like the above example more uniform, you first need to understand the ***TIMESERIES clause and time series aggregate functions*** (page 235), which are described in the following topic.

The TIMESERIES Clause and Aggregates

The TIMESERIES clause and time series aggregates help solve the original problem, which was to normalize the data into 3-second time slices and interpolate the bid price when necessary (e.g., when there were gaps in the input data).

- The SELECT..TIMESERIES clause, an important component of time series analytics computation, provides gap-filling and interpolation (GFI) computation. The clause applies to the timestamp columns/expressions in the data, and the syntax is:

```
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.

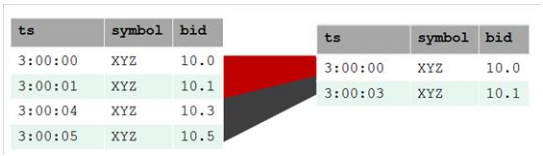
For additional details, see SELECT..TIMESERIES Clause in the SQL Reference Manual.

- Time series 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.

  The following table shows 3-second time slices where

  - The first two rows fall within the first time slice [3:00:00, 3:00:03), and they 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 [3:00:04, 3:00:05), and they are the input rows for the TSA function's output for the time slice starting at 3:00:04.

  The result is the start of each time slice.



For additional details, see Timeseries Aggregate (TSA) Functions in the SQL Reference Manual

Examples

The following statement uses both the TIMESERIES clause and the TS_FIRST_VALUE timeseries aggregate 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. Vertica interpolates the last known value and fills in the gap, returning 10 at 3:00:03:

Original query

```
     slice_time      | bid
---------------------+-----
 2009-01-01 03:00:00 |  10
 2009-01-01 03:00:03 |10.5
(2 rows)
```

==>

Interpolated value

```
     slice_time      | bid
---------------------+-----
 2009-01-01 03:00:00 |  10
 2009-01-01 03:00:03 |  10
(2 rows)
```

Linear Interpolation

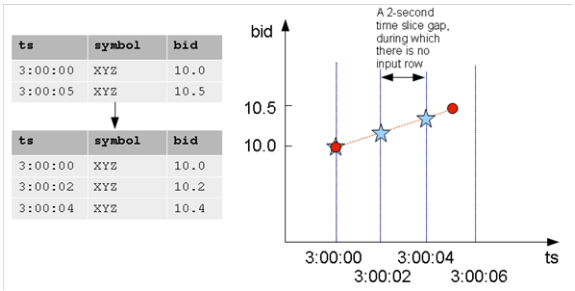
So far, the topics in this section have discussed an interpolation policy where the value is set to the *last seen value*, also called constant interpolation. The second interpolation policy provided is linear interpolation, where Vertica interpolates values in a linear slope based on the specified time 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)
```

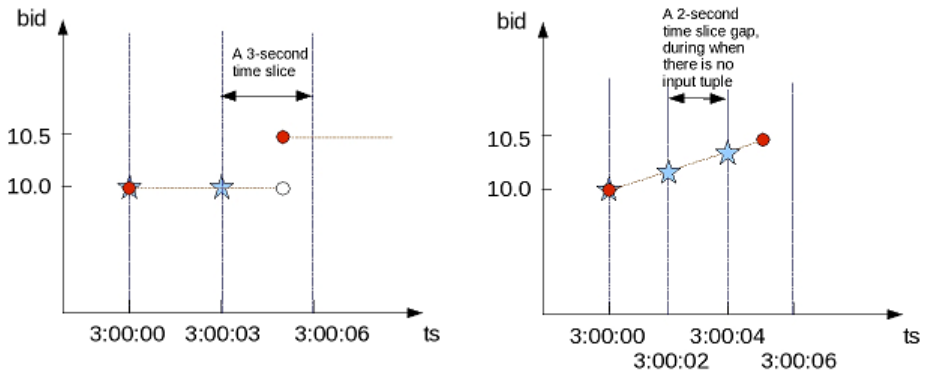
The following figure illustrates the previous query results, showing the 2-second time gaps (3:00:02 and 3:00:04) in which no input record occurs.

Using linear interpolation, 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 PM and 3:00:05). At 3:00:05, the value would change to 10.5. The known data points are represented by a red dot, and the interpolated values are represented by blue stars.



The following is a side-by-side comparison of the two interpolation schemes.

CONST interpolation **LINEAR interpolation**



Gap Filling and Interpolation Examples

This topic illustrates some of the queries you can write using the two difference Vertica interpolation schemes, constant and linear.

Constant interpolation

The examples in this section use the TIMESERIES clause along with TS_FIRST_VALUE and TS_LAST_VALUE functions with the default interpolation scheme, which is based on the last seen value as a constant.

The first query uses the time series aggregate function, TS_FIRST_VALUE, with 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, this next query uses the time series aggregate function, 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

The examples in this section use the linear interpolation scheme.

Based on the same input described in the previous 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

Here's an example using the analytic LAST_VALUE function, 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)
```

Creating a dense time series

The examples that follow use the same schema defined in ***Gap Filling and Interpolation (GFI)*** (page 234) to create 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 from the previous examples in the ***Gap Filling and Interpolation (GFI)*** (page 234) topic, 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);
```

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 03:00:00.50
 2009-01-01 03:00:01
 2009-01-01 03:00:01.50
 2009-01-01 03:00:02
 2009-01-01 03:00:02.50
 2009-01-01 03:00:03
 2009-01-01 03:00:03.50
 2009-01-01 03:00:04
 2009-01-01 03:00:04.50
 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 and, 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 Nulls

Although null values are not common inputs for gap-filling and interpolation (GFI) computation, if there are null argument values to time series aggregate functions, the presence or absence of the IGNORE NULLS keywords can affect the interpolated values.

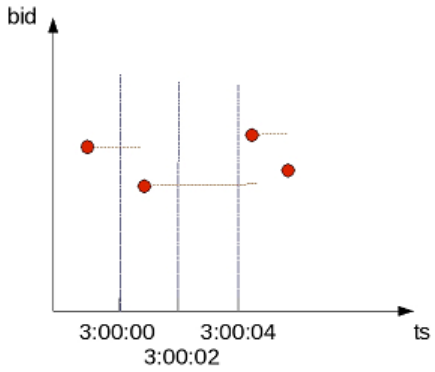
This section describes how Vertica handles such cases:

- For an input row with a null value in its timestamp (`ts`) column, that row is ignored or treated as though it had been filtered out just before the GFI computation occurred.
- For an input row with a null value in column `bid` that is not `ts`, say its `ts` value is `t`. In the interpolated result of `bid`, the `bid` values around time `t` are null. In other words, if the value on either side is null, the result is null.

Constant interpolation with null values

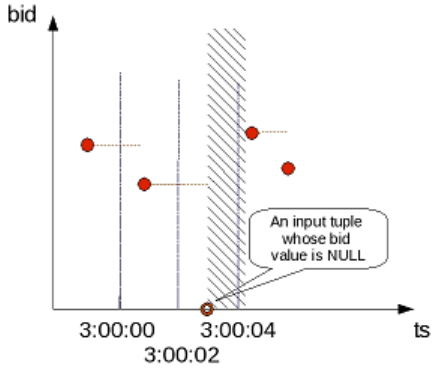
The following Figure 1 illustrates the constant interpolation result on four input rows where there is no null value.

Figure 1: CONST-interpolated bid values with no nulls



The same four input rows are present in Figure 2. However, you'll notice an additional input row with `bid` value of null and a `ts` value of 3:00:03. This input row is represented in the figure by a red ring:

Figure 2: CONST-interpolated bid values with NULLs



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 in the time interval denoted by the shaded region null. 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.

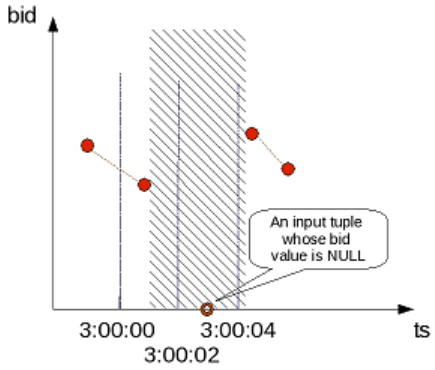
The last value of the 3:00:02 time slice is null; therefore, the first value for the next time slice (3:00:04) is null. However, if you had specified IGNORE NULLS, then the value at 3:00:04 would be the same value as it was at 3:00:02.

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. This is because in the presence of an input null value at 3:00:03, Vertica cannot linearly interpolate the bid value around that time point.

Note: 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.

Figure 3: LINEAR-interpolated bid values with NULLs



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.

Vertica supports the IGNORE NULLS option for TS_FIRST_VALUE and TS_LAST_VALUE, similar to their analytic function (FIRST_VALUE/LAST_VALUE) counterparts. If the timestamp itself is null, it would be the same as if Vertica filter it out before gap filling and interpolation occurred.

For example, `TS_FIRST_VALUE(bid IGNORE NULLS)` applied to the input illustrated in Figure 6 performs its computation as though it were processing the input in Figure 4. You can achieve the same results by filtering out rows whose bid is null before you perform GFI computation. The null value for the column on which a time series aggregate is applied, for example bid, is ignored and filled per the interpolation scheme.

Notes

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, Vertica does not support the following query:

```
=> SELECT pb, slice_time, TS_FIRST_VALUE(a IGNORE NULLS) AS fv
   FROM table1
   WHERE slice_time = '2009-9-28 10:00:00'
   TIMESERIES slice_time as '2 seconds' over (partition by pb order by ts);
```

Instead, you could write a subquery and put the predicate on `slice_time` in the outer query:

```
=> SELECT * FROM (
     SELECT pb, slice_time,
       TS_FIRST_VALUE(a IGNORE NULLS) AS fv
     FROM table1
     TIMESERIES slice_time AS '2 seconds'
     OVER (PARTITION BY pb ORDER BY ts) ) sq
   WHERE slice_time = '2009-9-28 10:00:00';
```

Event Series Joins

An event series join is a 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*** (page 205), 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:

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

The examples that follow use the sample schemas in this topic.

Tip: 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*** (page 248).

The hTicks and aTicks tables

```
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;
```

Query hTicks to see its contents:

```
=> 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)
```

Query aTicks to see its contents:

```
=> SELECT * FROM aTicks;
```

Notice there are no entry records at 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)
```

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)
```

The bid and ask tables

```
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;
```

Query the `bid` table to see its contents:

```
=> 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)
```

Query the `ask` table to see its contents:

```
=> 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)
```

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 ***example schema*** (page 245) used for 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, and this becomes challenging when you want to compare two stocks whose timestamps don't match.

The hTicks and aTicks tables

In the following 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.

```
=> SELECT * FROM hTicks;
 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;
 stock |   time   | price
-------+----------+--------
 ACME  | 12:00:00 | 340.00
 ACME  | 12:03:00 | 340.10
 ACME  | 12:05:00 | 333.80
 ACME  | 12:05:00 | 340.20
(4 rows)
```

Querying event series data with full outer joins

Using a traditional full outer join, the 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 | NULL  | NULL     | NULL
 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 | NULL  | NULL     | NULL
 NULL  | NULL     | NULL  | ACME  | 12:03:00 | 340.10
(6 rows)
```

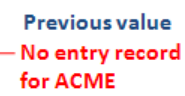
To replace the gaps with interpolated values for those missing data points, use the INTERPOLATE predicate, which represents an event series join. The join condition is restricted to the ON clause, which evaluates the equality predicate on the timestamp columns from the two input tables. In other words, 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);
```

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 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*** (page 245) for the `bid` and `ask` tables, here is their output:

bid table

```
=> SELECT * FROM bid;
 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 table

```
=> SELECT * FROM ask;
 stock |   time   | price
-------+----------+--------
 ACME  | 12:00:00 |  80.00
 ACME  | 12:02:00 |  75.00
 HPQ   | 12:01:00 | 101.00
(3 rows)
```

Write a full outer join that interpolates 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 | NULL   | NULL     | NULL
 HPQ    | 12:01:00 | 100.00 | HPQ    | 12:01:00 | 101.00
(6 rows)
```

Note that the same row (`ACME,12:02,75`) from the `ask` table appears multiple times. The first appearance is because no matching rows are present in the `bid` table for the row in `ask`, and so the appropriate bid row is filled in using the ACME value. at 12:02 (75). 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 table schema:

```
CREATE TABLE clickstream_log (
```

```
  uid INT,             --user ID
  sid INT,             --browsing session ID, produced by previous sessionization computation
  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/shamwow', 'V');
insert into clickstream_log values (1,100,'12:03','website2.com/shamwow','website2.com/buy', '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

The example that follows (which includes the Vertica pattern matching functions) analyzes users' browsing history over website2.com and 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 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 outout below, the first 4 rows represent the pattern for one user's browsing activity, while the following 3 rows show another user'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 and Pattern Matching Functions in the SQL Reference Manual

Perl Regular Expressions Documentation (***http://perldoc.perl.org/perlre.html***)

Collecting Statistics

The Vertica cost-based query optimizer relies on representative statistics on the data, statistics that are used to determine the final plan to execute a query.

Various optimizer decisions rely on having up-to-date statistics, which means choosing between:

- Multiple eligible projections to answer the query
- The best order in which to perform joins
- Plans involving different algorithms, such as HASH JOIN versus MERGE JOIN or HASH GROUP BY versus PIPELINED GROUP BY
- Data distribution algorithms, such as broadcast and re-segmentation

Without reasonably accurate statistics, the optimizer could choose a suboptimal projection or a suboptimal join order for a query.

To understand how Vertica collects statistics, consider this common use case where you load timestamp data into a fact table on an ongoing basis (hourly, daily, etc.) and then run queries that select the recently-loaded rows from the fact table.

If you load, for example, days 1 through 15 and run the ANALYZE_STATISTICS() function, a subsequent query that asks for "yesterday's data" (filtering on the timestamp column) is planned correctly by the optimizer. If on the next day, you load day 16 data and run the same query, but do not rerun ANALYZE_STATISTICS, the optimizer might conclude that the predicate results in only one row being returned because the date range drops off the end of the histogram range and the data becomes stale.

You can resolve the issue by running ANALYZE_STATISTICS() after day 16 data is loaded. For example, when the optimizer detects that statistics are not current for a particular predicate (such as when a timestamp predicate is out of a histogram's boundary), Vertica plans those queries using other considerations, such as FK-PK constraints, when available.

You can also look for statistics in the EXPLAIN plan; for example, when statistics are off outside a histogram's boundaries, the EXPLAIN plan is annotated with a status. See ***Reacting to Stale Statistics*** (page 263) for details.

SQL syntax and parameters for the functions and system tables described in the topics in this section are described in the SQL Reference Manual:

- ANALYZE_HISTOGRAM()
- ANALYZE_STATISTICS()
- ANALYZE_WORKLOAD()
- DROP_STATISTICS()
- EXPORT_STATISTICS()
- IMPORT_STATISTICS()
- V_CATALOG.PROJECTION_COLUMNS

Statistics Used by the Query Optimizer

Vertica uses the estimated values of the following statistics in its cost model:

- Number of rows in the table
- Number of distinct values of each column (cardinality)
- Minimum/maximum values of each column
- An equi-height histogram of the distribution of values each column
- Space occupied by the column on disk

The Vertica query optimizer and the Database Designer both use the same set of statistics. When there are ties, the optimizer chooses the projection that was created earlier.

How Statistics are Collected

Statistics computation is a cluster-wide operation that accesses data using a historical query (at epoch latest) without any locks. Once computed, statistics are stored in the catalog and replicated on all nodes. This operation requires an exclusive lock on the catalog for a very short duration, similar to a DDL operation. In fact, these operations require a COMMIT for the current transaction.

Vertica provides three ways to manually collect statistics:

- ANALYZE ROW COUNT
- ANALYZE_STATISTICS
- ANALYZE_HISTOGRAM

ANALYZE ROW COUNT

The `ANALYZE ROW COUNT` is a lightweight operation that automatically collects the number of rows in a projection every 60 seconds to collect a minimal set of statistics and aggregates row counts calculated during loads.

If you wanted to change the 60-second interval to 1 hour (3600 seconds), you would issue the following command:

```
=> SELECT SET_CONFIG_PARAMETER('AnalyzeRowCountInterval', 3600);
```

To reset the interval to the default of 1 minute (60 seconds):

```
=> SELECT SET_CONFIG_PARAMETER('AnalyzeRowCountInterval', 60);
```

See Configuration Parameters for additional information. This function can also be invoked manually using the DO_TM_TASK('analyze_row_count') function.

ANALYZE_STATISTICS

The ANALYZE_STATISTICS function computes full statistics and must be explicitly invoked by the user. It can be invoked on all objects or on a per-table or per-column basis.

The ANALYZE_STATISTICS() function:

- Lets you analyze tables on a per-column basis for improved performance.

- Performs faster data sampling, which expedites the analysis of relatively small tables with a large number of columns.
- Includes data from WOS.
- Recognizes deleted data, instead of ignoring delete markers.
- Requires less memory to execute.
- Lets you cancel the function mid analysis by issuing CTRL-C on vsql or invoking the INTERRUPT_STATEMENT() function.
- Records the last time statistics were run for a table so that subsequent calls to the function can be optimized. See V_CATALOG.PROJECTION_COLUMNS for details.

ANALYZE_HISTOGRAM

ANALYZE_STATISTICS() is an alias for ANALYZE_HISTOGRAM(). The only difference is that ANALYZE_HISTOGRAM lets you decide on the tradeoff between sampling accuracy over speed by specifying what fraction of data (1-100) to read from disk.

Note: If you specify the percent parameter as 100, the entire projection is read from disk, and 128K rows are chosen at random. Otherwise, (row count) * (percent/100) rows are read from the projection in 100 contiguous bands with a minimum of 128K rows read. The one exception is if the column is first in the chosen projection's sort order, then all data is read from disk.

Examples:

In the following command, the system reads 10% of data from disk (default) and returns 0 for success:

```
=> SELECT ANALYZE_STATISTICS('shipping_dimension.shipping_key');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

The next command performs the same functionality as the previous ANALYZE_STATISTICS() command and returns 0 for success:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key');
 ANALYZE_HISTOGRAM
-------------------
                 0
(1 row)
```

With the percent parameter specified as 100, the following command performs a full column scan and returns 0 for success:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key', 100);
 ANALYZE_HISTOGRAM
-------------------
                 0
(1 row)
```

In this command, only 0.1% (1/1000) of the disk is read:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key', 0.1);
 ANALYZE_HISTOGRAM
-------------------
```

```
                  0
(1 row)
```

How Statistics are Computed

Vertica does not compute statistics incrementally, nor does it update full statistics during load operations.

For large tables exceeding 250,000 rows, histograms for minimum, maximum, and column value distribution are calculated on a sampled subset of rows. The default maximum number of samples for each column is approximately 2^17 (131702) samples or the number of rows that fits within 1GB of memory, whichever is smaller; for example, the number of samples used for large VARCHAR columns could be less.

Vertica does not provide a configuration setting to change the number of samples, but you can decide on the tradeoff between accuracy of sampling over speed by specifying what fraction of data to read from disk, from 1 to 100 (a full table scan). See ANALYZE_HISTOGRAM() in the SQL Reference Manual for details.

Note: If you specify the percent parameter as 100, the entire projection is read from disk, and 128K rows are chosen at random. Otherwise, (row count) * (percent/100) rows are read from the projection in 100 contiguous bands with a minimum of 128K rows read. The one exception is if the column is first in the chosen projection's sort order, then all data is read from disk.

How Statistics Are Reported

Hints about statistics are exposed in a couple ways. The EXPLAIN plan, for example, is annotated with a status (NO STATISTICS or STALE STATISTICS). See ***Reacting to Stale Statistics*** (page 263).

Vertica also records the last time ANALYZE_STATISTICS() was run for a table in order that subsequent calls to the function be optimized. This is useful during the database design process because if the Database Designer does not collect statistics when adding design tables, it generates a warning indicating that statistics are old. You can then decide if you want to run ANALYZE_STATISTICS before you proceed with the design.

This information is captured in two columns in the V_CATALOG.PROJECTION_COLUMNS system table:

- `STATISTICS_TYPE`—Returns the type of statistics the column contains (NONE, ROWCOUNT or FULL).
- `STATISTICS_COLLECTION_TIME`—Returns the last time statistics were collected in this table.

The new Workload Analyzer automatically suggests which tables/table columns should be re-analyzed with the ANALYZE_STATISTICS function.

Best Practices for Statistics Collection

The query optimizer requires representative statistics; however, for most applications statistics do not have to be accurate to the minute. The function DO_TM_TASK('analyze_row_count') collects partial statistics automatically by default and can be sufficient for many optimizer choices. For example, the following command analyzes the row count on the Vmart Schema database:

```
=> SELECT DO_TM_TASK('analyze_row_count');
                                DO_TM_TASK
-------------------------------------------------------------------------------
row count analyze for projection 'call_center_dimension_DBD_27_seg_temp_init_temp_init'
row count analyze for projection 'call_center_dimension_DBD_28_seg_temp_init_temp_init'
row count analyze for projection 'online_page_dimension_DBD_25_seg_temp_init_temp_init'
row count analyze for projection 'online_page_dimension_DBD_26_seg_temp_init_temp_init'
row count analyze for projection 'online_sales_fact_DBD_29_seg_temp_init_temp_init'
row count analyze for projection 'online_sales_fact_DBD_30_seg_temp_init_temp_init'
row count analyze for projection 'customer_dimension_DBD_1_seg_temp_init_temp_init'
row count analyze for projection 'customer_dimension_DBD_2_seg_temp_init_temp_init'
row count analyze for projection 'date_dimension_DBD_7_seg_temp_init_temp_init'
row count analyze for projection 'date_dimension_DBD_8_seg_temp_init_temp_init'
row count analyze for projection 'employee_dimension_DBD_11_seg_temp_init_temp_init'
row count analyze for projection 'employee_dimension_DBD_12_seg_temp_init_temp_init'
row count analyze for projection 'inventory_fact_DBD_17_seg_temp_init_temp_init'
row count analyze for projection 'inventory_fact_DBD_18_seg_temp_init_temp_init'
row count analyze for projection 'product_dimension_DBD_3_seg_temp_init_temp_init'
row count analyze for projection 'product_dimension_DBD_4_seg_temp_init_temp_init'
row count analyze for projection 'promotion_dimension_DBD_5_seg_temp_init_temp_init'
row count analyze for projection 'promotion_dimension_DBD_6_seg_temp_init_temp_init'
row count analyze for projection 'shipping_dimension_DBD_13_seg_temp_init_temp_init'
row count analyze for projection 'shipping_dimension_DBD_14_seg_temp_init_temp_init'
row count analyze for projection 'vendor_dimension_DBD_10_seg_temp_init_temp_init'
row count analyze for projection 'vendor_dimension_DBD_9_seg_temp_init_temp_init'
row count analyze for projection 'warehouse_dimension_DBD_15_seg_temp_init_temp_init'
row count analyze for projection 'warehouse_dimension_DBD_16_seg_temp_init_temp_init'
row count analyze for projection 'store_dimension_DBD_19_seg_temp_init_temp_init'
row count analyze for projection 'store_dimension_DBD_20_seg_temp_init_temp_init'
row count analyze for projection 'store_orders_fact_DBD_23_seg_temp_init_temp_init'
row count analyze for projection 'store_orders_fact_DBD_24_seg_temp_init_temp_init'
row count analyze for projection 'store_sales_fact_DBD_21_seg_temp_init_temp_init'
row count analyze for projection 'store_sales_fact_DBD_22_seg_temp_init_temp_init'
(1 row)
```

Running full `ANALYZE_STATISTICS` on a table is an efficient but potentially long-running operation that analyzes each unique column exactly once across all projections. It can be run concurrently with queries and loads in a production environment. Given that statistics gathering consumes resources (CPU and memory) from queries and loads, Vertica recommends that you run full `ANALYZE_STATISTICS` on a particular table whenever:

- The table is first bulk loaded.
- A new projection using that table is created and refreshed.
- The number of rows in the table changes by 50%.
- The minimum/maximum values in the table's columns change by 50%.
- New primary key values are added to tables with referential integrity constraints. Both the primary key and foreign key tables should be reanalyzed.
- Relative size of a table, compared to tables it is being joined to, has changed materially; for example, the table is now only five times larger than the other when previously it was 50 times larger.

- There is a significant deviation in the distribution of data, which would necessitate recalculation of histograms. For example, there is an event that caused abnormally high levels of trading for a particular stock. This is application specific.
- There is a down-time window when the database is not in active use.

Notes

- You can analyze statistics on a single table column, rather than on the entire table. Running statistics on a single important column (such as the predicate column) is useful for large tables, which could take a long time to compute. It's also a good idea to run statistics on a column after you alter a table to add or remove a column.
- Projections that have no data never have full statistics. Use the PROJECTION_STORAGE system table to see if your projection contains data.

Once your system is running well, Vertica recommends that you save exported statistics for all tables. In the unlikely scenario that statistics changes impact optimizer plans, particularly after an upgrade, you can always revert back to the exported statistics. See ***Importing and Exporting Statistics*** (page 259) for details.

See Also

Analyzing Workloads (page 266)

Importing and Exporting Statistics

Use the EXPORT_STATISTICS() function to generates an XML file that contains statistics for the database.

For example, the following command exports statistics on the VMart example database to a file:

```
vmart=> SELECT EXPORT_STATISTICS('/vmart/statistics/vmart_stats');
         export_statistics
-----------------------------------
 Statistics exported successfully
(1 row)
```

The `IMPORT_STATISTICS()` function can be used to import saved statistics from the XML file generated by the EXPORT_STATISTICS() command into the catalog where the saved statistics override existing statistics for all projections on the table.

The `IMPORT` and `EXPORT` functions are lightweight because they operate only on metadata.

For details about these functions, see the SQL Reference Manual.

Determining When Statistics Were Last Updated

The V_CATALOG.PROJECTION_COLUMNS system table returns information about projection columns, including the type of statistics, and the the time at which column statistics were last updated.

The following example illustrates how you can examine the run status for statistics on your tables.

On a single-node cluster, the following sample schema defines a table named trades, which groups the highly-correlated columns `bid` and `ask` and stores the `stock` column separately:

```
=> CREATE TABLE trades (stock CHAR(5), bid INT, ask INT);
=> CREATE PROJECTION trades_p (stock ENCODING RLE, GROUPED(bid ENCODING
     DELTAVAL, ask)) AS (SELECT * FROM trades) ORDER BY stock, bid;
=> INSERT INTO trades VALUES('acme', 10, 20);
=> COMMIT;
```

Query the `PROJECTION_COLUMNS` table for table `trades`:

```
=> \x
Expanded display is on.
=> SELECT * FROM PROJECTION_COLUMNS WHERE table_name = 'trades';
```

Notice that the `statistics_type column` returns `NONE` for all three columns in the `trades` table. Also, there is no value in the `statistics_updated_timestamp` field because statistics have not yet been run on this table.

```
-[ RECORD 1 ]----------------+------------------
projection_name              | trades_p
projection_column_name       | stock
column_position              | 0
sort_position                | 0
column_id                    | 45035996273743508
data_type                    | char(5)
encoding_type                | RLE
access_rank                  | 0
group_id                     | 0
table_schema                 | public
table_name                   | trades
table_column_name            | stock
statistics_type              | NONE
statistics_updated_timestamp |
-[ RECORD 2 ]----------------+------------------
projection_name              | trades_p
projection_column_name       | bid
column_position              | 1
sort_position                | 1
column_id                    | 45035996273743510
data_type                    | int
encoding_type                | DELTAVAL
access_rank                  | 0
group_id                     | 45035996273743512
table_schema                 | public
table_name                   | trades
table_column_name            | bid
statistics_type              | NONE
statistics_updated_timestamp |
-[ RECORD 3 ]----------------+------------------
projection_name              | trades_p
projection_column_name       | ask
column_position              | 2
sort_position                |
column_id                    | 45035996273743514
data_type                    | int
encoding_type                | AUTO
access_rank                  | 0
```

```
group_id                     | 45035996273743512
table_schema                 | public
table_name                   | trades
table_column_name            | ask
statistics_type              | NONE
statistics_updated_timestamp |
```

Now run statistics on the `stock` column:

```
=> SELECT ANALYZE_STATISTICS('trades.stock');
```

The system returns 0 for success:

```
-[ RECORD 1 ]------+--
ANALYZE_STATISTICS | 0
```

Now query PROJECTION_COLUMNS again:

```
=> SELECT * FROM PROJECTION_COLUMNS where table_name = 'trades';
```

This time, `statistics_type` changes to FULL for the `trades.stock` column (representing full statistics were run), and the `statistics_updated_timestamp` column returns the time the `stock` columns statistics were updated. Note that the timestamp for the `bid` and `ask` columns have not changed because statistics were not run on those columns. Also, the `bid` and `ask` columns changed from NONE to ROWCOUNT. This is because Vertica automatically updates ROWCOUNT statistics from time to time. The statistics are created by looking at existing catalog metadata.

```
-[ RECORD 1 ]----------------+-----------------------------
projection_name              | trades_p
projection_column_name       | stock
column_position              | 0
sort_position                | 0
column_id                    | 45035996273743508
data_type                    | char(5)
encoding_type                | RLE
access_rank                  | 0
group_id                     | 0
table_schema                 | public
table_name                   | trades
table_column_name            | stock
statistics_type              | FULL
statistics_updated_timestamp | 2011-03-31 13:32:49.083544-04
-[ RECORD 2 ]----------------+-----------------------------
projection_name              | trades_p
projection_column_name       | bid
column_position              | 1
sort_position                | 1
column_id                    | 45035996273743510
data_type                    | int
encoding_type                | DELTAVAL
access_rank                  | 0
group_id                     | 45035996273743512
table_schema                 | public
table_name                   | trades
table_column_name            | bid
statistics_type              | ROWCOUNT
statistics_updated_timestamp | 2011-03-31 13:31:50.017845-04
```

```
-[ RECORD 3 ]----------------+-----------------------------
projection_name              | trades_p
projection_column_name       | ask
column_position              | 2
sort_position                |
column_id                    | 45035996273743514
data_type                    | int
encoding_type                | AUTO
access_rank                  | 0
group_id                     | 45035996273743512
table_schema                 | public
table_name                   | trades
table_column_name            | ask
statistics_type              | ROWCOUNT
statistics_updated_timestamp | 2011-03-31 13:31:50.017872-04
```

If you run statistics on the bid column and then query this system table again, only RECORD 2 is updated:

```
=> SELECT ANALYZE_STATISTICS('trades.bid');
-[ RECORD 1 ]------+--
ANALYZE_STATISTICS | 0
=> SELECT * FROM PROJECTION_COLUMNS where table_name = 'trades';

-[ RECORD 1 ]----------------+-----------------------------
projection_name              | trades_p
projection_column_name       | stock
column_position              | 0
sort_position                | 0
column_id                    | 45035996273743508
data_type                    | char(5)
encoding_type                | RLE
access_rank                  | 0
group_id                     | 0
table_schema                 | public
table_name                   | trades
table_column_name            | stock
statistics_type              | FULL
statistics_updated_timestamp | 2011-03-31 13:32:49.083544-04
-[ RECORD 2 ]----------------+-----------------------------
projection_name              | trades_p
projection_column_name       | bid
column_position              | 1
sort_position                | 1
column_id                    | 45035996273743510
data_type                    | int
encoding_type                | DELTAVAL
access_rank                  | 0
group_id                     | 45035996273743512
table_schema                 | public
table_name                   | trades
table_column_name            | bid
statistics_type              | FULL
statistics_updated_timestamp | 2011-03-31 13:35:35.817222-04
-[ RECORD 3 ]----------------+-----------------------------
```

```
projection_name              | trades_p
projection_column_name       | ask
column_position              | 2
sort_position                |
column_id                    | 45035996273743514
data_type                    | int
encoding_type                | AUTO
access_rank                  | 0
group_id                     | 45035996273743512
table_schema                 | public
table_name                   | trades
table_column_name            | ask
statistics_type              | ROWCOUNT
statistics_updated_timestamp | 2011-03-31 13:31:50.017872-04
```

You can quickly query just the timestamp column to see when the columns were updated:

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

=> SELECT ANALYZE_STATISTICS('trades');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
=> SELECT projection_column_name, statistics_type,
   statistics_updated_timestamp
   FROM PROJECTION_COLUMNS where table_name = 'trades';
 projection_column_name | statistics_type | STATISTICS_UPDATED_TIMESTAMP
------------------------+-----------------------------------
 stock                  | FULL            | 2011-03-31 13:39:16.968177-04
 bid                    | FULL            | 2011-03-31 13:39:16.96885-04
 ask                    | FULL            | 2011-03-31 13:39:16.968883-04
(3 rows)
```

See V_CATALOG.PROJECTION_COLUMNS in the SQL Reference Manual for details.

Reacting to Stale Statistics

During predicate selectivity estimation, the query optimizer can identify when the histograms are not available or are likely out of date (e.g., the value in the predicate is outside the histogram's max range).

- During predicate selectivity estimation, if the value in the predicate is outside the histogram's max range

During predicate selectivity estimation if no histograms are available When the optimizer detects stale statistics, it takes these actions:

- Generates a message (and log), recommending that you run ANALYZE_STATISTICS().
- Annotates EXPLAIN plans describing when statistics are outside the histogram's range.
- Ignores the state statistics when generating a query. Here, the optimizer executes queries using other considerations, such as FK-PK constraints, when available

The following EXPLAIN fragment shows no statistics (histograms unavailable):

```
 | | +-- Outer -> STORAGE ACCESS for fact [Cost: 604, Rows: 10K (NO STATISTICS)]
```

The following EXPLAIN fragment shows stale statistics (where the predicate is outside the range of the histogram):

Example with stale statistics (where the predicate falls off the end of the histogram):

```
 | | +-- Outer -> STORAGE ACCESS for fact [Cost: 35, Rows: 1 (STALE STATISTICS)]
```

Information about which table column has no statistics is available in a system table. You can, for example, view the timestamp for when statistics were last run by querying the V_CATALOG.PROJECTION_COLUMNS system table.

For example, run full statistics on table 'trades':

```
=> SELECT ANALYZE_STATISTICS('trades');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

Next, query the projection_column_name, statistics_type, and statistics_updated_timestamp columns:

```
=> SELECT projection_column_name, statistics_type,
   statistics_updated_timestamp
   FROM PROJECTION_COLUMNS where table_name = 'trades';
 projection_column_name | statistics_type | STATISTICS_UPDATED_TIMESTAMP
------------------------+-----------------+-------------------------------
 stock                  | FULL            | 2011-03-31 13:39:16.968177-04
 bid                    | FULL            | 2011-03-31 13:39:16.96885-04
 ask                    | FULL            | 2011-03-31 13:39:16.968883-04
(3 rows)
```

You can also query the V_CATALOG.PROJECTIONS.HAS_STATISTICS column, which returns `true` only when all non-epoch columns for a table have full statistics. Otherwise the column returns false.

See Also

Analyzing Workloads (page 266)

PROJECTIONS and PROJECTION_COLUMNS in the SQL Reference Manual

Canceling and Removing Statistics

You can cancel statistics mid analysis by issuing CTRL-C on vsql or invoking the INTERRUPT_STATEMENT() function.

Use the `DROP_STATISTICS()` function to remove statistics for the specified table or type.

Caution: Once statistics are dropped, it can be time consuming to regenerate them.

Troubleshooting Issues Using Statistics

To help expedite the resolution of your issue, before you contact Vertica Technical Support include the system diagnostics, schema (or table and projection definitions), output of the EXPLAIN plan, and the output of EXPORT_STATISTICS().

1. Run the Diagnostics Utility using the following command.

   ```
   # /opt/vertica/bin/diagnostics [ command ... ]
   ```

2. Send the resulting `.zip` file from the Diagnostics Utility command to Vertica ***Technical Support*** (on page 1).
3. Run `/opt/vertica/scripts/collect_diag_dump.sh` and send the resulting .tar.gz file to ***Technical Support*** (on page 1).

   Note: The `collect_diag_dump` file contains the catalog and statistics, as well other important information that helps Technical Support profile and troubleshoot your case.

Analyzing Workloads

The Workload Analyzer is a utility that intelligently monitors the performance of SQL queries and workload history, resources, and configurations to identify underperforming queries, as well as the root causes for poor query performance. The Workload Analyzer makes it easy to fine-tune query performance without requiring sophisticated skills on the part of the database administrator.

Database administrators run the Workload Analyzer using the ANALYZE_WORKLOAD() function, which returns tuning recommendations. For example, the Workload Analyzer might find that statistics are stale and recommend that you regather them on a particular table or column.

For syntax and examples, see ANALYZE_WORKLOAD() in the SQL Reference Manual.

Note: Recommendation output can also be queried using the V_MONITOR.TUNING_RECOMMENDATIONS system table

Optimizing Query Performance

Your SQL queries tell the database what you want, and the Vertica query optimizer plans the most efficient way to get that information to you. So by carefully writing queries, you can often help improve Vertica performance.

Sort Optimizations

Vertica can avoid having to sort all of the data in a query when the underlying projection is already sorted, as illustrated in this example.

The first statement creates a simple table with four columns:

```
CREATE TABLE tab (
    a INT NOT NULL,
    b INT NOT NULL,
    c INT,
    d INT
);
```

The next statement creates a projection and specifies ordering on columns `a,b,c`:

```
CREATE PROJECTION tab_p (
   a_proj,
   b_proj,
   c_proj,
   d_proj )
AS SELECT * FROM tab
ORDER BY a,b,c
UNSEGMENTED ALL NODES;
```

For queries to benefit from the underlying optimization, sort the columns in the same order as those defined by the `CREATE PROJECTION` statement. For example, if the query contains an `ORDER BY a` or `a,b`, or `a,b,c` clause, the query is optimized. If you include column `d` in the query, Vertica cannot skip sorting all the data because column d is not in the projection sort order, and the query loses the sort optimization.

The following example is optimized because the query sort order matches the projection sort order:

```
SELECT * FROM tab
ORDER BY a,b,c;
 a  | b  | c  | d
------------------
 13 | 37 | 84 | 87
 15 | 25 | 80 | 76
 33 | 42 | 62 | 65
 44 | 17 | 77 | 45
 88 | 27 | 37 | 39
(5 rows)
```

See Also

CREATE PROJECTION in the SQL Reference Manual

Physical Schema in the Concepts Guide

Designing a Physical Schema and Designing for GROUP BY Queries in the Administrator's Guide

GROUP BY Pipelined or Hash

The examples in this section refer to the table and projection schema introduced in ***Sort Optimizations*** (page 268).

Vertica chooses the faster GROUP BY pipelined over GROUP BY hash, if the conditions listed in this section are met.

Condition #1: Given a particular projection sort order, 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, Vertica groups by hash instead of pipelined, losing the performance benefits.

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 the group by pipeline operator because columns a,b,c are included in the projection sort columns.

`GROUP BY a,b,c,d`	The query optimizer uses hash because 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 appear *first* in the projection's ORDER BY clause. For example, given a projection sort order `ORDER BY a,b,c` and a query construct that uses `GROUP BY a,c` Vertica uses GROUP BY hash because column b from the projection sort order is skipped in the GROUP BY clause.

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`	Uses pipelined because all columns preceding “b” in projection sort order appear as constant equality predicates.
`SELECT a FROM tab WHERE a = 10` `GROUP BY a, b`	Uses pipelined even if redundant grouping column “a” is present.
`SELECT a FROM tab WHERE a = 10` `GROUP BY b, c`	Uses pipelined because all columns preceding “b” and “c” in projection sort order appear as constant equality predicates.
`SELECT a FROM tab WHERE a = 10` `GROUP BY c, b`	Uses pipelined because all columns preceding “b” and “c” in projection sort order appear as constant equality predicates.
`SELECT a FROM tab WHERE a = 10` `AND b = 100 GROUP BY c`	Uses pipelined because all columns preceding “b” and “c” in projection sort order appear as constant equality

	predicates.

See Also

Designing for Group By Queries in the Administrator's Guide

Null Placement

Performance Optimization for Analytic Sort Computation

Vertica stores data in projections that is sorted in a specific way. All columns are stored in ASC (ascending) order, but the placement of nulls depends on the column's data type.

The analytic ORDER BY (window_order_clause) and the SQL ORDER BY clause also perform slightly different sort operations:

- The analytic window_order_clause sorts data that is used by 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 default sort order:
  - ASC + NULLS LAST. Null values are placed at the end of the sorted result
  - DESC + NULLS FIRST. Null values are placed at the beginning of the sorted result
- The SQL ORDER BY clause specifies only ascending or descending order; however, the following is the default for null placement in Vertica:
  - NUMERIC, INTEGER, DATE, TIME, TIMESTAMP, and INTERVAL columns. NULLS FIRST (null values are stored at the beginning of a sorted projection).
  - FLOAT, STRING, and BOOLEAN columns. NULLS LAST (null values are stored at the end of a sorted projection).
  - No matter what the data type, if you specify NULLS AUTO, Vertica chooses the most efficient placement of nulls (for example, either NULLS FIRST or NULLS LAST) based on your query.

If you do not care about null placement in queries that involve analytics computation, or if you know that columns contain no null values, specify NULLS AUTO, and Vertica chooses the placement that gives the fastest performance. Otherwise you can specify NULLS FIRST or NULLS LAST.

You can also carefully formulate queries so Vertica can avoid sorting the data and can process the query more quickly, as illustrated by the following example.

Example

In the following example, Vertica sorts inputs from table t on column x, as specified in the OVER(ORDER BY) clause. Then it 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 above SELECT statement, Vertica can eliminate the ORDER BY clause and run the query quickly because column x is a FLOAT data type; thus, the projection sort order matches the analytic default ordering (ASC + NULLS LAST). Vertica can also avoid having to sort the data when the underlying projection is already sorted.

Assume, however, that column x had been defined as INTEGER. Vertica cannot avoid sorting the data because the projection sort order for INTEGER data types (ASC + NULLS FIRST) does not match default analytic ordering (ASC + NULLS LAST). To help Vertica eliminate the sort, specify the placement of nulls to match default ordering:

```
=> SELECT x, RANK() OVER (ORDER BY x NULLS FIRST) FROM t;
```

If column x is defined as a STRING, the following query would eliminate the sort:

```
=> SELECT x, RANK() OVER (ORDER BY x NULLS LAST) FROM t;
```

Note that omitting NULLS LAST in the above query still 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 Vertica.

Top-K Optimizations

Queries that use the SQL LIMIT clause with ORDER BY or the SQL-99 analytic function ROW_NUMBER() return a specific subset of rows in the query result. This is known as Top-K Optimization, which works on all data types. By not having to to sort the entire data set, a Top-K operation can significantly improve performance because Vertica does much less work than when producing the full result set.

For example, in the following typical Top-K query, Vertica extracts only the 3 smallest rows from column `x`, as specified by the LIMIT clause:

```
=> SELECT * FROM t1 ORDER BY x LIMIT 3;
```

If table `t1` contained millions of rows, you can imagine how time consuming it would be to sort all the `x` values. Instead, Vertica, returns only the the smallest 3 values in x.

Note: Omitting the ORDER BY clause could produce nondeterministic results because the query retrieves any number of records set by the LIMIT clause, thereby losing Top-K performance benefits.

The following list illustrates the LIMIT clause queries that Vertica supports:

```
=> SELECT * FROM (SELECT * FROM t1 ORDER BY x ) alias LIMIT 3;
=> SELECT * FROM (SELECT * FROM t1 ORDER BY x LIMIT 5) alias LIMIT 3;
=> SELECT * FROM (SELECT * FROM t1 ORDER BY x) alias LIMIT 4 OFFSET 3;
=> SELECT * FROM t1 UNION SELECT * FROM t2 LIMIT 3;
=> SELECT * FROM fact JOIN dim using (x) LIMIT 3;
=> SELECT * FROM t1 JOIN t2 USING (x) LIMIT 3;
```

GROUP BY operations are not affected by Top-K.

Sort operations that often precede an analytics computation benefit from Top-K optimization if the query contains an OVER(ORDER BY) clause, such as in the following ROW_NUMBER() query:

```
=> SELECT x FROM
     (SELECT *, ROW_NUMBER() OVER (ORDER BY x) AS row
      FROM t1) t2 WHERE row <= 3;
```

The above query has the same behavior as the following query, which uses LIMIT:

```
=> SELECT ROW_NUMBER() OVER (ORDER BY x) AS RANK FROM t1 LIMIT 3;
```

You can also 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;
```

Notes

- When the OVER() clause includes the `window_partition_clause`, Top-K optimization occurs only the analytic sort node matches the projection; for example, if the projection is sorted on `x`, `y` in table `t1`.

- The configuration parameter `TopKHeapMaxMem` controls how much memory can be used for TopK(Heap). If *K* rows can fit into the space allocated by this parameter (default 80MB), the optimizer uses TopK(Heap); otherwise no TopK is used (the query is sorted and loses Top-K optimization).

  Once the optimizer chooses TopK(Heap), the Resource Manager can reject the plan if the TopK operator requires too much memory. To prevent the query from being rejected, you can lower the parameter `TopKHeapMaxMem`, but be careful in changing the setting. Too low and no TopK used (you lose the optimization); too high and the query could get rejected. In most cases, the default setting of 80MB should work, and the the configuration parameter is provided as a tool.

See Also

Designing for GROUP BY Queries in the Administrator's Guide

Configuration Parameters in the Administrator's Guide

Joins Optimizations

You can affect join performance in Vertica by specifying which input is OUTER (left input) and which is INNER (right input).

- On Foreign-Key/Primary-Key joins, if you estimate the FK side to be larger (or if it has no ***statistics*** (page 254)) write your query so the FK table is the OUTER table.
- On OUTER/SEMI/ANTI joins, the preserved side is the OUTER table.

Otherwise, Vertica recommends that you write your queries so the larger (number of rows times size per row) input is the OUTER table. Consider also making the smaller input OUTER if it is not yet fully materialized.

Joins and Equality Predicates

Joins run faster if the columns on the left side of an equality predicate come from one table and the columns on the right side of the equality predicate come from another; for example:

```
=> SELECT * FROM T JOIN X WHERE T.a + T.b = X.x1 - X.x2;
```

If you include columns from different tables, your query loses the performance improvements:

```
=> SELECT * FROM T JOIN X WHERE T.a = X.x1 + T.b
```

Merge Joins for Insert-Select Queries

The ordering used for the select part (that also has joins) of an insert-select query is determined by the choice of the outer (fact) projection for the select's join. This means that it is not possible for it to use optimizations, such as merge-join, based on the order of the 'inner' projection. To facilitate a merge-join, add an ORDER BY clause to the SELECT if the incoming data isn't already sorted correctly for the Merge-Join. This creates a SORT operator to facilitate the merge-join.

The following example illustrates this concept by generating a hash-join instead of a merge join for a FK-PK validation. It also illustrates how to use ORDER BY to force a merge-join.

```
-- Should be getting a MERGE JOIN for the 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), b varchar(10));
CREATE TABLE d1(a varchar(10), b varchar(10));
CREATE TABLE f1_staging(a varchar(10), b varchar(10));

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');
-- Getting HASH JOIN instead of MERGE JOIN
\o explain.out
explain
INSERT INTO f1
SELECT f1s.a, f1s.b
FROM f1_staging f1s join d1
on f1s.a=d1.a and f1s.b=d1.b;
\o
```

```
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 ORDER BY results in the desired MERGE JOIN
\o explain_orderby.out
explain
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;
\o

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;
```

Using Identically Segmented Projections

You can help improve query performance when you join multiple tables if the system contains projections that are identically segmented by the join keys. Identically segmenting projections allow the joins to occur locally on each node without any data movement across the network at query time.

The 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 useful with distributed execution plans only.

Join Conditions for Identically Segmented Projections (ISP)

In particular, a projection called `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 hold 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 be of the following forms:
  `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 cannot be a FLOAT.
  - If `t1.j1` is a CHAR(10) then `t2.j1` can be any CHAR or VARCHAR (e.g., 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}`, then each such segmentation column `t1.sX` is in the join column set {`t1.jX`}.
- If `p2` is segmented by an expression on columns `{t2.s1, t2.s2, ... t2.sN}`, then each such segmentation column `t2.sX` is in the join column set {`t2.jX`}.
- The segmentation expressions of `p1` and `p2` must be structurally equivalent:

  Example:

  If p1 is SEGMENTED BY hash(t1.x), if p2 is SEGMENTED BY hash(t2.x), p1 and p2 are identically segmented.

  If p1 is SEGMENTED BY hash(t1.x), if 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 `p1` and `p2` are range segmented, the ranges must be identical.

If Vertica finds projections for `t1` and `t2` that are not segmented identically, the data is redistributed across the network during query run-time, as necessary.

Tip: If creating custom designs, try to use segmented projections for ISP joins whenever possible. See "Designing Identically Segmented Projections for K-Safety" below.

The following syntax provides an example of two tables and 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 the above 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 segmented identically to one another.

The following syntax illustrates suboptimal buddy projection design because the projections are not identically segmented to the each other in that their OFFSETs differ:

```
CREATE PROJECTION t1_b1 (id, x1, y1)   CREATE PROJECTION t2_b1 (id, x2, y2)
AS SELECT * FROM t1                    AS SELECT * FROM t2
SEGMENTED BY HASH(id)                  SEGMENTED BY HASH(id)
ALL NODES OFFSET 1;                    ALL NODES OFFSET 2;
```

The following syntax is another example of suboptimal buddy projection design. The projections are not identically segmented to each other in that their segmentation expressions differ; thus, the projections do not qualify as buddies:

```
CREATE PROJECTION t1_b2 (id, x1, y1)   CREATE PROJECTION t2_b2 (id, x2, y2)
AS SELECT * FROM t1                    AS SELECT * FROM t2
SEGMENTED BY HASH(id, x1)              SEGMENTED BY HASH(id)
ALL NODES OFFSET 1;                    ALL NODES OFFSET 2;
```

Buddy projections can use different sort orders. For details, see Hash Segmentation in the SQL Reference Manual.

Examples

- Vertica recommends that you use Database Designer to create projections, which uses HASH and ALL NODES syntax.
- Hash segmentation is the preferred method of segmentation. For detailed information about using hash segmentation in a projection, see the CREATE PROJECTION statement in the SQL Reference Manual.

See Also

Partitioning and Segmenting Data

CREATE PROJECTION in the the SQL Reference Manual

Optimizing Query Speed with Predicates

In the following example, if the predicate column in the outer query *only* references the PARTITION BY columns of the subquery, the predicate can be pushed into the subquery so that it is evaluated before the time series or analytic computation, improving query performance.

```
SELECT symbol, AVG(first_bid) as avg_bid FROM
   (SELECT symbol, slice_time, TS_FIRST_VALUE(bid1) AS first_bid
    FROM Tickstore
    WHERE symbol IN ('MSFT', 'IBM')
    TIMESERIES slice_time AS 5 seconds
    OVER (PARTITION BY symbol ORDER BY ts)) AS resultOfGFI
WHERE symbol IN ('MSFT', 'IBM')
GROUP BY symbol;
```

In the above query, for example, the outer WHERE clause predicate is pushed into the subquery.

Note: The only predicates pushed into the subquery are predicates on PARTITION BY columns.

This predicate optimization is also true for analytic functions, where only the set intersection of PARTITION BY columns are pushed down. For example:

```
RANK() OVER(PARTITION BY a, b, c ORDER BY d)
DENSE_RANK() OVER(PARTITION BY d, b, c ORDER BY a)
```

In the above example, even though DENSE_RANK has column `d` in its partition clause and RANK has `a` in its partition clause, only predicates referring to `b` or `c` can be pushed down.

More formally:

```
{a, b, c} ^ {d, b, c} = {b, c}
```

Constant Propagation and IN-list Constant Folding

At query planning time, Vertica can simplify portions of predicates that it determines cannot be true. These optimization are typically relevant for automatically generated SQL. For example:

```
... WHERE id = '5' AND (month = 'jan' OR id IN (7,8))
```

Gets converted into:

```
... WHERE id = '5' AND month = 'jan'
```

INSERT-SELECT Optimizations

When doing INSERT-SELECT operations, if the projection sort order of the target table is the same as the input select query, the SORT phase of the insert can be avoided.

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 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;
```

This insert will not require a sort for the data target (writing data to projection):

```
=> INSERT /*+direct*/ INTO destination SELECT * FROM source;
```

This insert require a sort. Note the switched column orders:

```
=> INSERT /*+direct*/ INTO destination SELECT col1, col3, col2 FROM source;
```

This insert does not require a sort at the destination. Note the switched column orders but explicit GROUP BY that causes the output to be sorted by c1, c3, c2 in Vertica:

```
=> INSERT /*+direct*/ INTO destination SELECT col1, col3, col2 FROM source GROUP
BY col1, col3, col2 ;
```

Optimizing Deletes and Updates

Vertica is optimized for query intensive workloads, so deletes and updates might not achieve the same level of performance as queries. Deletes and updates go to the WOS by default, but if the data is sufficiently large and would not fit in memory, Vertica automatically switches to using the ROS. See Using INSERT, UPDATE, and DELETE.

The topics that follow discuss best practices when using delete and update operations in Vertica.

Performance Considerations for Deletes and Updates

Query Performance after Large Deletes

A large number of (un-purged) deleted rows could negatively affect query and recovery performance.

To eliminate the rows that have been deleted from the result, a query must do extra processing. It has been observed if 10% or more of the total rows in a table have been deleted, the performance of a query on the table slows down. However your experience may vary depending upon the size of the table, the table definition, and the query. The same problem can also happen during the recovery. To avoid this, the delete rows need to be purged in Vertica. For more information, see Purge Procedure.

See ***Optimizing Deletes and Updates for Performance*** (page 282) for more detailed tips to help improve delete performance.

Concurrency

Deletes and updates take exclusive locks on the table. Hence, only one delete or update transaction on that table can be in progress at a time and only when no loads (or INSERTs) are in progress. Deletes and updates on different tables can be run concurrently.

Pre-join Projections

Avoid pre-joining dimension tables that are frequently updated. Deletes and updates to Pre-join projections cascade to the fact table causing a large delete or update operation.

Optimizing Deletes and Updates for Performance

The process of optimizing a design for deletes and updates is the same. Some simple steps to optimize a projection design or a delete or update statement can increase the query performance by tens to hundreds of times. The following section details several proposed optimizations to significantly increase delete and update performance.

Note: For large bulk deletion, Vertica recommends using Partitioned Tables where possible because it can provide the best delete performance and also improve query performance.

Designing Delete- or Update-Optimized Projections

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.

Example

```
CREATE TABLE t (a integer, b integer, c integer);
CREATE PROJECTION p1 (a ENCODING RLE,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 example, both p1 and p2 are eligible for delete and update optimization because the *a* column is available:

```
DELETE from t WHERE a = 1;
```

In the following example, only p1 is eligible for delete and update optimization because the b column is not available in p2:

```
DELETE from t WHERE b = 1;
```

Delete and Update Considerations for Sort Order of Projections

You should design your projections so that frequently used delete or update predicate columns appear in the SORT ORDER of all projections for large deletes and updates.

For example, suppose most of the deletes you perform on a projection look like the following example:

```
DELETE from t where time_key < '1-1-2007'
```

To optimize the deletes, you would make “time_key” appear in the `ORDER BY` clause of all your projections. This schema design enables Vertica to optimize the delete operation.

Further, 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. See Choosing Sort-orders for Low Cardinality Predicates. You can use the EVALUATE_DELETE_PERFORMANCE function to analyze projections for sort order issues.

The following three examples demonstrate some common scenarios for delete optimizations. Remember that these same optimizations work for optimizing for updates as well.

In the first scenario, the data is deleted given a time constraint, in the second scenario the data is deleted by a single primary key and in the third scenario the original delete query contains two primary keys.

Scenario 1: Delete by Time

This example demonstrates increasing the performance of deleting data given a date range. You may have a query that looks like this:

```
delete from trades
where trade_date between '2007-11-01' and '2007-12-01';
```

To optimize this query, start by determining whether all of the projections can perform the delete in a timely manner. Issue a SELECT COUNT(*) on each projection, given the date range and notice the response time. For example:

```
SELECT COUNT(*) FROM [projection name i.e., trade_p1, trade_p2]
WHERE trade_date BETWEEN '2007-11-01' AND '2007-12-01;
```

If one query 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 Run Length Encoded (RLE). RLE replaces sequences of the same data values within a column by a single value and a count number.

If the number of unique columns is unsorted, or the average number of repeated rows is less than ten, `trade_date` is too close to being unique and cannot be RLE. If you find this to be the case, add a new column to minimize the search scope.

In this example, add a column for trade year = 2007. However, first determine if the `trade_year` returns a manageable result set. The following query returns the data grouped by trade year.

```
SELECT DATE_TRUNC('year', trade_date),count(*)
FROM trades
GROUP BY DATE_TRUNC('year',trade_date);
```

Assuming that trade_year = 2007 is near 8k (8k integer is 64k), a column for `trade_year` can be added to the trades table. The final DELETE statement then becomes:

```
DELETE FROM trades
WHERE trade_year = 2007
AND trade_date BETWEEN '2007-11-01' AND '2007-12-01';
```

Vertica makes the populating of extra columns easier with the ability to define them as part of the COPY statement.

Scenario 2: Delete by a Single Primary Key

This example demonstrates increasing the performance of deleting data given a table with a single primary key. Suppose you have the following query:

```
DELETE FROM [table]
WHERE pk IN (12345, 12346, 12347,...);
```

You begin optimizing the query by creating a new column called `'buckets'`, which is assigned the value of one the primary key column divided by 10k; in the above example, buckets=(int) pk/10000. This new column can then be used in the query to limit the search scope. The optimized delete would be:

```
DELETE FROM [table]
WHERE bucket IN (1,...)
AND pk IN (12345, 12346, 12347,...);
```

Scenario 3: Delete by Multiple Primary Keys

This example demonstrates deleting data given a table with multiple primary keys. Suppose you have the following query:

```
DELETE FROM [table]
WHERE (pk1, pk2) IN ((12345,5432),(12346,6432),(12347,7432), ...);
```

Similar to the previous example, you create a new column called `'buckets'`, which is assigned the value of one of the primary key column values divided by 10k; in the above example, buckets=(int) pk1/10000. This new column can then be used in the query to limit the search scope.

In addition, you can further optimize the original search by reducing the primary key `IN` list from two primary key columns to one column by creating a second column. For example, you could create a new column named 'pk1-2' that contains the concatenation of the two primary key columns. For example, pk1-2 = 'pk1' || '-' || 'pk2'.

Your optimized delete statement would then be:

```
DELETE FROM [table]
WHERE bucket IN (1,. . .)
AND pk1-2 IN ('12345-5432', '12346-6432', '12347-7432',...);
```

Caution: Remember that 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. See Purging Deleted Data for information on how to configure the appropriate amount of history to be retained.

Using External Procedures

An external procedure is a procedure external to 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*** (page 287).
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*** (page 288).
4. ***Create the external procedure in Vertica*** (page 289).

Once a procedure is created in Vertica, you can ***execute*** (page 290) or ***drop*** (page 291) 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 Vertica to return *0* when the script succeeds.

Handling Procedure Output

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 Vertica resource manager is unaware of resources used by external procedures. Additionally, 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 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

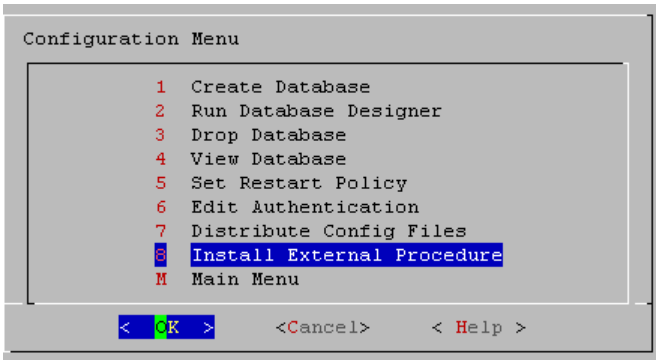
```
#!/bin/bash
echo "hello planet argument: $1" >> /tmp/myprocedure.log
exit 0
```

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

1. Run the Administration Tools.
   ```
   $ /opt/vertica/bin/adminTools
   ```
2. On the AdminTools **Main Menu**, click **Configuration Menu**, and then click **OK**.
3. On the **Configuration Menu**, click **Install External Procedure** and then click **OK**.



4. Select the database on which you want to install the external procedure.
5. Either select the file to install or manually type the complete file path, and then click **OK**.
6. 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.
7. 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 Vertica aware of it. To do so, use the `CREATE PROCEDURE` statement, but review ***Creating External Procedures*** (page 289) first.

Creating External Procedures

Once you have installed an external procedure, you need to make Vertica aware of it. To do so, use the `CREATE PROCEDURE` statement.

By default, only the superuser can create and execute a procedure. However, the 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*** (page 287).

```
=> 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 and GRANT (Procedure) in the SQL Reference Manual

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. 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. 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.

Note: By default, only the superuser can execute an external procedure. However, the superuser can grant the right to execute an external procedure to a user on the operating system. (See Procedure Privileges in the Administrator's Guide for details.).

See Also

CREATE PROCEDURE in the SQL Reference Manual

Procedure Privileges in the Administrator's Guide

Dropping External Procedures

Only a superuser can drop an external procedure. To drop the definition for an external procedure from 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 Vertica uses for a procedure cannot be altered; it can only be dropped.

Example

```
=> DROP PROCEDURE helloplanet(arg1 varchar);
```

See Also

DROP PROCEDURE in the SQL Reference Manual

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 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 `zeroifnull` that accepts an `INTEGER` argument and returns an `INTEGER` result.

```
=> CREATE FUNCTION zeroifnull(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 (`zeroifnull`) 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 `zeroifnull` function in a `SELECT` statement, where the function calls `col1` from table tabwnulls:

```
=> SELECT zeroifnull(col1) FROM tabwnulls;
 zeroifnull
------------
          1
          0
          0
(3 rows)
```

Use the `zeroifnull` function in the `GROUP BY` clause:

```
=> SELECT COUNT(*) FROM tabwnulls GROUP BY zeroifnull(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 zeroifnull(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 Vertica catalog, see ***Viewing Information About SQL Functions*** (page 294) in this guide.

See Also

CREATE FUNCTION and USER_FUNCTIONS in the SQL Reference Manual

Altering and Dropping User-Defined SQL Functions

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 zeroifnull();
ROLLBACK:  Function with specified name and parameters does not exist: zeroifnull
```

Note: Only the superuser or owner can alter or drop a SQL Macro.

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 `zeroifnull`. The following command renames the `zeroifnull` function to `zerowhennull`:

```
=> ALTER FUNCTION zeroifnull(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 Macro

The DROP FUNCTION command drops a SQL function from the 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
```

Vertica does not check for dependencies, so if you drop a SQL function where other objects reference it (such as views or other SQL Macros), 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 and DROP FUNCTION in the SQL Reference Manual

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 `zeroifnull` function:

```
=> GRANT EXECUTE ON FUNCTION zeroifnull (x INT) TO Fred;
```

To revoke EXECUTE privileges from user Fred on the `zeroifnull` function:

```
=> REVOKE EXECUTE ON FUNCTION zeroifnull (x INT) FROM Fred;
```

See Also

GRANT (Function) and REVOKE (Function) in the SQL Reference Manual

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          | zeroifnull
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 zeroifnull(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          | zeroifnull
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 `zeroifnull` function:

```
=> CREATE OR REPLACE FUNCTION zeroifnull(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 `zeroifnull` 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 `zeroifnull` function:

```
=> SELECT * FROM USER_FUNCTIONS;
-[ RECORD 1
]----------+------------------------------------------------------------
schema_name            | public
function_name          | zeroifnull
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          | zeroifnull
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 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 zeroifnull();
ROLLBACK:  Function with specified name and parameters does not exist: zeroifnull
```

See Also

USER_FUNCTIONS in the SQL Reference Manual

Migrating Built-in SQL Functions

If you have built-in SQL functions from another RDBMS that do not map to a Vertica-supported function, you can migrate them into your 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:

- DAY()
- DAYOFYEAR()
- YEAR()
- UCASE()
- LCASE()
- LOCATE()
- POSSTR()
- CONCAT()

DAY()

The first script creates a user-defined SQL function for the DAY() function:

```
=> CREATE OR REPLACE FUNCTION DAY(x DATE)
   RETURN INT
   AS BEGIN
   RETURN EXTRACT(DAY FROM x);
   END;
=> CREATE OR REPLACE FUNCTION DAY(x TIMESTAMP)
   RETURN INT
```

```
   AS BEGIN
   RETURN EXTRACT(DAY FROM x);
   END;
=> CREATE OR REPLACE FUNCTION DAY(x INTERVAL)
   RETURN INT
   AS BEGIN
   RETURN EXTRACT(DAY FROM x);
   END;
```

DAYOFYEAR()

This script creates a user-defined SQL function for the `DAYOFYEAR()` function:

```
=> CREATE OR REPLACE FUNCTION DAYOFYEAR(x DATE)
   RETURN INT
   AS BEGIN
   RETURN EXTRACT(DOY FROM x);
   END;
=> CREATE OR REPLACE FUNCTION DAYOFYEAR(x TIMESTAMP)
   RETURN INT
   AS BEGIN
   RETURN EXTRACT(DOY FROM x);
   END;
```

YEAR()

This script creates a user-defined SQL function for the `YEAR()` function:

```
=> CREATE OR REPLACE FUNCTION YEAR(x DATE)
   RETURN INT
   AS BEGIN
   RETURN EXTRACT(YEAR FROM x);
   END;
=> CREATE OR REPLACE FUNCTION YEAR(x TIMESTAMP)
   RETURN INT
   AS BEGIN
   RETURN EXTRACT(YEAR FROM x);
   END;
=> CREATE OR REPLACE FUNCTION YEAR(x INTERVAL)
   RETURN INT
   AS BEGIN
   RETURN EXTRACT(YEAR FROM x);
   END;
```

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;
```

CONCAT()

This script creates a user-defined SQL function for the `CONCAT()` function:

```
=> CREATE OR REPLACE FUNCTION CONCAT(a VARCHAR, b VARCHAR)
   RETURN VARCHAR
   AS BEGIN
   RETURN a||b;
   END;
```

Developing and Using User Defined Functions

User-Defined Functions (UDFs) are libraries of functions that you develop in C++ and load into 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.

UDF's primary strengths are:

- They run almost as fast as Vertica's own functions because they are loaded as a library by the Vertica server process.
- 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 Vertica's distributed computing features. Functions are executed in parallel on each node in the cluster.
- 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 Vertica. Your main programming task is to read in data, process it, and then write it using simple APIs.

There are several drawbacks to UDFs:

- Because they run within the Vertica process, poorly-coded UDF's can cause the database to become unstable. External procedures run in a separate process, and are unlikely to cause database stability issues. SQL macros use Vertica's own statements and functions and are likewise unlikely to cause stability issues. To avoid causing instability and data loss, you should test your UDF code extensively before deploying it to a production environment.
- The only supported language for UDFs is C++. External procedures can be programmed in any language that is supported on your cluster, and SQL macros are coded in SQL.

Since they run on the Vertica cluster, you should not use UDF's for highly memory or CPU-intensive tasks. UDFs that do consume lots of computing resources can result in poor database performance. They also should not perform potentially blocking operations (such as network communications), since they would cause the Vertica process to block as well.

This section explains how to create and use user-defined functions (UDFs).

How UDFs Work

User Defined Functions are contained in libraries. Multiple functions can be defined in a library, and multiple libraries can be loaded by Vertica. You load a library by:

1 Copying the library .so 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 UDFs using the CREATE FUNCTION SQL statement. This statement assigns SQL function names to the UDF classes in the library. From then on, you are able to use your function within your SQL statements. Whenever you call a UDF, Vertica creates an instance of the UDF class on each node in the cluster and passes it data to process.

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 (Function) and GRANT (Transform Function) in the SQL Reference Manual for details.

Types of UDFs

There are two 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 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*** (page 319).

There are many similarities in developing the two 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*** (page 302) for details).

Setting up a UDF Development Environment

You should develop your UDF code on the same Linux platform that you use on your Vertica database cluster. This will ensure that your UDF library is compatible with the Vertica version deployed on your cluster.

At a minimum, you need to install the following on your development machine:

- ***g++ http://gcc.gnu.org/*** and its associated tool chain such as ld. (**Note:** some Linux distributions package g++ separately from gcc.)
- A copy of the Vertica SDK. See ***The Vertica SDK*** (page 301) for details.

Note: The 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 Vertica database for testing and debugging. You may want to install a single-node 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 trhe cluster. See ***Compiling Your UDF*** (page 317) for details.)

The Vertica SDK

The Vertica 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 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 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.

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 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 Vertica SDK API Documentation

This documentation only provides a brief overview of the classes and class functions defined by the User Defined Function API. To learn more, see the Vertica SDK API. You can find this documentation in two locations:

- In the same directory as the other Vertica SDK files: `/opt/vertica/sdk/doc`.

- Included with the full documentation set, available either online or for download. See Installing Vertica Documentation in the Installation Guide.

Note: If you are viewing the HTML version of the documentation, you can follow this link to view the ***SDK API documentation*** (***../../SDK/html/index.htm***).

Developing a UDF

To create a UDF, 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 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*** (page 300) for details).

The following sections explain how you develop and compile the code for your UDF.

Vertica SDK Data Types

The Vertica SDK has typedefs and classes for representing Vertica data types within your UDF code. Using these typedefs ensures datatype compatibility between the data your UDF processes and generates and the Vertica database. The following table describes some of the typedefs available. Consult ***the Vertica SDK API Documentation*** (page 301) for a complete list, as well as lists of helper functions to convert and manipulate these data types.

Type Definition	Description
Interval	A Vertica interval
IntervalYM	A Vertica year-to-month interval.
Timestamp	A Vertica timestamp
vint	A standard Vertica 64-bit integer
vint_null	A null value for integer values
vbool	A Boolean value in Vertica
vbool_null	A null value for a Boolean data types
vfloat	A Vertica floating point value
VString	String data types (such as varchar and char)
VNumeric	Fixed-point data types from Vertica

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 Vertica function can be used. Their primary use is to perform data manipulations that would be difficult to perform or far too slow using SQL statements and functions. They also let you access analytic functionality provided by third-party libraries while maintaining high performance.

The topics in this section guide you through developing a UDSF.

UDSF Requirements

There are several requirements that your UDSF must meet:

- Your UDSF must return a value for every input row (unless it generates an error, see ***Handling Errors*** (page 315) for details). Failing to return a value for a row will result in incorrect results, and potentially destabilizing the Vertica server.
- Your UDSF must not allow an exception to be passed back to 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 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*** (page 314) for more information.

Remember that your UDSF runs within the 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 Vertica SDK: `Vertica::ScalarFunction` (`../../SDK/html/class_vertica_1_1_scalar_function.htm`) and `Vertica::ScalarFunctionFactory` (`../../SDK/html/class_vertica_1_1_scalar_function_factory.htm`).

The `ScalarFunctionFactory` `(../../SDK/html/class_vertica_1_1_scalar_function_factory.htm)` performs two roles:

- It lists the parameters accepted by the UDSF and the data type of the UDSF's return value. 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 Vertica can call to process data.

The `ScalarFunction` (`../../SDK/html/class_vertica_1_1_scalar_function.htm`) class is where you define the `processBlock` function, which performs the data processing that you want your UDSF to perform. When a user calls your UDSF function in a SQL statement, 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 readthe 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*** (page 314) 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 Vertica SDK code to make calls back into the 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 Vertica to allocate the memory for the new object. You generally will not interact with this object directly, but instead pass it along to Vertica SDK function and macro calls.

Subclassing ScalarFunction

The `ScalarFunction` (`../../SDK/html/class_vertica_1_1_scalar_function.htm`) 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, Vertica 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 Vertica SDK header file (`Vertica.h`) and using the Vertica namespace. While not required, using the namespace saves you from having to prefix every Vertica SDK class reference with `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 calues 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` (`../../SDK/html/class_vertica_1_1_block_reader.htm`) object using data-type-specific class functions.
- Process the data in some manner.
- Output the resulting value using one of the `BlockWriter` (`../../SDK/html/class_vertica_1_1_block_writer.htm`) 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 is no more rows data to be read (`BlockReader.next()` returns false).

Note: 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 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 Vertica (see ***Handling Errors*** (page 315) for more). In that case, Vertica does not attempt to read the incomplete result set generated by the UDSF.

Subclassing ScalarFunctionFactory

The `ScalarFunctionFactory` (`../../SDK/html/class_vertica_1_1_scalar_function_factory.htm`) class tells 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` (`../../SDK/html/class_vertica_1_1_scalar_function.htm`) subclass for Vertica.

After defining your factory class, you need to call the `RegisterFactory` macro. This macro instantiates a member of your factory class, so 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*** (page 304).

```
/*
* This class provides metadata about the ScalarFunction class, and
* also instantiates a member of that class when needed.
*/
class Add2intsInfo : 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 Vertica about the parameters and return type for your UDSF. In addition to a `ServerInterface` object, this function gets two `ColumnTypes` (`../../SDK/html/class_vertica_1_1_column_types.htm`) 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 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 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 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 ScalarFactorySubclass, you need to register it using the RegisterFactory macro. This macro instantiates your factory class and makes the metadata it contains available for Vertica to access. To call this macro, you just pass it the name of your factory class.

```
// Register the factory with Vertica
RegisterFactory(Add2intsInfo);
```

Developing a User Defined Transform Function

A User Defined Transform Function (UDTF) reads one or more rows of data, and returns zero or more rows of data. They can only be used in the SELECT list that contains just the UDTF call and optionally a PARTITION BY expression.

The topics in this section guide you through developing a UDTF.

UDTF Requirements

There are several requirements for UDTFs:

- 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. Doing so could result in a memory leak, as any memory allocated by the exception is not 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 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*** (page 314) for more information.

Remember that your UDTF runs within the Vertica process. Any problems it causes may result in database instability or even data loss.

UDTF Class Overview

You create your UDTF by subclassing two classes defined by the Vertica SDK: `Vertica::TransformFunction` (`../../SDK/html/class_vertica_1_1_transform_function.htm`) and `Vertica::TransformFunctionFactory` (`../../SDK/html/class_vertica_1_1_transform_function_factory.htm`).

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. 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 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, 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` (`../../SDK/html/class_vertica_1_1_partition_reader.htm`) and `Vertica::PartitionWriter` (`../../SDK/html/class_vertica_1_1_partition_writer.htm`) 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*** (page 314) 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 Vertica SDK code to make calls back into the 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 Vertica to allocate the memory for the new object. You generally will not interact with this object directly, but instead pass it along to Vertica SDK function and macro calls.

Subclassing TransformFunction

Your subclass of `Vertica::TransformFunction` (`../../SDK/html/class_vertica_1_1_transform_function.htm`) 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` (`../../SDK/html/class_vertica_1_1_partition_reader.htm`) object, and writes output values to a `Vertica::PartitionWriter` (`../../SDK/html/class_vertica_1_1_partition_writer.htm`) 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.

```
#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 &input_reader,
```

```
                                PartitionWriter &output_writer)
  {
    // Loop through the input rows
    do
      {
        // Get a single varchar as input.
        const VString &sentence = input_reader.getStringRef(0);
        // If input string is NULL, then output is NULL as well if
(sentence.isNull())
          {
            VString &word = output_writer.getStringRef(0);
            word.setNull();
            output_writer.next();
          }
        else
          {
            // Otherwise, tokenize the string and output the words
            stringstream ss(stringstream::in | stringstream::out);
            ss.write(sentence.data(), sentence.length());
            char buffer[sentence.length()]; // a word can't be longer than the
sentence
            // loop through string, outputting a row containing a word until //
reaching the end of the string.  while (!ss.eof())
              {
                // Get next part of string terminated by a space.
                ss.getline(buffer, sentence.length(), ' ');
                // Get the size of the word
                vsize wordlen = strlen(buffer);

                // Get a buffer from the PartitionWriter where
                // the output string should go. and copy it there
                VString &word = output_writer.getStringRef(0);
                word.copy(buffer, wordlen);
                // Done with this row. Advance to the next row of output
output_writer.next();
              }
          }
      }
    while (input_reader.next());   // Loop until no more input rows
  }
};
```

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 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` `../../SDK/html/class_vertica_1_1_v_string.htm` input parameter from the `PartitionReader` object. The `VString` class represents a 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*** (page 313) 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*** (page 313) 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. Once the input rows are exhausted, your UDTF will usually exit, so its results are returned back to Vertica.

Subclassing TransformFunctionFactory

Your subclass of the `TransformFunctionFactory` (`../../SDK/html/class_vertica_1_1_transform_function_factory.htm`) provides metadata about your UDTF to 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` (`../../SDK/html/class_vertica_1_1_column_types.htm`) objects that describe the columns your UDTF takes as input and returns as output.
- `createTransformFunction` instantiates a member of your `TransformFunction` (`../../SDK/html/class_vertica_1_1_transform_function.htm`) subclass that Vertica can call to process data.
- `getReturnType` tells 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*** (page 310).

```
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).getVarcharLength();
```

```
    // 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` (`../../SDK/html/class_vertica_1_1_column_types.htm`) entry in the 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 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 Vertica loads the shared library containing your UDTF. having your factory class instantiated is the only way that 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);
```

Allocating Resources

You have two options for allocating memory and file handles for your User Defined Functions (UDFs):

- Use Vertica SDK macros to allocate resources. This is the preferred method, since it uses Vertica's own resource manager, and guarantees that resources used by your UDF are reclaimed. See ***Allocating Resources with the SDK Macros*** (page 315).
- Allocate resources in your UDFs yourself. You must free these resources before your UDF exists.

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 impacts on Vertica's performance and even stability.

Whichever method you choose, you usually allocate resources in the `ScalarFunction.setup` or `TransformFunction.setup` function. These functions are called after your class is instantiated, but before Vertica calls it to process data.

If you allocate memory on your own, you usually free it in the `ScalarFunction.destroy` or `TransformFunction.destroy` functions. These functions are called after your UDF has performed all of its processing. These functions are also called if your UDF returns an error (see ***Handling Errors*** (page 315)).

Note: You should 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 Vertica's own memory pools. 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.

Allocating Resources with the SDK Macros

The 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 arbitrarilty-sized block of memory.

All of these macros allocate their memory from memory pools managed by Vertica. The main benefit of allowing 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.

You do not free any of the memory you allocate through any of these macros. The memory is automatically reclaimed by the Vertica process when the UDF has finished running. Attempting to free this memory will result in runtime errors.

Handling Errors

If your UDF encounters some sort of error, it can report back it back to 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 Vertica reports to the user. The error number is not defined by 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 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
```

Compiling Your UDF

g++ is the only supported compiler for compiling User Defined Function libraries (see ***Setting up a UDF Development Environment*** (page 300) for details). You should compile your UDF code on the same version of Linux that you use on your 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 Vertica. Supplying this file as C++ source rather than a library limits library compatibility issues.
- Add the 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 Vertica SDK directory is located at `/opt/vertica/sdk/doc`.

Note: Vertica 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 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 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. In this case, you will need to manually install this external library on each of your 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 dynamically-linked before adding the node. In additional, you need to ensure the same version of the library is installed on each node.

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 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 Vertica test environment to initially debug your UDF.

Write Messages to the Vertica Log

You can write status messages to the Vertica log 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. For example, you could add logging to find the input values the Add2ints example code receives like this:

```
const vint a = arg_reader.getIntRef(0);
const vint b = arg_reader.getIntRef(1);
srvInterface.log("Add2ints got a: %d and b: %d", (int) a, (int) b);
```

This code generates an entry in the Vertica log file. 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
```

Vertica does not automatically insert the name of your UDF, so you should include it to make finding your UDF's output easier. See Monitoring the Log Files in the Administrator's Guide for details on viewing the Vertica log files.

Deploying and Using UDSFs

To deploy a UDSF on your Vertica database:

1. Copy the UDF shared library file (`.so`) that contains your function to a node on your 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 Vertica. You pass this statement the location of the UDF library file you copied to the node earlier. Vertica distributes the library to each node in the cluster, and each Vertica process loads a copy of the library.
4. Use the CREATE FUNCTION statement to add the functions to the 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*** (page 321) 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 'Add2intsInfo' 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 and CREATE FUNCTION in the SQL Reference Manual.

Deploying and Using User Defined Transforms

To deploy a UDTF on your Vertica database:

1. Copy the UDF shared library file (`.so`) that contains your function to a node on your 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 Vertica. You pass this statement the location of the UDF library file you copied to the node earlier. Vertica distributes the library to each node in the cluster, and each Vertica process loads a copy of the library.
4. Use the CREATE TRANSFORM FUNCTION statement to add the function to the 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*** (page 321) 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 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)
=> 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)
```

Listing the UDFs Contained in a Library

Once a library has been loaded using the CREATE LIBRARY statement, you can find the UDFs 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 pass to the CREATE FUNCTION statement to access the UDF from SQL.

Using the Hadoop Connector

Hadoop is a software platform for performing distributed data processing. Vertica has created an interface between Vertica and Hadoop that lets you take advantage of both platform's strengths. With it, a Hadoop appplication can access and store data in a Vertica database.

Prerequisites

Before you can use Vertica's Hadoop Connector, you must install and configured Hadoop and be familiar with developing Hadoop applications. For details on installing and using Hadoop, please see the ***Apache Hadoop Web site http://hadoop.apache.org/***.

The Vertica Hadoop connector currently supports Hadoop version 20.2, and Pig version 0.7.

How Hadoop and Vertica Work Together

Hadoop and Vertica share some common features. They are both platforms that use clusters of commodity hardware to store and operate on very large sets of data. Their key difference is the type of data they operate on the best. Hadoop is best suited for tasks involving unstructured data, such as natural language or images. Vertica works best on structured data which can be loaded into database tables.

Vertica's Hadoop Connector lets you use these two platforms together to take advantage of their strengths. With it, you can use data from Vertica in a Hadoop application, and store the results of a Hadoop application into a Vertica database. For example, you can use Hadoop to extract and process keywords from a mass of unstructured text messages from a social media site, turning it into structured data that is then loaded into a Vertica database. Once you have loaded the data into Vertica, you can perform many different analytic queries on the data far faster than you would be able to using Hadoop alone.

For more on how Hadoop and Vertica work together, see the ***Vertica Map Reduce Web page http://www.vertica.com/mapreduce***.

Note: Vertica supports external procedures written any programming language supported on your database cluster (shell scripts, interpreted languages such a Python and Perl, and compiled languages such as C++). Similarly to Hadoop, you can use these external procedures to perform processing that SQL isn't well suited for or requires external data or resources. Depending on your application, they may be a better solution than using Hadoop, since they are easier to set up than a Hadoop cluster and require no further hardware investment.

However, external procedures do have a downside—since they run on the Vertica nodes, they can impact database performance if they require a significant amount of RAM, CPU cycles, or both. See the Programmer's Guide's ***Creating External Procedures*** (page 289) topic in the Vertica documentation for more information.

In general, if your application just needs to perform a quick operation on data that is hard to do in SQL, you might consider using external procedures rather than Hadoop. If your application needs to perform significant processing of data, Hadoop is usually a better solution.

Hadoop and Vertica Cluster Scaling

Nodes in the Hadoop cluster connect directly to Vertica nodes when retrieving or storing data, allowing large volumes of data to be transferred in parallel. If the Hadoop cluster is larger than the Vertica cluster, this parallel data transfer can negatively impact the performance of the Vertica database.

To avoid performance impacts on your Vertica database, you should ensure that your Hadoop cluster cannot overwhelm your Vertica cluster. The exact sizing of each cluster depends on how fast your Hadoop cluster generates data requests and the load placed on the Vertica database by queries from other sources. A good rule of thumb is to follow is your Hadoop cluster should be no larger than your Vertica cluster.

Hadoop Connector Features

Vertica's Hadoop Connector:

- gives Hadoop to access data stored in Vertica.
- lets Hadoop store its results in Vertica. The Connector will create a table for the data if it does not already exist.
- lets applications written in ***Apache Pig http://pig.apache.org/*** access and store data in Vertica.
- works with ***Hadoop streaming http://wiki.apache.org/hadoop/HadoopStreaming***.

The Hadoop Connector runs on each node in the Hadoop cluster, so the Hadoop nodes and Vertica nodes communicate with each other directly. Direct connections allow data to be transferred in parallel, dramatically increasing processing speed.

The Connector is written in Java, and is compatible with all platforms supported by Hadoop.

Hadoop Connector Installation Procedure

Follow these steps to install Vertica's Hadoop Connector:

If you have not already done so, download the Hadoop Connector package from Vertica's Hadoop Connector download page. You will also need a copy of the Vertica JDBC driver, which you can download from the ***Vertica Client Drivers Download page http://www.vertica.com/v-zone/downloads/client-tools/platform-overview***. On **each node** in your Hadoop cluster, you will need to perform the following steps:

1. Copy the Hadoop Connector `.zip` archive you downloaded to a temporary location.
2. Copy the Vertica JDBC driver .jar file to the same location on your node. If you haven't already, you can download this driver from the ***Vertica Download page http://www.vertica.com/v-zone/download_vertica***.
3. Unzip the Hadoop Connector `.zip` archive into a temporary directory. On Linux, you usually use the command `unzip`.
4. Locate the Hadoop home directory (the directory where Hadoop is installed). The location of this directory depends on how you installed Hadoop (manual install versus a package supplied by your Linux distribution or Cloudera). If you do not know the location of this directory, you can try the following steps to see if you can locate it:

- See if the HADOOP_HOME environment variable is set by issuing the command `echo $HADOOP_HOME` on the command line.
- See if Hadoop is in your path by typing `hadoop classpath` on the command line. If it is, this command lists the paths of all the jar files used by Hadoop, which should tell you the location of the Hadoop home directory.
- If you installed using a `.deb` or `.rpm` package, you can look in `/usr/lib/hadoop`, as this is often the location where these packages install Hadoop.

5 Copy the file `hadoop-vertica.jar` from the directory where you unzipped the Hadoop Connector archive to the `lib` subdirectory in the Hadoop home directory.

6 Copy the Vertica JDBC driver file (`vertica_x.x.x_jdk_5.jar`) to the `lib` subdirectory in the Hadoop home directory.

7 If you want your application written in Pig to be able to access Vertica, you need to:

1. Locate the Pig home directory. Often, this directory is in the same parent directory as the Hadoop home directory.
2. Copy the file named `pig-vertica.jar` from the directory where you unpacked the Hadoop Connector `.zip` file to the `lib` subdirectory in the Pig home directory.

Accessing Vertica Data from Hadoop

You need to follow three steps to have Hadoop fetch data from Vertica:

- Set the Hadoop application's input format to be `VerticaInputFormat`.
- Give the `VerticaInputFormat` class a query to be used to extract data from Vertica.
- Create a Mapper class that accepts `VerticaRecord` values as input.

The following sections explain each of these steps in greater detail.

Selecting VerticaInputFormat

The first step in reading data from Vertica from within a Hadoop application is to configure Vertica to be the input format by setting the Hadoop job's input format to be the VerticaInputFormat class.This is usually done within the `run()` method in your Hadoop application's class.

```
public int run(String[] args) throws Exception {
    // Set up the configuration and job objects
    Configuration conf = getConf();
    Job job = new Job(conf);

                        .    .    .

    // Set the input format to retrieve data from
    // Vertica.
    job.setInputFormatClass(VerticaInputFormat.class);

                        .    .    .
```

Setting the input to the `VerticaInputFormat` class means that the `map` method will get `VerticaRecord` objects as its input.

Setting the Query to Retrieve Data from Vertica

Your Hadoop application queries your Vertica database to extract its input data. You pass the query your Hadoop application should use to the `setInput` method of the `VerticaInputFormat` class. The Hadoop Connector sends this query to the Hadoop nodes which then individually connect to the Vertica and run the query to get their input data.

A primary consideration for this query is how it segments the data being retrieved from Vertica. Since each node in the Hadoop cluster needs data to process, the query result needs to be segmented between the nodes.

There are three ways to specify the query to retrieve data from Vertica, each of which impact how data is distributed across the Hadoop nodes:

- A simple, self-contained query.
- A parameterized query along with explicit parameters.
- A parameterized query along with a second query to retrieve the parameter values from Vertica.

The following sections explains each of these methods in detail.

Using a Simple Query to Extract Data from Vertica

The simplest way to specify the query Hadoop uses to extract data from Vertica is to pass a self-contained hard-coded query to the `setInput` method of the `VerticaInputFormat` class. You will usually make this call in the `run` method of your Hadoop application class. For example, the following code retrieves the entire contents of the table named allTypes.

```
// Sets the query to use to get the data from the Vertica database.
// Simple query with no parameters
VerticaInputFormat.setInput(job,
    "SELECT * FROM allTypes ORDER BY key;");
```

The query you supply must have an ORDER BY clause, since the Hadoop Connector uses this clause to figure out how to segment the query results between the Hadoop nodes. When it gets a simple query, the Hadoop Connector calculates limits and offsets to be sent to each node in the Hadoop cluster, so they each retrieve a portion of the query results to process.

Having Hadoop use a simple query to retrieve data from Vertica is the least efficient method, since the Hadoop Connector needs to perform extra processing to determine how the data should be segmented across the Hadoop nodes.

Using a Parameterized Query and Parameter Lists

You can have Hadoop retrieve data from Vertica using a parametrized query, to which you supply a set of parameters. The parameters in the query are represented by a question mark (`?`).

You pass the query and the parameters to the `setInput` method of the `VerticaInputFormat` class. You have two options for passing the parameters: using a discrete list, or by using a `Collection` object.

Using a Discrete List of Values

To pass a discrete list of parameters for the query, you include them in the `setInput` method call in a comma-separated list of string values, as shown in the next example:

```
// Simple query with supplied parameters
VerticaInputFormat.setInput(job,
    "SELECT * FROM allTypes WHERE key = ?", "0", "1", "2");
```

The Hadoop Connector sends this query to three nodes in the Hadoop cluster, with each of the nodes getting one of the parameter values. Each node then connects to a host in the Vertica database and executes the query, substituting in the parameter values it received.

This is a useful method when you have a discrete set of parameters that are unlikely to change over time. It is not very flexible, however, since any changes to the parameter list requires your application to be recompiled. Also, the query can contain just a single parameter, since you can only supply a single list to the `setInput` method. The more flexible way to use parameterized queries is to use a collection to contain the parameters.

Using a Collection Object

The more flexible method of supplying the parameters for the query is to store them into a `Collection` object, then include the object in the `setInput` method call. This method allows you to build the list of parameters at runtime, rather than having them hard-coded. You can also use multiple parameters in the query, since you will pass a collection of `ArrayList` objects to `setInput` statement. Each `ArrayList` object supplies one set of parameter values, and can contain values for each parameter in the query.

The following example demonstrates using a collection to pass the parameter values for a query containing two parameters. The collection object passed to `setInput` is an instance of the `HashSet` class. This object contains four `ArrayList` objects added within the `for` loop. This example just adds dummy values (the loop counter and the string `"FOUR"`). In your own application, you usually calculate parameter values in some manner before adding them to the collection.

Note: If your parameter values are stored in Vertica, you should specify the parameters using a query instead of a collection. See ***Using a Query to Retrieve Parameters for a Parameterized Query*** (page 327) for details.

```
// Collection to hold all of the sets of parameters for the query.
Collection<List<Object>> params = new HashSet<List<Object>>() {
};

// Each set of parameters lives in an ArrayList. Each entry
// in the list supplies a value for a single parameter in
// the query. Here, ArrayList objects are created in a loop
// that adds the loop counter and a static string as the
// parameters. The ArrayList is then added to the collection.
for (int i = 0; i < 4; i++) {
    ArrayList<Object> param = new ArrayList<Object>();
    param.add(i);
    param.add("FOUR");
    params.add(param);
```

```
}

VerticaInputFormat.setInput(job,
        "select * from allTypes where key = ? AND NOT varcharcol = ?",
        params);
```

Scaling Parameter Lists for the Hadoop Cluster

Whenever possible, you should make the number of parameter values you pass to the Hadoop Connector equal to the number of nodes in the Hadoop cluster, since each parameter value is assigned to a single Hadoop node. This ensures that the workload is spread across the entire Hadoop cluster. If you supply fewer parameter values than there are nodes in the Hadoop cluster, some of the nodes will not get a value and will sit idle. If there are more parameter values than there are nodes in the cluster, Hadoop will randomly assign the extra values to nodes in the cluster. (Hadoop does not perform scheduling, and therefore will not wait to see what nodes finish their task first before assigning additional tasks.) This means a node could become a bottleneck if it is assigned the most processing-intensive portions of the job.

In addition to the number of parameters in the query, you should also try to make the parameter values yield roughly the same number of results. This helps prevent a single node in the Hadoop cluster from becoming a bottleneck by having to process far more data than the other nodes in the cluster.

Using a Query to Retrieve Parameter Values for a Parameterized Query

You can pass the Hadoop Connector a query to extract the parameter values for a parameterized query. This query must return a single column of data, which is then used as values for running the parameterized query.

To use a query to retrieve the parameter values, you supply the `VerticaInputFormat` class's `setInput` method with the parameterized query and the query to retrieve parameters. For example:

```
// Sets the query to use to get the data from the Vertica database.
// Query using a parameter that is supplied by another query
VerticaInputFormat.setInput(job,
        "select * from allTypes where key = ?",
        "select distinct key from regions");
```

When it receives a query for parameters, the Hadoop Connector runs the query itself, then groups the results together to send out to the Hadoop nodes, along with the parameterized query. The Hadoop nodes then run the parameterized query using the set of parameter values sent to them by the Hadoop Connector.

Writing a Map Class that Processes Vertica Data

Once you have set up your Hadoop application to read data from Vertica, you need to create your `Map` class that will actually process the data. Your `Map` class's `map` method receives `LongWritable` values as keys and `VerticaRecord` objects as values. The key values are just sequential numbers that identify the row in the query results. The `VerticaRecord` class represents a single row from the result set returned by the query you supplied to the `VerticaInput.setInput` method.

Working with the VerticaRecord Class

Your `map` method extracts the data it needs from the `VerticaRecord` class. This class contains three main methods you use to extract data from the record set:

- `get` retrieves a single value, either by index value or by name, from the row sent to the map method.
- `getValues` returns all of the values in the row as a `List` object. This lets you easily iterate over the entire content of the rows.
- `getTypes` returns a `List` object containing the data types of the columns in the row. This can be useful if you are unsure the data types of the columns returned by the query. The types are stored as integer values defined by the `java.sql.Types` class.

The following example shows a `Mapper` class and `map` method that accepts `VerticaRecord` objects. In this example, no real work is done. Instead two values are selected as the key and value to be passed on the to reducer.

```
public static class Map extends
        Mapper<LongWritable, VerticaRecord, Text, DoubleWritable> {
    // This mapper accepts VerticaRecords as input.
    public void map(LongWritable key, VerticaRecord value, Context context)
            throws IOException, InterruptedException {
        // Gets a record, which contains a row returned by the
        // query passed to the VerticaInput class.
        // This example gets the entire row's values as a List
        // object. You can also get individual values using the get method.
        List<Object> record = value.getValues();
        // In your mapper, you would do actual processing here.
        // This simple example just passes two values to the
        // reducer as the key and value.
        if (record.get(3) != null && record.get(0) != null) {
            context.write(new Text((String) record.get(3)),
               new DoubleWritable((Long) record.get(0)));
        }
    }
}
```

If your Hadoop application has a reduce stage, all of the output of your `map` method is managed by Hadoop. It is not stored or manipulated in any way by Vertica. If your Hadoop application does not have a reduce stage, and it will be storing its output into Vertica, your map method will need to output its keys as `Text` objects and values as `VerticaRecord` objects.

Writing Data to Vertica from Hadoop

There are three steps you need to take for your Hadoop application to store data in Vertica:

- Set the output value class of your Hadoop application to `VerticaRecord`.
- Set the details of the Vertica table where you want to store your data in the `VerticaOutputFormat` class.
- Create a `Reduce` class that adds data to a `VerticaRecord` object and calls its write method to store the data.

The following sections explain these steps in more detail.

Configuring Hadoop to Output to Vertica

To tell your Hadoop application to output data to Vertica you configure your Hadoop application to output to the Vertica Hadoop Connector. You will normally perform these steps in your Hadoop application's `run` method. There are three methods that need to be called in order to set up the output to be sent to the Hadoop Connector and to set the output of the `Reduce` class, as shown in the following example:

```
// Set the output format of Reduce class. It will
// output VerticaRecords that will be stored in the
// database.
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(VerticaRecord.class);

// Tell Hadoop to send its output to the Vertica
// Hadoop Connector.
job.setOutputFormatClass(VerticaOutputFormat.class);
```

The call to `setOutputValueClass` tells Hadoop that the output of the `Reduce.reduce` method is a `VerticaRecord` class object. This object represents a single row of a Vertica database table. Your Hadoop application knows to send the data to the Hadoop Connector by setting the output format class to `VerticaOutputFormat`.

Defining the Output Table

Your Hadoop application needs to define the table that will hold its output. You define the table by calling the `VerticaOutputFormat.setOutput` method:

```
VerticaOutputFormat.setOutput(jobObject, tableName, truncate, "columnName1 dataType1", ...,
"columnNameN dataTypeN");
```

jobObject	The Hadoop job object for your application.
tableName	The name of the table to store Hadoop's output. If this table does not exist, the Hadoop Connector will automatically create it.
truncate	A Boolean value controlling whether tableName should be truncated if it already exists. If set to `true`, any existing data in the table is deleted before Hadoop's output is stored. When set to `false`, the Hadoop output is added to the existing data in the table.
"*columnName1 dataType1*"	The definition of a column in the table. *columnName* is the name of the column and *dataType* is the SQL data type to store in the column. These two values are separated by a space.

The first three parameters are required. You add as many column definitions as you need in your output table.

You usually call the `setOutput` method in your Hadoop class's `run` method, where all other setup work is done. The following example sets up an output table named mrtarget that contains 8 columns, each containing a different data type:

```
// Sets the output format for storing data in Vertica. It defines the
// table where data is stored, and the columns it will be stored in.
VerticaOutputFormat.setOutput(job, "mrtarget", true, "a int",
     "b boolean", "c char(1)", "d date", "f float", "t timestamp",
     "v varchar", "z varbinary");
```

If the mrtarget table already exists in the Vertica database, the Hadoop Connector deletes its contents before the Hadoop application's output is stored, since the method call's truncate parameter is set to `true`.

Note: If the table already exists in the database, the Hadoop Connector does not verify that the column defined in the existing table match those defined in the `setOutput` method call. This discrepancy can cause your Hadoop application to throw casting exceptions if the values being stored in the column cannot be converted to the existing column definition.

Writing the Reduce Class

Once your Hadoop application is configured to output data to Vertica and has its output table defined, you need to create the `Reduce` class that actually formats and writes the data for storage in Vertica.

The first step your `Reduce` class should take is to instantiate a `VerticaRecord` object to hold the output of the `reduce` method. This is a little more complex than just instantiating a base object, since the `VerticaRecord` object must have the columns defined in it that match the out table's columns (see ***Defining the Output Table*** (page 329) for details).

To get the properly configured `VerticaRecord` object, call the `VerticaOutputFormat.getValue` method which takes the configuration context as input. You usually make this method call in your `Reduce` class's `setup` method, which Hadoop calls before it calls the `reduce` method. For example:

```
// Sets up the output record that will be populated by
// the reducer and eventually written out.
public void setup(Context context) throws IOException,
        InterruptedException {
    super.setup(context);
    try {
        // Need to call VerticaOutputFormat to get a record object
        // that has the proper column definitions. The object is
        // stored in the record field for use later.
        record = VerticaOutputFormat.getValue(context
                .getConfiguration());
     } catch (Exception e) {
          throw new IOException(e);
     }
}
```

Storing Data in the VerticaRecord

Your `reduce` method starts the same way any other Hadoop reduce method does—it processes its input the key and values, performing whatever reduction task your application needs. Afterwards, your `reduce` method adds the data to be stored in Vertica to the `VerticaRecord` object that was instantiated earlier. Usually you use the `set` method to add the data:

```
VerticaRecord.set(column, value [,validate]);
```

column	The column to store the value in. This is either an integer (the column number) or a `String` (the column name, as defined in the table definition). **Note:** The set method throws an exception if you pass it the name of a column that does not exist. You should always use a try/catch block around any set method call that uses a column name.
value	The value to store in the column. The data type of this value must match the definition of the column in the table.
validate	A Boolean that controls whether the value's data type is compared to the data type for the column. When omitted or set to `false`, the value is not checked by the `set` method. When set to `true`, the `set` method verifies that the data type of the value matches the column. If they do not match, it throws a `ClassCastException`.

Note: If you do not have the `set` method validate that the data types of the value and the column match, the Hadoop Connector throws a `ClassCastException` if it finds a mismatch when it tries to commit the data to the database. This exception causes a rollback of the entire result. By having the `set` method validate the data type of the value, you can catch and resolve the exception before it causes a rollback.

In addition to the `set` method, you can also use the `setFromString` method to have the Hadoop Connector convert the value from `String` to the proper data type for the column:

```
VerticaRecord.setFromString(column, "value");
```

column	The column number to store the value in, as an integer.
value	A `String` containing the value to store in the column. If the `String` cannot be converted to the correct data type to be stored in the column, `setFromString` throws an exception (`ParseException` for date values, `NumberFormatException` numeric values).

After it populates the `VerticaRecord` object, your `reduce` method calls the `Context.write` method, passing it the name of the table to store the data in as the key, and the `VerticaRecord` object as the value.

The following example shows a simple `Reduce` class that stores data into Vertica. To make the example as simple as possible, the code doesn't actually process the input it receives, and instead just writes dummy data to the database. In your own application, you would process the key and values into data that you then store in the `VerticaRecord` object.

```
public static class Reduce extends
        Reducer<Text, DoubleWritable, Text, VerticaRecord> {
    // Holds the records that the reducer writes its values to.
    VerticaRecord record = null;
    // Sets up the output record that will be populated by
    // the reducer and eventually written out.
    public void setup(Context context) throws IOException,
            InterruptedException {
        super.setup(context);
        try {
            // Need to call VerticaOutputFormat to get a record object
            // that has the proper column definitions.
            record = VerticaOutputFormat.getValue(context
                    .getConfiguration());
        } catch (Exception e) {
            throw new IOException(e);
        }
    }

    // The reduce method.
    public void reduce(Text key, Iterable<DoubleWritable> values,
            Context context) throws IOException, InterruptedException {
        // Ensure that the record object has been set up properly. This is
        // where the results are written.
        if (record == null) {
            throw new IOException("No output record found");
        }

        // In this part of your application, your reducer would process the
        // key and values parameters to arrive at values that you want to
        // store into the database. For simplicity's sake, this example
        // skips all of the processing, and just inserts arbitrary values
        // into the database.
        //
        // Use the .set method to store a value in the record to be stored
        // in the database. The first parameter is the column number,
        // the second is the value to store, and the third is a boolean
        // indicating whether the value's type should be validated before
        // committing it.
        //
        // Set record 0 to an integer value, and verify that column 0 is
        // an integer. If you use the optional validate parameter, you should
        // always use a try/catch block to catch the exception.
        try {
            record.set(0, 125, true);
        } catch (ClassCastException e) {
            // Handle the improper data type here.
            e.printStackTrace();
        }
```

```
            // You can also set column values by name rather than by column
            // number. However, this requires a try/catch since specifying a
            // non-existent column name will throw an exception.
            try {
                // The second column, named "b",1 contains a Boolean value.
                record.set("b", true);
            } catch (Exception e) {
                // Handle an improper column name here.
                e.printStackTrace();
            }

            // Column 2 stores a single char value.
            record.set(2, 'c', true);
            // Column 3 is a date.
            record.set(3, Calendar.getInstance().getTime(), true);

            // You can use the setFromString method to convert a string
            // value into the proper data type to be stored in the column.
            // You need to use a try...catch block in this case, since the
            // string to value conversion could fail (for example, trying to
            // store "Hello, World!" in a float column is not going to work).
            try {
                record.setFromString(4, "234.567");
            } catch (ParseException e) {
                // Thrown if the string cannot be parsed into the data type
                // to be stored in the column.
                e.printStackTrace();
            }

            // Column 5 stores a timestamp
            record.set(5, Calendar.getInstance().getTime(), true);
            // Column 6 stores a varchar
            record.set(6, "example string", true);
            // Column 7 stores a varbinary
            record.set(7, new byte[10], true);

            // Once the columns are populated, write the record to store
            // the row into the database.
            context.write(new Text("mrtarget"), record);
        }
    }
```

Passing Parameters to the Hadoop Connector at Runtime

You need to pass connection parameters to the Hadoop Connector when starting your Hadoop application. At a minimum, these parameters must include the list of hostnames in the Vertica database cluster, the name of the database, and the user name. The common parameters for accessing the database appear in the following table. Usually, you will only need the basic parameters listed in this table in order to start your Hadoop application.

Parameter	Description	Required	Default
`mapred.vertica.hostnames`	A comma-separated list of the names or IP addresses of the hosts in the Vertica cluster. You should list all of the nodes in the cluster here, since individual nodes in the Hadoop cluster connect directly with a randomly assigned host in the cluster. The hosts in this cluster are used for both reading from and writing data to the Vertica database, unless you specify a different output database (see below).	Yes	none
`mapred.vertica.port`	The port number for the Vertica database.	No	5433
`mapred.vertica.database`	The name of the database the Hadoop application should access.	Yes	
`mapred.vertica.username`	The username to use when connecting to the database.	Yes	
`mapred.vertica.password`	The password to use when connecting to the database.	No	empty

You pass the parameters to the Hadoop Connector using the `-D` command line switch in the command you use to start your Hadoop application. For example:

```
hadoop jar myHadoopApp.jar com.myorg.hadoop.myHadoopApp \
    -Dmapred.vertica.hostnames=Vertica01,Vertica02,Vertica03,Vertica04 \
    -Dmapred.vertica.port=5433 -Dmapred.vertica.username=exampleuser \
    -Dmapred.vertica.password=password123 -Dmapred.vertica.database=ExampleDB
```

Parameters for a Separate Output Database

The parameters in the previous table are all you need if your Hadoop application accesses a single Vertica database. You can also have your Hadoop application read from one Vertica database and write to a different Vertica database. In this case, the parameters shown in the previous table apply to the input database (the one Hadoop reads data from). The following table lists the parameters that you use to supply your Hadoop application with the connection information for the output database (the one it writes its data to). None of these parameters is required. If you do not assign a value to one of these output parameters, it inherits its value from the input database parameters.

Parameter	Description	Default
mapred.vertica.hostnames.output	A comma-separated list of the names or IP addresses of the hosts in the output Vertica cluster.	Input hostnames
mapred.vertica.port.output	The port number for the output Vertica database.	5433
mapred.vertica.database.output	The name of the output database.	Input database name
mapred.vertica.username.output	The username to use when connecting to the output database.	Input database user name
mapred.vertica.password.output	The password to use when connecting to the output database.	Input database password

Example Hadoop Connector Application

This section presents an example of using the Hadoop Connector to retrieve and store data from a Vertica database. The example pulls together the code that has appeared on the previous sections of this guide to present a functioning example.

This application reads data from a table named allTypes. The mapper selects two values from this table to send to the reducer. The reducer doesn't perform any operations on the input, and instead inserts arbitrary data into the output table named mrtarget.

```
package com.vertica.hadoop;
import java.io.IOException;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

```
import com.vertica.hadoop.VerticaInputFormat;
import com.vertica.hadoop.VerticaOutputFormat;
import com.vertica.hadoop.VerticaRecord;

// This is the class that contains the entire Hadoop example.
public class VerticaExample extends Configured implements Tool {
    public static class Map extends
            Mapper<LongWritable, VerticaRecord, Text, DoubleWritable> {
        // This mapper accepts VerticaRecords as input.
        public void map(LongWritable key, VerticaRecord value, Context context)
                throws IOException, InterruptedException {
            // Gets a record, which contains a row returned by the
            // query passed to the VerticaInput class.
            // This example gets the entire row's values as a List
            // object. You can also get individual values using the get method.
            List<Object> record = value.getValues();
            // In your mapper, you would do actual processing here.
            // This simple example just passes two values to the
            // reducer as the key and value.
            if (record.get(3) != null && record.get(0) != null) {
                context.write(new Text((String) record.get(3)),
                   new DoubleWritable((Long) record.get(0)));
            }
        }
    }

    public static class Reduce extends
            Reducer<Text, DoubleWritable, Text, VerticaRecord> {
        // Holds the records that the reducer writes its values to.
        VerticaRecord record = null;
        // Sets up the output record that will be populated by
        // the reducer and eventually written out.
        public void setup(Context context) throws IOException,
                InterruptedException {
            super.setup(context);
            try {
                // Need to call VerticaOutputFormat to get a record object
                // that has the proper column definitions.
                record = VerticaOutputFormat.getValue(context
                        .getConfiguration());
            } catch (Exception e) {
                throw new IOException(e);
            }
        }

        // The reduce method.
        public void reduce(Text key, Iterable<DoubleWritable> values,
                Context context) throws IOException, InterruptedException {
            // Ensure that the record object has been set up properly. This is
            // where the results are written.
            if (record == null) {
                throw new IOException("No output record found");
            }
```

```
// In this part of your application, your reducer would process the
// key and values parameters to arrive at values that you want to
// store into the database. For simplicity's sake, this example
// skips all of the processing, and just inserts arbitrary values
// into the database.
//
// Use the .set method to store a value in the record to be stored
// in the database. The first parameter is the column number,
// the second is the value to store, and the third is a boolean
// indicating whether the value's type should be validated before
// committing it.
//
// Set record 0 to an integer value, and verify that column 0 is
// an integer. If you use the optional validate parameter, you should
// always use a try/catch block to catch the exception.
try {
    record.set(0, 125, true);
} catch (ClassCastException e) {
    // Handle the improper data type here.
    e.printStackTrace();
}


// You can also set column values by name rather than by column
// number. However, this requires a try/catch since specifying a
// non-existent column name will throw an exception.
try {
    // The second column, named "b",l contains a Boolean value.
    record.set("b", true);
} catch (Exception e) {
    // Handle an improper column name here.
    e.printStackTrace();
}

// Column 2 stores a single char value.
record.set(2, 'c', true);
// Column 3 is a date.
record.set(3, Calendar.getInstance().getTime(), true);

// You can use the setFromString method to convert a string
// value into the proper data type to be stored in the column.
// You need to use a try...catch block in this case, since the
// string to value conversion could fail (for example, trying to
// store "Hello, World!" in a float column is not going to work).
try {
    record.setFromString(4, "234.567");
} catch (ParseException e) {
    // Thrown if the string cannot be parsed into the data type
    // to be stored in the column.
    e.printStackTrace();
}

// Column 5 stores a timestamp
record.set(5, Calendar.getInstance().getTime(), true);
// Column 6 stores a varchar
```

```
            record.set(6, "example string", true);
            // Column 7 stores a varbinary
            record.set(7, new byte[10], true);

            // Once the columns are populated, write the record to store
            // the row into the database.
            context.write(new Text("mrtarget"), record);
        }
    }

    @Override
    public int run(String[] args) throws Exception {
        // Set up the configuration and job objects
        Configuration conf = getConf();
        Job job = new Job(conf);
        conf = job.getConfiguration();
        conf.set("mapreduce.job.tracker", "local");
        job.setJobName("vertica test");
        // Set the input format to retrieve data from
        // Vertica.

        job.setInputFormatClass(VerticaInputFormat.class);
        // Set the output format of the mapper. This is the interim
        // data format passed to the reducer. Here, we will pass in a
        // Double. The interim data is not processed by Vertica in any
        // way.
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(DoubleWritable.class);
        // Set the output format of the Hadoop application. It will
        // output VerticaRecords that will be stored in the
        // database.
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(VerticaRecord.class);
        job.setOutputFormatClass(VerticaOutputFormat.class);
        job.setJarByClass(VerticaExample.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);

        // Sets the output format for storing data in Vertica. It defines the
        // table where data is stored, and the columns it will be stored in.
        VerticaOutputFormat.setOutput(job, "mrtarget", true, "a int",
                "b boolean", "c char(1)", "d date", "f float", "t timestamp",
                "v varchar", "z varbinary");
        // Sets the query to use to get the data from the Vertica database.
        // Query using a list of parameters.
        VerticaInputFormat.setInput(job,
                "select * from allTypes where key = ?",
                "0","1","2");
        job.waitForCompletion(true); return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new VerticaExample(),
                args);
```

```
        System.exit(res);
    }
}
```

Compiling and Running the Example Application

To run the example Hadoop application, you first need to set up the table that the example reads as input: allTypes. To set up the input table, follow these steps:

1. Locate the `hadoop-vertica-example.jar` file that was included in Vertica's Hadoop Connector download .zip archive.
2. Extract the file named `datasource` from the the `hadoop-vertica-example.jar` file:
   ```
   jar xf hadoop-vertica-example.jar datasource
   ```
3. Copy `datasource` to your Vertica database's Administration Host.
4. Connect to the host where you copied the `datasource` file.
5. Using vsql, connect to the Vertica database that you want the example Hadoop application to access using the Database Administrator.
6. Run the following query to set up the table:
   ```
   CREATE TABLE allTypes (key identity,intcol integer,
                  floatcol float,
                 charcol char(10),
                 varcharcol varchar,
                 boolcol boolean,
                 datecol date,
                 timestampcol timestamp,
                 timestampTzcol timestamptz,
                 timecol time,
                 timeTzcol timetz,
                 varbincol varbinary,
                 bincol binary,
                 numcol numeric(38,0),
                 intervalcol interval
           );
   ```
7. Run the following query to load data from the `datasource` file into the table:
   ```
   COPY allTypes COLUMN OPTION (varbincol FORMAT 'hex', bincol FORMAT 'hex')
   FROM path-to-datasource/datasource DIRECT;
   ```
   Replace *`path-to-datasource`* with the absolute path to the `datasource` file you copied to the host earlier.

Compiling the Example (optional)

The example code presented in this section is based on example code distributed along with Vertica's Hadoop Connector in the file `hadoop-vertica-example.jar`. If you just want to run the example, skip to the next section and use the `hadoop-vertica-example.jar` file that came as part of the Hadoop Connector package rather than a version you compiled yourself.

To compile the example code listed in ***Example Hadoop Connector Application*** (page 335), follow these steps:

1. Log into a node on your Hadoop cluster.
2. Locate the Hadoop home directory. See ***Hadoop Connector Installation Procedure*** (page 323) for tips on how to find this directory.
3. If it is not already set, set the environment variable `HADOOP_HOME` to the Hadoop home directory:

   ```
   export HADOOP_HOME=path_to_Hadoop_home
   ```

   If you installed Hadoop using an `.rpm` or `.deb` package, Hadoop is usually installed in `/usr/lib/hadoop`:

   ```
   export HADOOP_HOME=/usr/lib/hadoop
   ```
4. Save the example source code to a file named `VerticaExample.java` on your Hadoop node.
5. In the same directory where you saved `VerticaExample.java`, create a directory named classes. On Linux, the command is:

   ```
   mkdir classes
   ```
6. Compile the Hadoop example:

   ```
   javac -classpath \
   $HADOOP_HOME/hadoop-core.jar:$HADOOP_HOME/lib/hadoop-vertica.jar \
   -d classes VerticaExample.java \
   && jar -cvf hadoop-vertica-example.jar -C classes .
   ```

When the compilation finishes, you will have a file named `hadoop-vertica-example.jar` in the same directory as the `VerticaExample.java` file. This is the file you will have Hadoop run.

Running the Example Application

Once you have compiled the example, you run it using the following command line:

```
hadoop jar hadoop-vertica-example.jar \
  com.vertica.hadoop.VerticaExample \
  -Dmapred.vertica.hostnames=VerticaHost01,VerticaHost02,... \
  -Dmapred.vertica.port=portNumber \
  -Dmapred.vertica.username=userName \
  -Dmapred.vertica.password=dbPassword \
  -Dmapred.vertica.database=databaseName
```

This command tells Hadoop to run your application's `.jar` file, and supplies the parameters needed for your application to connect to your Vertica database. Fill in your own values for the hostnames, port, user name, password, and database name for your Vertica database.

After entering the command line, you will see output from Hadoop as it processes data that looks similar to the following:

```
11/03/01 15:16:55 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker,
sessionId=
11/03/01 15:16:56 INFO mapred.JobClient: Running job: job_local_0001
select * from allTypes where key = 1
11/03/01 15:16:56 INFO mapred.MapTask: io.sort.mb = 100
11/03/01 15:16:57 INFO mapred.MapTask: data buffer = 79691776/99614720
11/03/01 15:16:57 INFO mapred.MapTask: record buffer = 262144/327680
```

```
11/03/01 15:16:57 INFO mapred.MapTask: Starting flush of map output
11/03/01 15:16:57 INFO mapred.MapTask: Finished spill 0
11/03/01 15:16:57 INFO mapred.Task: Task:attempt_local_0001_m_000000_0 is done. And is in the process
of commiting
11/03/01 15:16:57 INFO mapred.LocalJobRunner:
11/03/01 15:16:57 INFO mapred.Task: Task 'attempt_local_0001_m_000000_0' done.
select * from allTypes where key = 2
11/03/01 15:16:57 INFO mapred.MapTask: io.sort.mb = 100
11/03/01 15:16:57 INFO mapred.JobClient:  map 100% reduce 0%
11/03/01 15:16:57 INFO mapred.MapTask: data buffer = 79691776/99614720
11/03/01 15:16:57 INFO mapred.MapTask: record buffer = 262144/327680
11/03/01 15:16:57 INFO mapred.MapTask: Starting flush of map output
11/03/01 15:16:57 INFO mapred.MapTask: Finished spill 0
11/03/01 15:16:57 INFO mapred.Task: Task:attempt_local_0001_m_000001_0 is done. And is in the process
of commiting
11/03/01 15:16:57 INFO mapred.LocalJobRunner:
11/03/01 15:16:57 INFO mapred.Task: Task 'attempt_local_0001_m_000001_0' done.
select * from allTypes where key = 0
11/03/01 15:16:57 INFO mapred.MapTask: io.sort.mb = 100
11/03/01 15:16:57 INFO mapred.MapTask: data buffer = 79691776/99614720
11/03/01 15:16:57 INFO mapred.MapTask: record buffer = 262144/327680
11/03/01 15:16:57 INFO mapred.MapTask: Starting flush of map output
11/03/01 15:16:57 INFO mapred.Task: Task:attempt_local_0001_m_000002_0 is done. And is in the process
of commiting
11/03/01 15:16:57 INFO mapred.LocalJobRunner:
11/03/01 15:16:57 INFO mapred.Task: Task 'attempt_local_0001_m_000002_0' done.
11/03/01 15:16:57 INFO mapred.LocalJobRunner:
11/03/01 15:16:57 INFO mapred.Merger: Merging 3 sorted segments
11/03/01 15:16:58 INFO mapred.Merger: Down to the last merge-pass, with 2 segments left of total size:
46 bytes
11/03/01 15:16:58 INFO mapred.LocalJobRunner:
11/03/01 15:16:58 INFO mapred.Task: Task:attempt_local_0001_r_000000_0 is done. And is in the process
of commiting
11/03/01 15:16:58 INFO mapred.LocalJobRunner: reduce > reduce
11/03/01 15:16:58 INFO mapred.Task: Task 'attempt_local_0001_r_000000_0' done.
11/03/01 15:16:58 WARN mapred.FileOutputCommitter: Output path is null in cleanup
11/03/01 15:16:58 INFO mapred.JobClient:  map 100% reduce 100%
11/03/01 15:16:58 INFO mapred.JobClient: Job complete: job_local_0001
11/03/01 15:16:58 INFO mapred.JobClient: Counters: 13
11/03/01 15:16:58 INFO mapred.JobClient:   FileSystemCounters
11/03/01 15:16:58 INFO mapred.JobClient:     FILE_BYTES_READ=23833
11/03/01 15:16:58 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=205737
11/03/01 15:16:58 INFO mapred.JobClient:   Map-Reduce Framework
11/03/01 15:16:58 INFO mapred.JobClient:     Reduce input groups=2
11/03/01 15:16:58 INFO mapred.JobClient:     Combine output records=0
11/03/01 15:16:58 INFO mapred.JobClient:     Map input records=2
11/03/01 15:16:58 INFO mapred.JobClient:     Reduce shuffle bytes=0
11/03/01 15:16:58 INFO mapred.JobClient:     Reduce output records=2
11/03/01 15:16:58 INFO mapred.JobClient:     Spilled Records=4
11/03/01 15:16:58 INFO mapred.JobClient:     Map output bytes=38
11/03/01 15:16:58 INFO mapred.JobClient:     Combine input records=0
11/03/01 15:16:58 INFO mapred.JobClient:     Map output records=2
11/03/01 15:16:58 INFO mapred.JobClient:     SPLIT_RAW_BYTES=330
11/03/01 15:16:58 INFO mapred.JobClient:     Reduce input records=2
```

Note: The version of the example supplied in the Hadoop Connector download package will produce more output, since it runs several input queries.

Verifying the Results

Once your Hadoop application finishes, you can verify it ran correctly by looking at the mrtarget table in your Vertica database:

Connect to your Vertica database using vsql and run the following query:

```
=> SELECT * FROM mrtarget;
```

The results should look like this:

```
 a | b | c |     d      |    f    |          t          |       v        | z
---+---+---+------------+---------+---------------------+----------------+---
 3 | t | c | 2011-03-01 | 234.567 | 2011-03-01 15:12:12 | example string |
 3 | t | c | 2011-03-01 | 234.567 | 2011-03-01 15:12:12 | example string |
(2 rows)
```

Using Hadoop Streaming with the Vertica's Hadoop Connector

Hadoop streaming allows you to create an ad-hoc Hadoop job that uses standard commands (such as UNIX command-line utilities) for its map and reduce processing. See the Hadoop wiki's ***topic on streaming http://wiki.apache.org/hadoop/HadoopStreaming*** for more information. You can have a streaming job retrieve data from a Vertica database, store data into a Vertica database, or both.

Reading Data from Vertica in a Streaming Hadoop Job

To have a streaming Hadoop job read data from a Vertica database, you set the `inputformat` argument of the Hadoop command line to `com.vertica.deprecated.VerticaStreamingInput`. You also need to supply parameters that tell the Hadoop job how to connect to your Vertica database. See ***Passing Parameters to the Hadoop Connector at Runtime*** (page 334) for an explanation of these command-line parameters.

Note: The `VerticaStreamingInput` class is within the deprecated namespace because the current version of Hadoop (0.20.1) has not defined a current API for streaming. Instead, the streaming classes conform to the Hadoop version 0.18 API.

In addition to the standard command-line parameters that tell Hadoop how to access your database, there are additional streaming-specific parameters you need to use that supply Hadoop with the query it should use to extract data from Vertica and other query-related options.

Parameter	Description	Required	Default
`mapred.vertica.input.query`	The query to use to retrieve data from the Vertica database. See ***Setting the Query to Retrieve Data from Vertica*** (page 325) for more information.	Yes	none
`mapred.vertica.input.paramquery`	A query to execute to retrieve parameters for the query given in the `.input.query` parameter.	If query has parameter and no discrete parameters supplied	
`mapred.vertica.query.params`	Discrete list of parameters for the query.	If query has parameter and no parameter query	

Parameter	Description	Required	Default
		supplied	
mapred.vertica.input.delimiter	The character to use for separating column values. The command you use as a mapper needs to split individual column values apart using this delimiter.	No	0xa
mapred.vertica.input.terminator	The character used to signal the end of a row of data from the query result.	No	0xb

The following command demonstrates reading data from a table named allTypes. This command uses the UNIX cat command as a mapper and reducer so it will just pass the contents through.

```
hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-streaming-0.20.2+737.jar \
  -Dmapred.vertica.hostnames=VerticaHost01,VerticaHost02,... \
  -Dmapred.vertica.database=ExampleDB \
  -Dmapred.vertica.username=ExampleUser \
  -Dmapred.vertica.password=password123 \
  -Dmapred.vertica.input.query="select * from allTypes order by key" \
  -Dmapred.vertica.input.delimiter=, \
  -inputformat com.vertica.hadoop.deprecated.VerticaStreamingInput \
  -input /tmp/input -output /tmp/output -reducer /bin/cat -mapper /bin/cat
```

Note: Even though the input is coming from Vertica, you need to supply the `-input` parameter to Hadoop for it to process the streaming job.

Writing Data to Vertica in a Streaming Hadoop Job

Similar to reading from a streaming Hadoop job, you write data to Vertica by setting the `outputformat` parameter of your Hadoop command to `com.vertica.deprecated.VerticaStreamingOutput`. As with reading from Vertica, you need to supply parameters that tell the streaming Hadoop job how to connect to the database. See ***Passing Parameters to the Hadoop Connector at Runtime*** (page 334) for an explanation of these command-line parameters. If you are reading data from one Vertica database and writing to another, you need to use the output parameters, similarly if you were reading and writing to separate databases using a Hadoop application. There are also additional parameters that configure the output of the streaming Hadoop job, listed in the following table.

Parameter	Description	Required	Default
mapred.vertica.output.table.name	The name of the table where Hadoop should store its data.	Yes	none
mapred.vertica.output.table.def	The definition of the table. The format is the same as used for defining the output table for a Hadoop application. See ***Defining the Output Table*** (page 329) for details.	If the table does not already exist in the database	

Parameter	Description	Required	Default
mapred.vertica.output.table.drop	Whether to truncate the table before adding data to it.	No	false
mapred.vertica.output.delimiter	The character to use for separating column values.	No	0x7 (ASCII bell character)
mapred.vertica.output.terminator	The character used to signal the end of a row of data..	No	0x8 (ASCII backspace)

The following example demonstrates reading data from a Vertica database table named allTypes and writing it back to the same database in a table named hadoopout.

```
hadoop jar contrib/streaming/hadoop-streaming-0.20.2+737.jar
 -Dmapred.vertica.output.table.name=hadoopout \
 -Dmapred.vertica.hostnames=VerticaHost01,VerticaHost02,VerticaHost03 \
 -Dmapred.vertica.database=ExampleDB \
 -Dmapred.vertica.username=ExampleUser \
 -Dmapred.vertica.password=password123 \
 -Dmapred.vertica.input.query="select * from allTypes order by key" \
 -Dmapred.vertica.input.delimiter=, \
 -Dmapred.vertica.output.delimiter=, \
 -Dmapred.vertica.input.terminator=0x0a \
 -Dmapred.vertica.output.terminator=0x0a \
 -inputformat com.vertica.hadoop.deprecated.VerticaStreamingInput \
 -outputformat com.vertica.hadoop.deprecated.VerticaStreamingOutput \
 -input /tmp/input \
 -output /tmp/output \
 -reducer /bin/cat \
 -mapper /bin/cat
```

Accessing Vertica from Pig

Vertica's Hadoop Connector includes a Java package that lets you access a Vertica database. This .jar file must be places somewhere in your Pig installation's CLASSPATH (see ***Hadoop Connector Installation Procedure*** (page 323) for details).

Reading Data from Vertica

To read data from a Vertica database, you tell Pig Latin's LOAD statement to use a SQL query and to use the `VerticaLoader` class as the load function. Your query can be hard coded, or contain a parameter. See ***Setting the Query to Retrieve Data from Vertica*** (page 325) for details.

Note: You can only use a discrete parameter list or supply a query to retrieve parameter values—you cannot use a collection to supply parameter values as you can from within a Hadoop application.

The format for calling the `VerticaLoader` is:

```
com.vertica.pig.VerticaLoader('hosts','database','port','username','password');
```

hosts	A comma-separated list of the hosts in the Vertica cluster.
database	The name of the database to be queried.
port	The port number for the database.
username	The username to use when connecting to the database.
password	The password to use when connecting to the database. This is the only optional parameter. If not present, the Hadoop Connector assumes the password is empty.

The following Pig Latin command extracts all of the data from the table named allTypes using a simple query:

```
A = LOAD 'sql://{SELECT * FROM allTypes ORDER BY key}' USING
  com.vertica.pig.VerticaLoader('Vertica01,Vertica02,Vertica03',
  'ExampleDB','5433','ExampleUser','password123');
```

This example uses a parameter and supplies a discrete list of parameter values:

```
A = LOAD 'sql://{SELECT * FROM allTypes WHERE key = ?};{1,2,3}' USING
    com.vertica.pig.VerticaLoader('Vertica01,Vertica02,Vertica03',
    'ExampleDB','5433','ExampleUser','password123');
```

This final example demonstrates using a second query to retrieve parameters from the Vertica database.

```
A = LOAD 'sql://{SELECT * FROM allTypes WHERE key = ?};sql://{SELECT DISTINCT key FROM allTypes}'
    USING com.vertica.pig.VerticaLoader('Vertica01,Vertica02,Vertica03','ExampleDB',
    '5433','ExampleUser','password123');
```

Writing Data to Vertica

To write data to a Vertica database, you tell Pig Latin's STORE statement to save data to a database table (optionally giving the definition of the table) and to use the `VerticaStorer` class as the save function. If the table you specify as the destination does not exist, and you supplied the table definition, the table is automatically created in your Vertica database and the data from the relation is loaded into it.

The syntax for calling the `VerticaStorer` is the same as calling `VerticaLoader`:

```
com.vertica.pig.VerticaStorer('hosts','database','port','username','password');
```

The following example demonstrates saving a relation into a table named hadoopOut which must already exist in the database:

```
STORE A INTO '{hadoopOut}' USING
    com.vertica.pig.VerticaStorer('Vertica01,Vertica02,Vertica03','ExampleDB','5433',
    'ExampleUser','password123');
```

This example shows how you can add a table definition to the table name, so that the table is created in Vertica if it does not already exist:

```
STORE A INTO '{outTable(a int, b int, c float, d char(10), e varchar, f boolean, g date,
   h timestamp, i timestamptz, j time, k timetz, l varbinary, m binary,
   n numeric(38,0), o interval)}' USING
   com.vertica.pig.VerticaStorer('Vertica01,Vertica02,Vertica03','ExampleDB','5433',
```

```
'ExampleUser','password123');
```

Note: If the table already exists in the database, and the definition that you supply differs from the table's definition, the table is not dropped and recreated. This may cause data type errors when data is being loaded.

Using Informatica PowerCenter

Informatica's PowerCenter family of products let you collect, transform, and store data. They support a wide variety of data sources including databases, message queues, and many different file formats.

You can use Vertica with Informatica PowerCenter both as a source and as a target using an ODBC connection, the same way you would use any other ODBC data source with PowerCenter.

Note: The default buffer size for Informatica PowerCenter is set very conservatively. These settings can cause PowerCenter to send Vertica many small batches, rather than a few large batches. The overhead of these many small batches can cause loading performance issues. To resolve these performance issues, you should change PowerCenter's batch size settings, as described in ***Setting PowerCenter's Buffer Size*** (page 357).

There is a Vertica plug-in for PowerCenter that makes using Vertica as a target for PowerCenter that is more efficient than using ODBC. If you plan on using Vertica as a target for PowerCenter, you should install and use this plug-in.

Note: Currently, the Vertica plug-in for PowerCenter is write-only. If you need to use Vertica as a data source, you will need to use an ODBC connection.

The following sections explain how to use PowerCenter with Vertica.

Installing the Vertica Plug-in for PowerCenter

There is a client and a server component for the Vertica Plug-in for PowerCenter that you need to download from http://myvertica.vertica.com/v-zone/download_vertica ***http://myvertica.vertica.com/v-zone/download_vertica***.

The client portion of the plug-in is contained in a file named:

`vertica-informatica-plugin-client-5.0.`*`xx`*`.zip`

The *xx* is the minor release number of Vertica. This package contains three files:

- `vertica.xml` contains the metadata definition needed by the PowerCenter repository to allow communication between PowerCenter and Vertica.
- `verticacli.dll` is the Windows library for the PowerCenter client.
- `vertica.reg` contains the settings for the Windows registry to support the plug-in.

There are two server component packages available, one for each platform:

- For Windows servers, download `vertica-informatica-plugin-server-`5.0.*`xx`*`.zip`
- For Linux/Solaris servers, download `vertica-informatica-plugin-server-`5.0.*`xx`*`.tar.gz`

Each of these package contain libraries used by the PowerCenter server. The Windows package contains library files for both the 32-bit and 64-bit version of PowerCenter. The Linux/Solaris packages contains libraries for 32-bit and 64-bit Linux and Solaris 5.10.

Installing the Vertica plug-in is a multi-step process:

1. Register the plug-in's metadata with the PowerCenter Repository Service with which you that you want to access Vertica.
2. Add the client plug-in's configuration information to the Window's registry of all the PowerCenter Clients that need to access Vertica.
3. Copy the Vertica client plug-in library to the Informatica PowerCenter Client's binary folder.
4. Copy the server plug-in to the PowerCenter server binary directory.

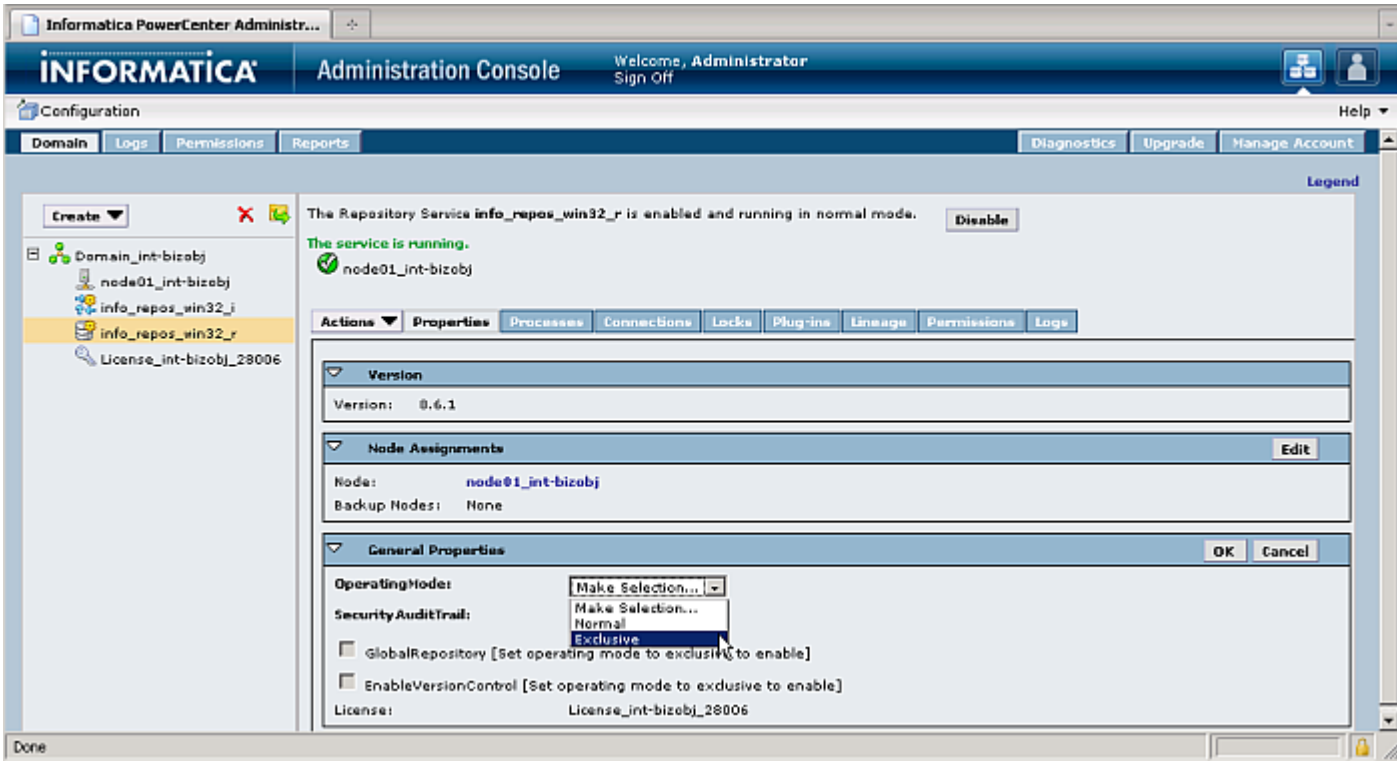
The following sections explain these steps in greater detail.

Registering the Plug-in's Metadata

The PowerCenter repository needs information about the Vertica plug-in in order to enable clients to use it. This information is supplied in an XML-format file named `vertica.xml` located in the Windows client package (`vertica-informatica-plugin-client-5.0.nn.zip`).

To register the plug-in's metadata:

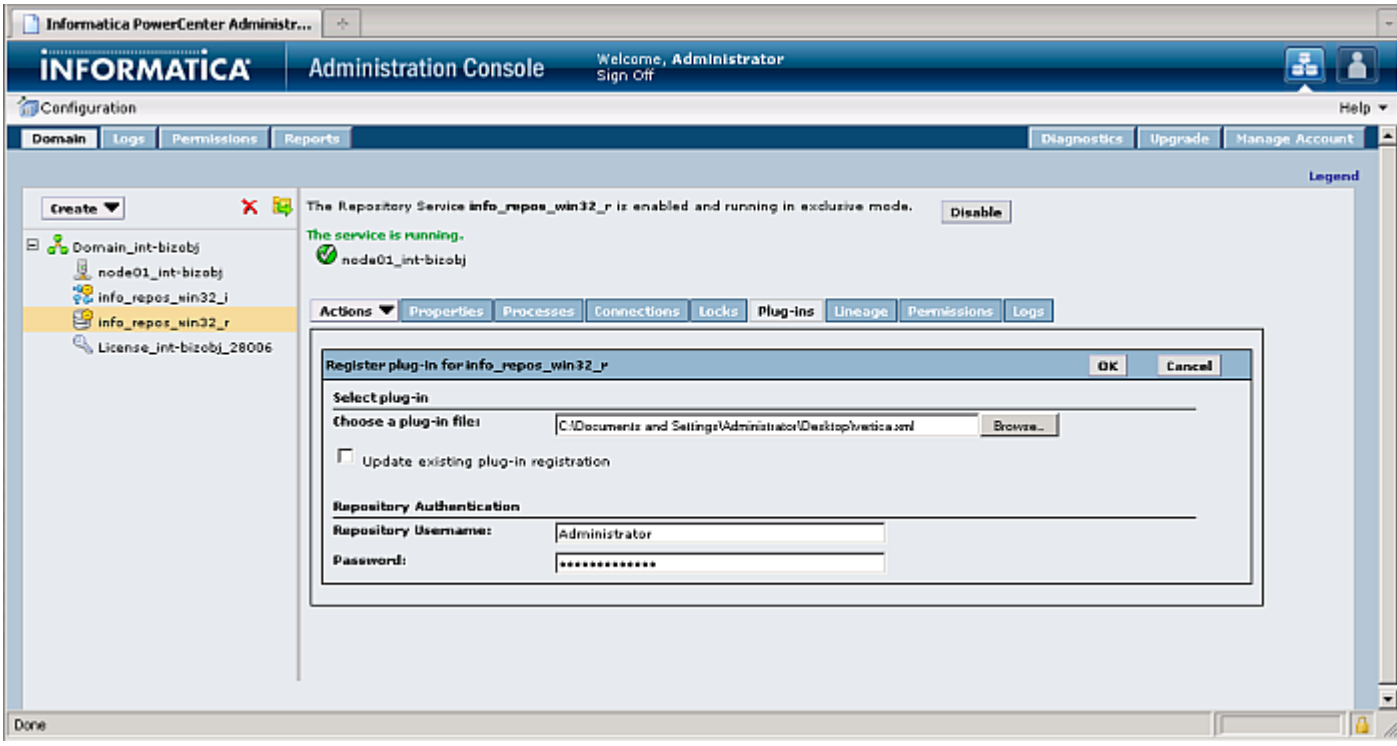
1. Unzip `vertica-informatica-plugin-client-5.0.nn.zip` to a convenient folder on your system.
2. Open a browser and log into the PowerCenter domain's Administration Console.
3. In the Navigator, click the entry for the repository that you want to connect to Vertica.
4. In the **Properties** tab click **Edit** in the the General Properties section.



5. In the **OperatingMode** box, click **Exclusive** then click **OK**.
6. In the Restart Repository Service window, click **Yes** to confirm switching to exclusive mode.

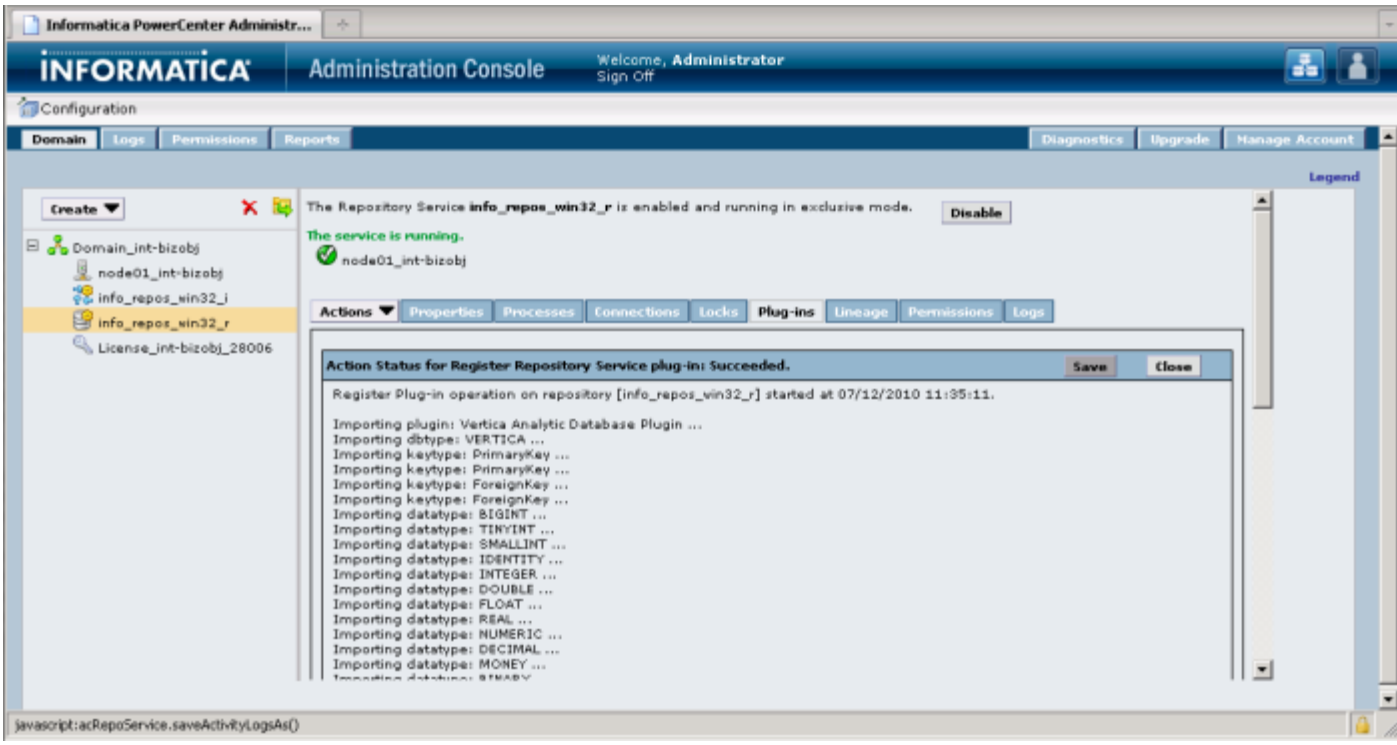
7 When prompted for a disable option, select **Complete** and click **OK**. The Repository Service may take several minutes to restart and re-enable itself. You should wait until you see the green "The service is running" status message before continuing.

8 On the Plug-ins tab, click **Register Repository Service plug-in**.



9 Next to **Choose a plug-in file**, click **Browse** and select the `vertica.xml` in the folder where you earlier unzipped the client plug-in `.zip` file.

10 Enter your repository username and password under the Repository Authentication section.



11 Click **OK** to upload the metadata file. The Administration Console uploads the metadata file and registers the plug-in data. You should see a notice indicating that the registration for the plug-in succeeded.

12 On the Properties tab's General Properties section, click **Edit**.

13 In the **OperatingMode** box, click **Normal**.

14 In the Restart Repository Service window, click **Yes** to to confirm switching to normal mode.

15 When prompted for a disable option, select **Complete** and click **OK**. The Repository Service may take several minutes to restart and re-enable itself.

Preparing the PowerCenter Client

Each PowerCenter client system that you want to use with Vertica needs to have a copy of the `verticacli.dll` file installed in the client binary folder. This folder is named `client\bin` in the PowerCenter install directory. For a typical PowerCenter install, the full path of this folder is:

```
C:\Informatica\PowerCenter8.6.1\client\bin
```

After copying the library file to the client binary directory, you need to add a registry entry to the Windows registry in that tells the PowerCenter Designer to load the plug-in library. The easiest way to do this is to double click the `vertica.reg` file in Windows Explorer. When asked whether you want to add the contents of the file to the registry, click Yes.

Note: The registry file is specific to Informatica PowerCenter version 8.6.1. The Vertica Plug-in for PowerCenter has only been tested with this version. If you want to try to use it with another version of PowerCenter, you will need to manually add configuration information to the Windows registry, as explained below.

If you prefer to add the registry entry manually, follow these steps:

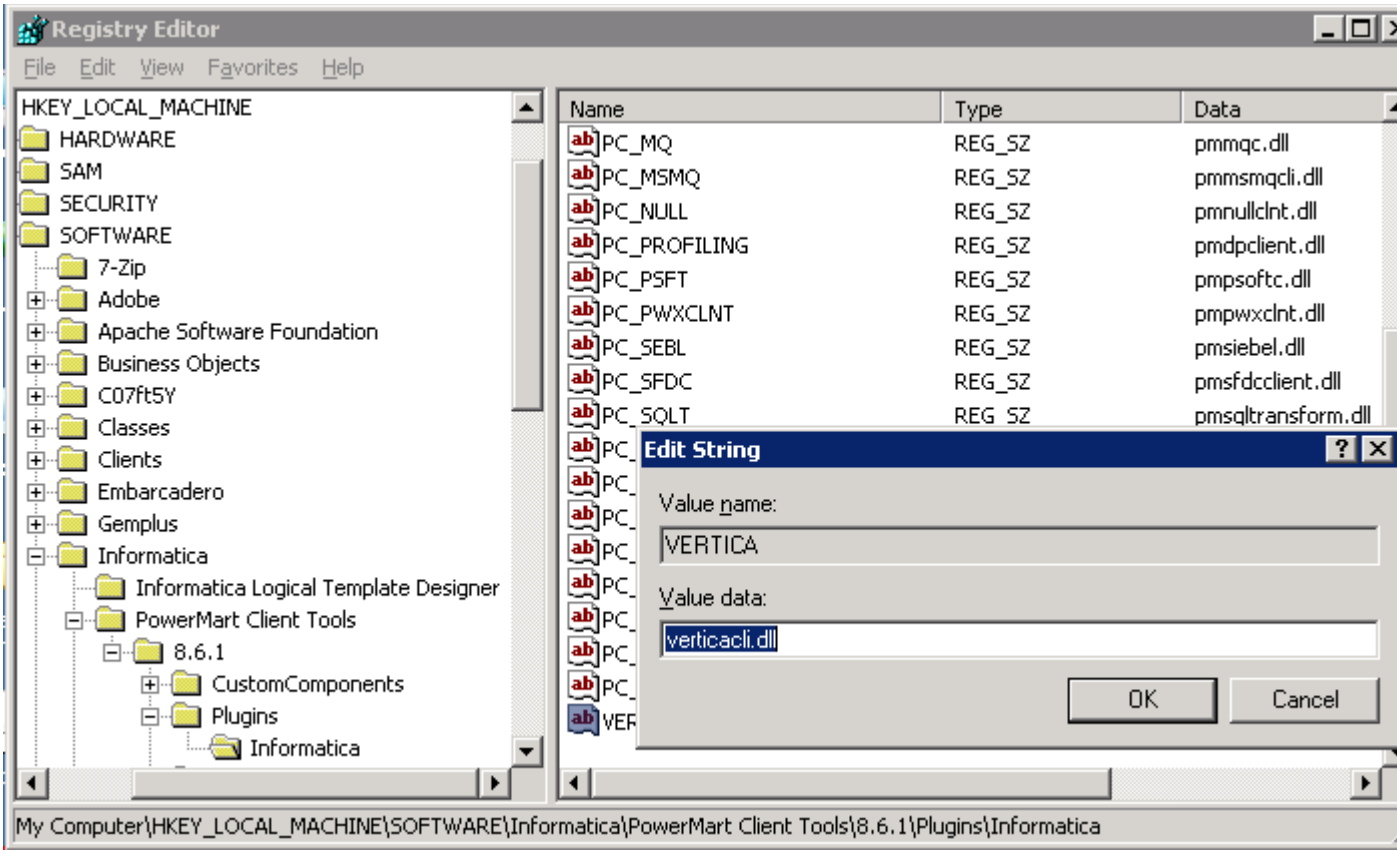
1. Start the registry editor by typing **`regedit.exe`** in the Windows Start menu's command run command box.
2. Navigate to the correct location in the registry:

   For 32-bit versions of Windows:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Informatica\PowerMart Client
     Tools\8.6.1\Plugins\Informatica
   ```

   For 64-bit versions of Windows:

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Informatica\PowerMart Client
     Tools\8.6.1\Plugins\Informatica
   ```
3. Right-click in the right pane of the Registry Editor window, select New then select String Value.
4. Change the name of the string value from New Value #1 to VERTICA.



5. Double-click the new VERTICA entry and enter **`verticacli.dll`** when prompted for a new value.
6. Exit the registry editor.

Copying the Plug-in Library on the Server

The final step in setting up the Vertica plug-in for PowerCenter is to copy the server-side library to the proper directory on the PowerCenter server. The particular library file you need to copy depends on the platform on which the PowerCenter server is running.

For a Windows server, unzip the `vertica-informatica-plugin-server-4.0.`*nn*`.zip`. There are two library files contained within the `.zip` file:

- `lib/verticawrt.dll` is for the PowerCenter 32-bit server.
- `lib64/verticawrt.dll` is for the PowerCenter 64-bit server.

Copy the appropriate library file to your server's binary directory which is the `\server\bin` subdirectory in the PowerCenter server install directory. The full path to this directory is usually:

```
C:\Informatica\PowerCenter8.6.1\server\bin
```

For a Linux or Solaris server, you need to untar `vertica-informatica-plugin-server-4.0.`*nn*`.tar.gz` to a temporary directory on the server:

```
# cd /tmp
# tar xzf  vertica-informatica-plugin-server-4.0.nn.tar.gz
```

This archive contains three library files:

- `linux/lib/libverticawrt.so` for PowerCenter Linux 32-bit server.
- `linux/lib64/libverticawrt.so` for PowerCenter Linux 64-bit server.
- `SunOS510/libverticawrt.so` for Solaris server.

Copy the appropriate library file to the `server/bin` subdirectory of the directory where PowerCenter is installed. For a typical PowerCenter install, this path is:

```
/Infra/PowerCenter8.6.1/server/bin/
```

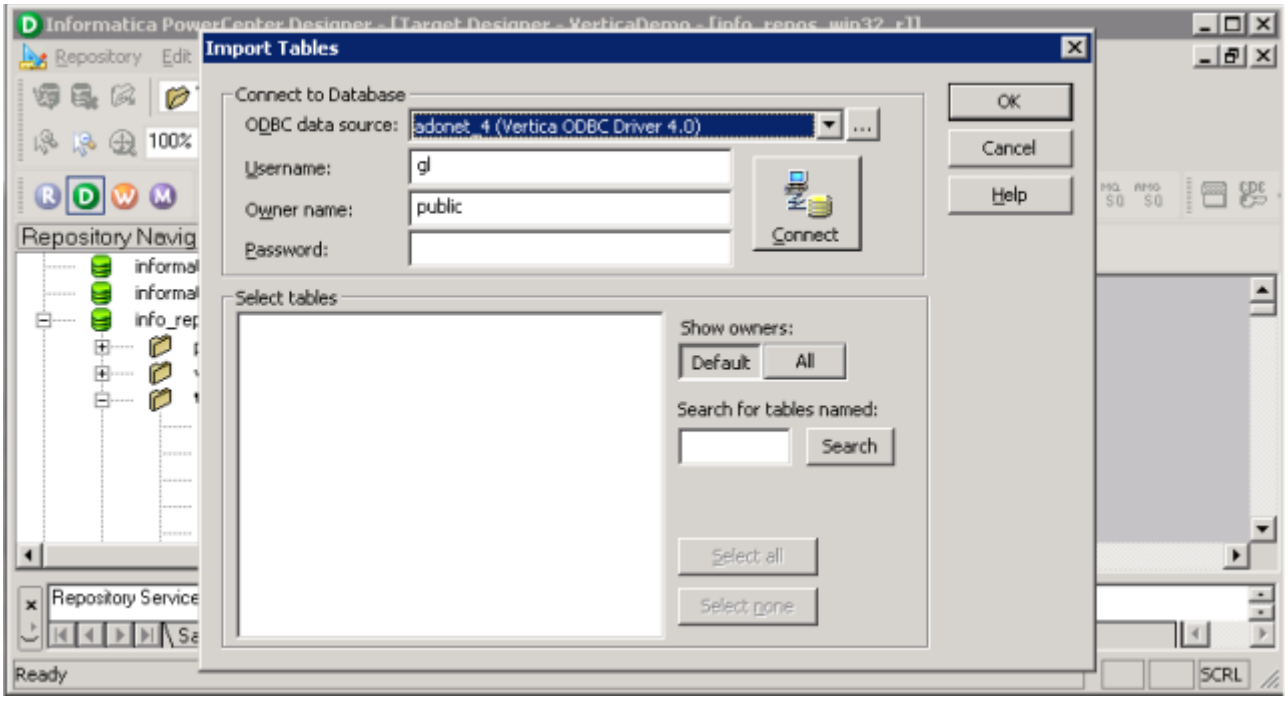
Using the Vertica Plug-in for PowerCenter

Once you have installed the Vertica plug-in for PowerCenter, you can use Vertica as a target in PowerCenter Designer. There is a slight complication caused by the fact that the Vertica plug-in for PowerCenter is read-only. This means that when you create a target definition for a Vertica database table, PowerCenter Designer cannot read the table's definition from the database. The best workaround is to manually define the table's columns in PowerCenter Designer. However, this solution is impractical for anything other than the simplest table.

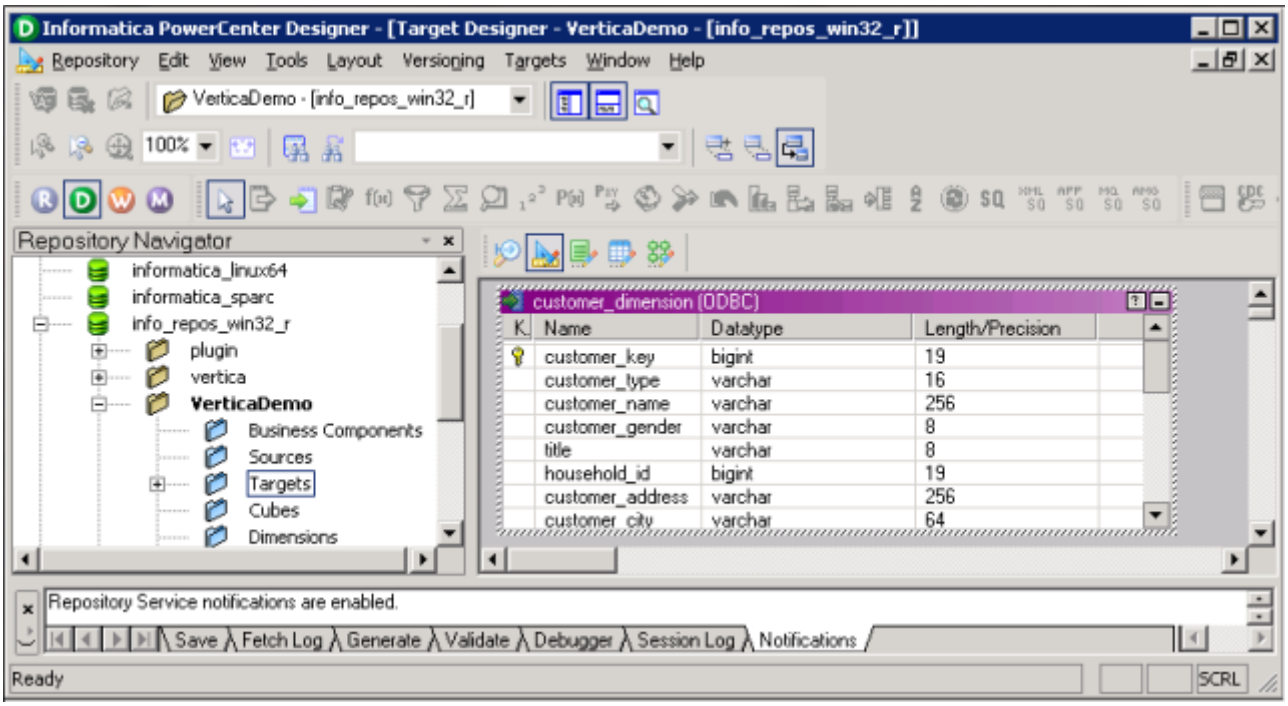
Instead of manually recreating the table's definition, you can create the target definition using an ODBC connection to the database. PowerCenter Designer can import table definitions from Vertica when using an ODBC connection. After the definitions have been imported, you change the table's database type to VERTICA, so it will use the plug-in to connect to Vertica. To use this technique, you first need to ***create a DSN for the Vertica database*** (page 27) even if you do not plan on connecting to the database using ODBC in your live environment.

For example, to target a table in a Vertica database, you could follow these steps:

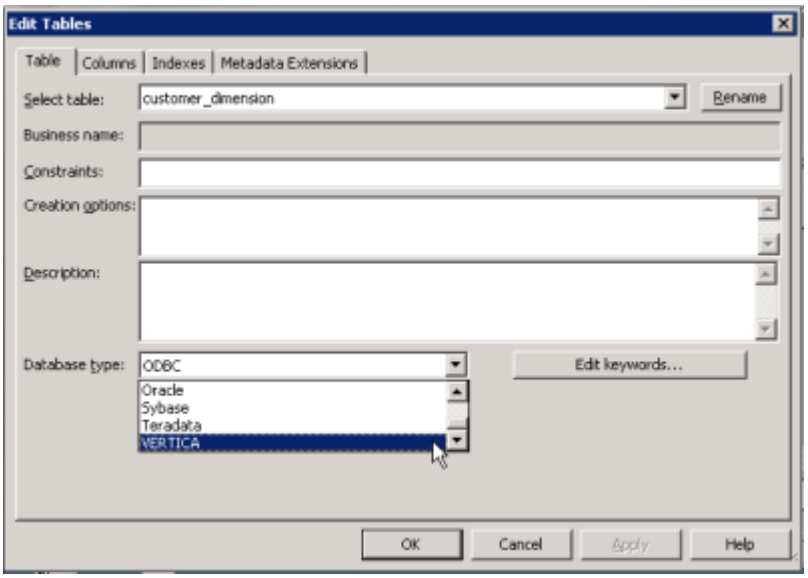
1. In PowerCenter Designer Navigator, select the folder in the repository where you want to create your Vertica target.
2. On the **Tools** menu, click **Target Designer**.
3. On the **Targets** menu, click **Import from Database**.



4. In the **ODBC data source** box, click the name of the DSN you created for your Vertica database.
5. Enter the **Username**, **Owner name**, and **Password** for your database, then click **Connect**. PowerCenter Designer connects to your database and retrieves a list of the tables it contains.
6. In the **Select tables** box, click the table into which you want PowerCenter to store data and click **OK**. PowerCenter Designer reads the definition of the table and displays it in the Workspace.



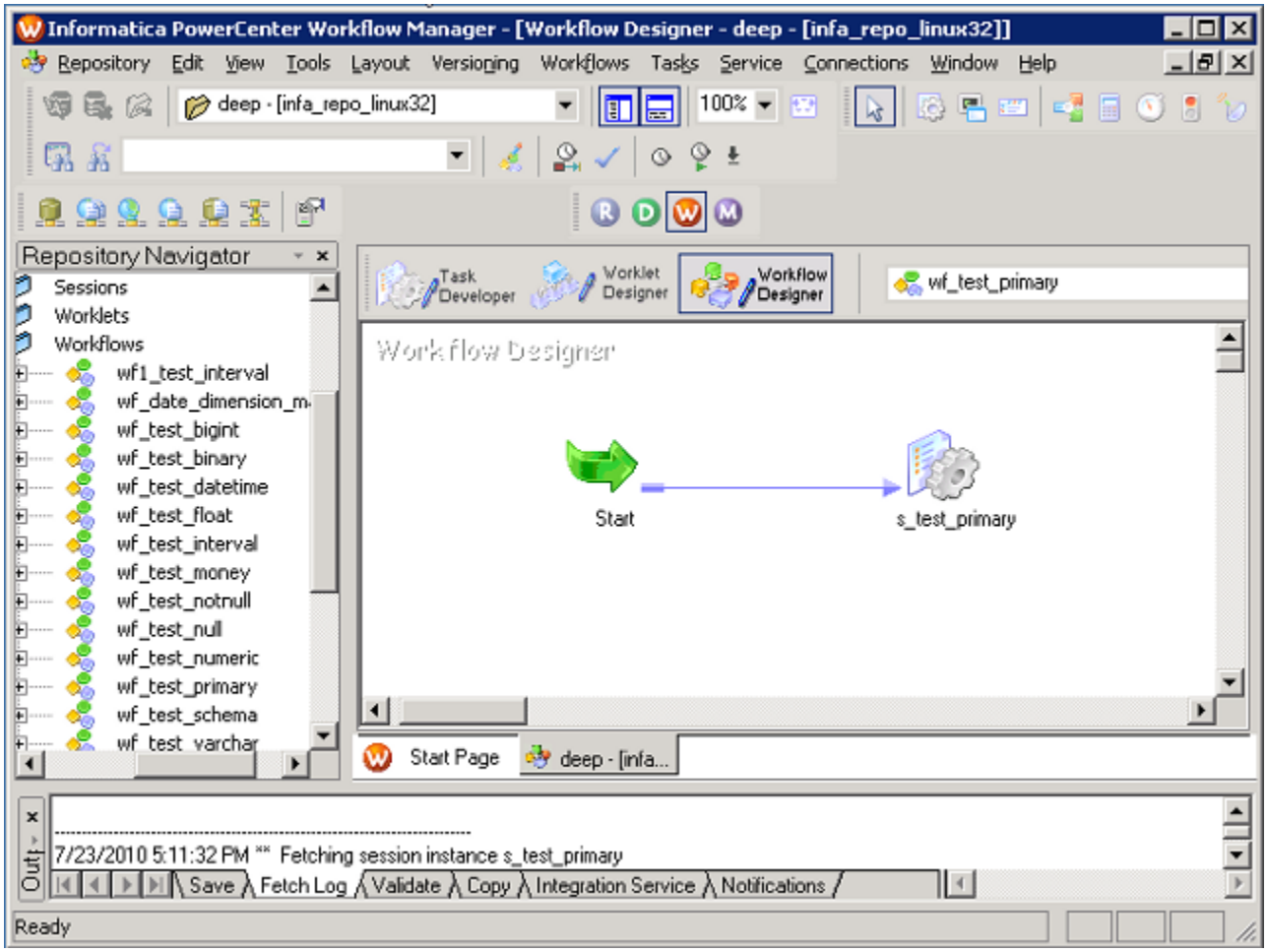
7 Right-click the table in the Workspace and click **Edit**.



8 In the Edit Table window's **Database type** box, click **VERTICA**, then click **OK**.

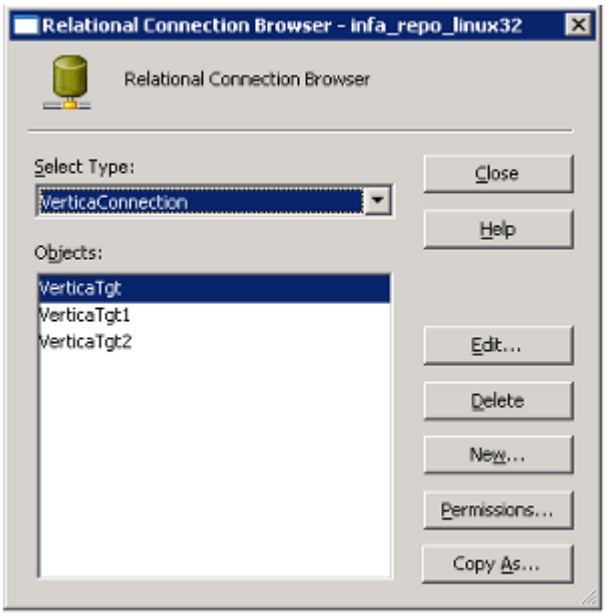
Using the Plug-in in a Workflow

To use the plug-in's connection to Vertica within your workflows, you need to select it from the workflow's connection properties.



1 In the Workflow Manager, select the workflow that you want to target Vertica.
2 On the **Connections** menu, click **Relational**.

3 In the Relational Connection Browser's **Select Type** box, click **VerticaConnection**.



4 In the **Objects** box, click the connection to the Vertica that you want to be the target of the workflow.

5 Before starting the workflow, you should change PowerCenter's buffer size to more efficiently load data into Vertica. See ***Setting PowerCenter's Buffer Size*** (page 357) for details.

Truncating the Target Table

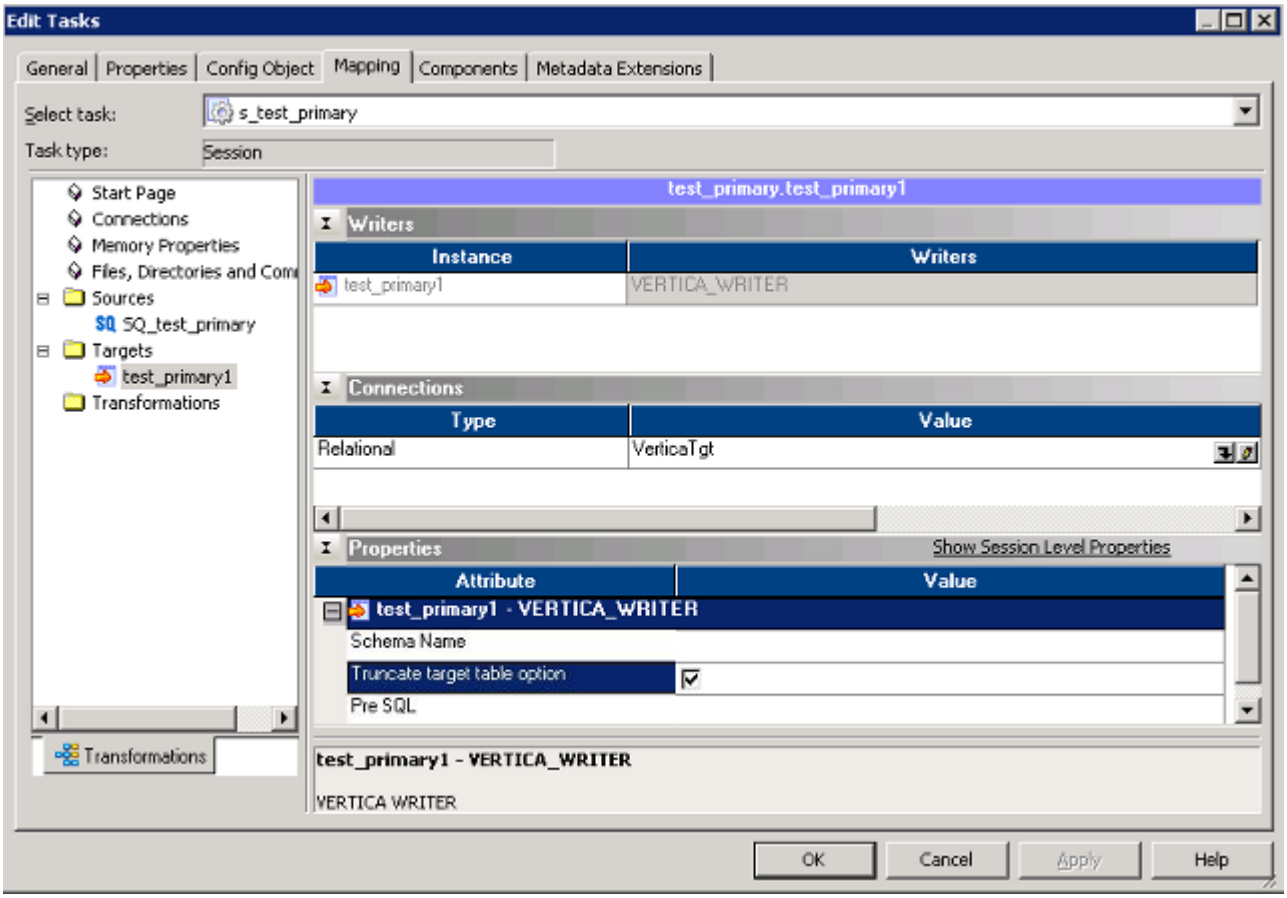
You may have a workflow that should truncate its targeted table before loading data. You can change a plug-in setting to truncate the table for you:

1 In Workflow Manager, select the workflow that should truncate its target table.

2 In the Workspace, double-click the data load task to open the Edit Tasks window.

3 In the Edit Tasks window, click the **Mapping** tab.

4 In the navigation pane, click the connection to your Vertica database under **Targets**.



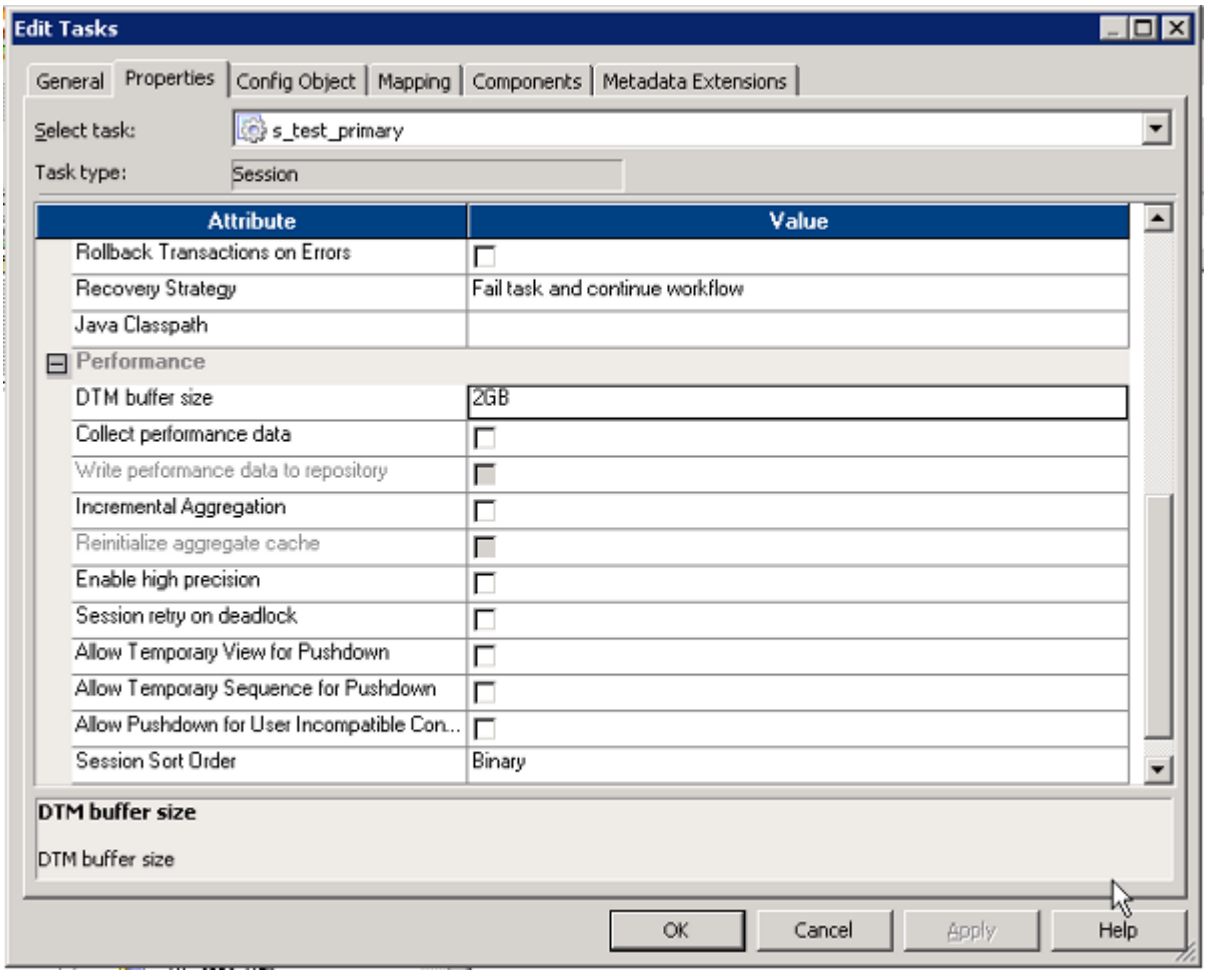
5 In the Properties section, select the **Truncate target table option**.

Setting PowerCenter's Buffer Size

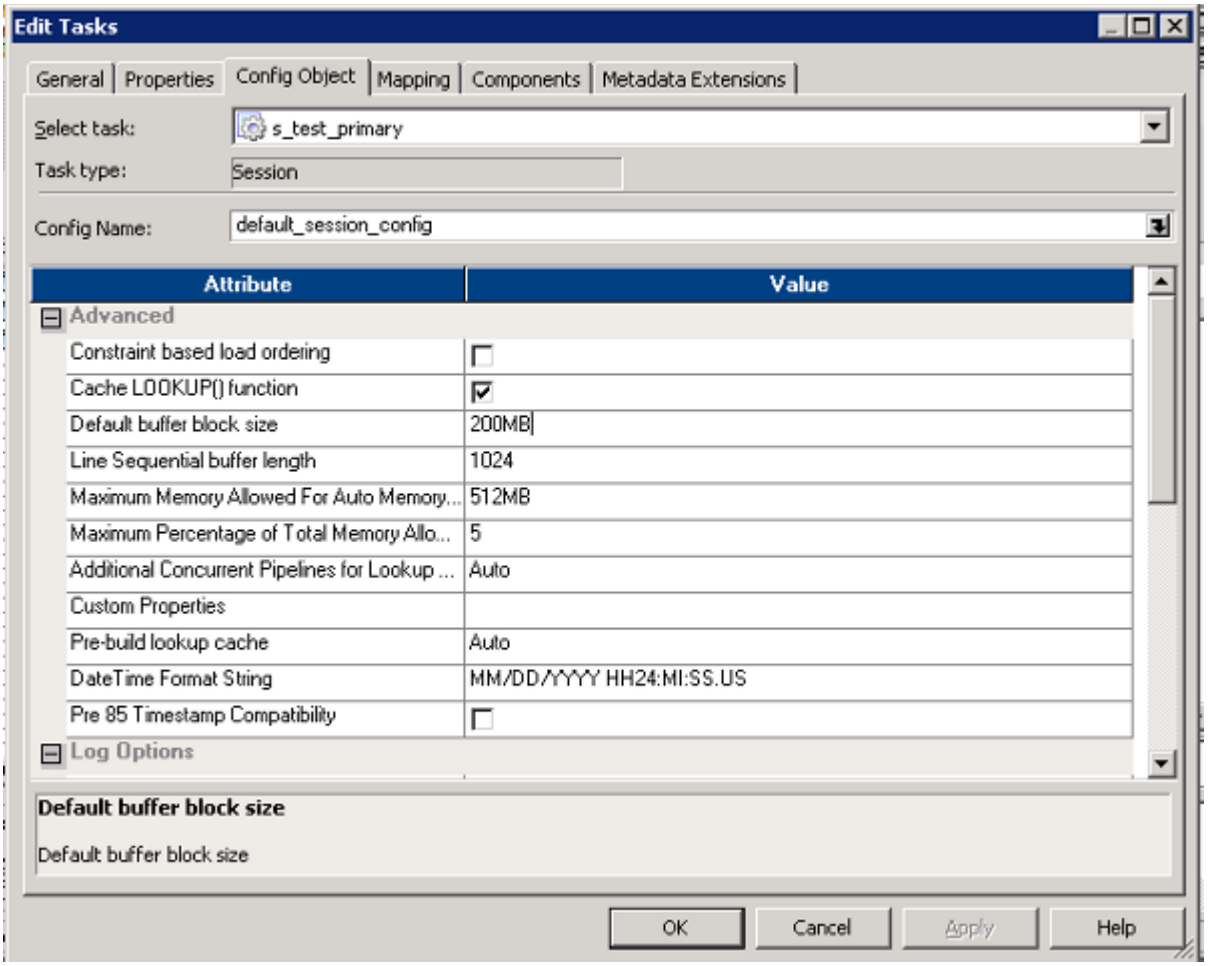
By default, Informatica Powercenter's buffer are set to very conservative values (12MB overall buffer size, with the buffer block size set automatically). This can cause performance issues when loading data into Vertica, since PowerCenter will send many small batches, rather than fewer large batches.

To improve performance, you should adjust the batch buffer sizes for connections to Vertica:

1 In the Workflow Manager, double-click the task that connects to Vertica.



2 In the Edit Tasks window's **Properties** tab, set the **DTM buffer size** to 2GB.



3 On the **Config Object** tab, set the **Default buffer** block size to 200MB.

Appendix: Error Codes

Error Codes

All messages emitted by the Vertica server are assigned five-character error codes that follow the SQL standard's conventions for "SQLSTATE" codes. Applications that need to know which error condition has occurred can test the error code, rather than looking at the textual error message. The error codes are less likely to change across Vertica releases.

Note: Some of the error codes produced by Vertica are defined by the SQL standard, According to the standard, the first two characters of an error code denote a class of errors, while the last three characters indicate a specific condition within that class. Thus, an application that does not recognize the specific error code can still infer what to do from the error class.

Vertica Error Codes

Error Code	Meaning	Example
Class 00	**ERRCODE_SUCCESSFUL_COMPLETION**	
00000	Successful completion	
Class 01	**WARNING**	**<Class01 Error Code Examples (page 374)>**
00001	Warning	
01003	ERRCODE_WARNING_NULL_VALUE_ELIMINATED_IN _SET_FUNCTION	
01004	ERRCODE_WARNING_STRING_DATA_RIGHT_ TRUNCATION	
01006	ERRCODE_WARNING_PRIVILEGE_NOT_REVOKED	
01007	ERRCODE_WARNING_PRIVILEGE_NOT_GRANTED	
01008	ERRCODE_WARNING_IMPLICIT_ZERO_BIT_PADDING	
0100C	ERRCODE_WARNING_DYNAMIC_RESULT_SETS_ RETURNED	

01V01	ERRCODE_WARNING_DEPRECATED_FEATURE	
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	
Class 08	**ERRCODE CONNECTION EXCEPTION**	**<Class08 Error Code Examples (page 374)>**
08000	ERRCODE_CONNECTION_EXCEPTION	
08001	ERRCODE_SQLCLIENT_UNABLE_TO_ESTABLISH_ SQLCONNECTION	
08003	ERRCODE_CONNECTION_DOES_NOT_EXIST	
08004	ERRCODE_SQLSERVER_REJECTED_ESTABLISHMENT _OF_SQLCONNECTION	
08006	ERRCODE_CONNECTION_FAILURE	
08007	ERRCODE_TRANSACTION_RESOLUTION_UNKNOWN	
08V01	0x01026200 ERRCODE_PROTOCOL_VIOLATION	
Class 09	**TRIGGERED ACTION EXCEPTION**	
09000	ERRCODE_TRIGGERED_ACTION_EXCEPTION	
Class 0A	**FEATURE NOT SUPPORTED**	**<Class0A ErrCode Examples**

		(page 375)>
0A000	ERRCODE_FEATURE_NOT_SUPPORTED	
Class 0B	**INVALID TRANSACTION INITIATION**	
000B0	ERRCODE_INVALID_TRANSACTION_INITIATION	
Class 0F	**LOCATOR EXCEPTION**	
0F000	ERRCODE_LOCATOR_EXCEPTION	
0F001	ERRCODE_L_E_INVALID_SPECIFICATION	
Class 0L	**INVALID GRANTOR**	**<Class0L ErrCode Examples (page 377)>**
0L000	ERRCODE_INVALID_GRANTOR	
0LV01	ERRCODE_INVALID_GRANT_OPERATION	
Class 0P	**INVALID ROLE SPECIFICATION**	
0P000	ERRCODE_INVALID_ROLE_SPECIFICATION	
Class 21	**CARDINALITY VIOLATION**	
21000	ERRCODE_CARDINALITY_VIOLATION	
Class 22	**DATA EXCEPTION**	**<Class22 Error Code Examples (page 377)>**
22000	ERRCODE_DATA_EXCEPTION	
22001	ERRCODE_STRING_DATA_RIGHT_TRUNCATION	
22002	ERRCODE_NULL_VALUE_NO_INDICATOR_PARAMETER	
22003	ERRCODE_NUMERIC_VALUE_OUT_OF_RANGE	
22004	ERRCODE_NULL_VALUE_NOT_ALLOWED	

22005	ERRCODE_ERROR_IN_ASSIGNMENT	
22007	ERRCODE_INVALID_DATETIME_FORMAT	
22008	ERRCODE_DATETIME_FIELD_OVERFLOW ERRCODE_DATETIME_VALUE_OUT_OF_RANGE	
22009	ERRCODE_INVALID_TIME_ZONE_DISPLACEMENT_ VALUE	
2200B	ERRCODE_ESCAPE_CHARACTER_CONFLICT	
2200C	ERRCODE_INVALID_USE_OF_ESCAPE_CHARACTER	
2200D	ERRCODE_INVALID_ESCAPE_OCTET	
2200F	ERRCODE_ZERO_LENGTH_CHARACTER_STRING	
2200G	ERRCODE_MOST_SPECIFIC_TYPE_MISMATCH	
22010	ERRCODE_INVALID_INDICATOR_PARAMETER_VALUE	
22011	ERRCODE_SUBSTRING_ERROR	
22012	ERRCODE_DIVISION_BY_ZERO	
22015	ERRCODE_INTERVAL_FIELD_OVERFLOW	
22018	ERRCODE_INVALID_CHARACTER_VALUE_FOR_CAST	
22019	ERRCODE_INVALID_ESCAPE_CHARACTER	
2201B	ERRCODE_INVALID_REGULAR_EXPRESSION	
2201E	ERRCODE_INVALID_ARGUMENT_FOR_LOG	
2201F	ERRCODE_INVALID_ARGUMENT_FOR_PO	

	WER_ FUNCTION	
2201G	ERRCODE_INVALID_ARGUMENT_FOR_WIDTH_ BUCKET_FUNCTION	
22020	ERRCODE_INVALID_LIMIT_VALUE	
22021	ERRCODE_CHARACTER_NOT_IN_REPERTOIRE	
22022	ERRCODE_INDICATOR_OVERFLOW	
22023	ERRCODE_INVALID_PARAMETER_VALUE	
22024	ERRCODE_UNTERMINATED_C_STRING	
22025	ERRCODE_INVALID_ESCAPE_SEQUENCE	
22026	ERRCODE_STRING_DATA_LENGTH_MISMATCH	
22027	ERRCODE_TRIM_ERROR	
2202E	ERRCODE_ARRAY_ELEMENT_ERROR ERRCODE_ARRAY_SUBSCRIPT_ERROR	
22906	ERRCODE_NONSTANDARD_USE_OF_ESCAPE_ CHARACTER	
22V01	ERRCODE_FLOATING_POINT_EXCEPTION	
22V02	ERRCODE_INVALID_TEXT_REPRESENTATION	
22V03	0x03026082 ERRCODE_INVALID_BINARY_ REPRESENTATION	
22V04	ERRCODE_BAD_COPY_FILE_FORMAT	
22V05	ERRCODE_UNTRANSLATABLE_CHARACTER	

22V21	ERRCODE_INVALID_EPOCH	
Class 23	**INTEGRITY CONSTRAINT VIOLATION**	
23000	ERRCODE_INTEGRITY_CONSTRAINT_VIOLATION	
23001	ERRCODE_RESTRICT_VIOLATION	
23502	ERRCODE_NOT_NULL_VIOLATION	ROLLBACK "column \"%s\" contains null values" WARNING "column \"%s\" definition changed to NOT NULL"
23503	ERRCODE_FOREIGN_KEY_VIOLATION	ROLLBACK "Nonexistent foreign key value detected in FK-PK join %s; value %s"
23505	ERRCODE_UNIQUE_VIOLATION	ROLLBACK "Duplicate primary key detected in FK-PK join %s, value %s"
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	
25004	ERRCODE_INAPPROPRIATE_ISOLATION_LEVEL_ FOR_BRANCH_TRANSACTION	
25005	ERRCODE_NO_ACTIVE_SQL_TRANSACTI	

	ON_FOR_ BRANCH_TRANSACTION	
25006	ERRCODE_READ_ONLY_SQL_TRANSACTION	ERROR "Cannot issue this command in a read-only transaction"
25007	ERRCODE_SCHEMA_AND_DATA_STATEMENT_MIXING _NOT_SUPPORTED	
25008	ERRCODE_HELD_CURSOR_REQUIRES_SAME_ ISOLATION_LEVEL	
25V01	ERRCODE_NO_ACTIVE_SQL_TRANSACTION	ROLLBACK "cannot advance epoch without a transaction"
25V02	ERRCODE_IN_FAILED_SQL_TRANSACTION	
Class 26	**Invalid SQL Statement Name**	**<Class26 Error Code Examples (page 379)>**
26000	ERRCODE_INVALID_SQL_STATEMENT_NAME ERRCODE_UNDEFINED_PSTATEMENT	
Class 27	**TRIGGERED DATA CHANGE VIOLATION**	
27000	ERRCODE_TRIGGERED_DATA_CHANGE_VIOLATION	
Class 28	**INVALID AUTHORIZATION SPECIFICATION**	**<Class28 Error Code Examples (page 379)>**
28000	ERRCODE_INVALID_AUTHORIZATION_ SPECIFICATION	
Class 2B	**Dependent Privilege Descriptors Still Exist**	
2B000	ERRCODE_DEPENDENT_PRIVILEGE_DESCRIPTORS _STILL_EXIST	
2BV01	ERRCODE_DEPENDENT_OBJECTS_STILL_EXIST	ERROR "DROP failed due to dependencies" ERROR "dependent privileges exist"

Class 2D	**Invalid Transaction Termination**	
2D000	ERRCODE_INVALID_TRANSACTION_TERMINATION	
Class 2F	**SQL Routine Exception**	
2F000	ERRCODE_SQL_ROUTINE_EXCEPTION	
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 ERRCODE_UNDEFINED_CURSOR	ERROR "portal \"%s\" does not exist"
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	
Class 3D	**Invalid Catalog Name**	
3D000	ERRCODE_INVALID_CATALOG_NAME ERRCODE_UNDEFINED_DATABASE	ERROR "database \"%s\" does not exist" FATAL "database \"%s\" does not exist" ROLLBACK "Unable to read catalog file %s"
Class 3F	**Invalid Schema Name**	
3F000	ERRCODE_INVALID_SCHEMA_NAME ERRCODE_UNDEFINED_SCHEMA	ERROR "no schema has been selected to create in" ERROR "schema \"%s\" does not exist"
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	ROLLBACK "Txn %#llx: %s error %s"
Class 42	**Syntax Error or Access Rule Violation**	**<Class42 Error Code Examples (page 379)>**
42000	ERRCODE_SYNTAX_ERROR_OR_ACCESS_RULE_VIOLATION	
42501	ERRCODE_INSUFFICIENT_PRIVILEGE	
42601	ERRCODE_SYNTAX_ERROR	
42602	ERRCODE_INVALID_NAME	
42611	ERRCODE_INVALID_COLUMN_DEFINITION	
42622	ERRCODE_NAME_TOO_LONG	
42701	ERRCODE_DUPLICATE_COLUMN	
42702	ERRCODE_AMBIGUOUS_COLUMN	
42703	ERRCODE_UNDEFINED_COLUMN	
42704	ERRCODE_UNDEFINED_OBJECT	
42710	ERRCODE_DUPLICATE_OBJECT	
42712	ERRCODE_DUPLICATE_ALIAS	
42723	ERRCODE_DUPLICATE_FUNCTION	
42725	ERRCODE_AMBIGUOUS_FUNCTION	
42803	ERRCODE_GROUPING_ERROR	

42804	ERRCODE_DATATYPE_MISMATCH	
42809	ERRCODE_WRONG_OBJECT_TYPE	
42830	ERRCODE_INVALID_FOREIGN_KEY	
42846	ERRCODE_CANNOT_COERCE	
42883	ERRCODE_UNDEFINED_FUNCTION	
42939	ERRCODE_RESERVED_NAME	
42V01	ERRCODE_UNDEFINED_TABLE	
42V02	ERRCODE_UNDEFINED_PARAMETER	
42V03	ERRCODE_DUPLICATE_CURSOR	
42V04	ERRCODE_DUPLICATE_DATABASE	
42V05	ERRCODE_DUPLICATE_PSTATEMENT	
42V06	ERRCODE_DUPLICATE_SCHEMA	
42V07	ERRCODE_DUPLICATE_TABLE	
42V08	ERRCODE_AMBIGUOUS_PARAMETER	
42V09	ERRCODE_AMBIGUOUS_ALIAS	
42V10	ERRCODE_INVALID_COLUMN_REFERENCE	
42V11	ERRCODE_INVALID_CURSOR_DEFINITION	
42V12	ERRCODE_INVALID_DATABASE_DEFINITION	
42V13	ERRCODE_INVALID_FUNCTION_DEFINITION	
42V14	ERRCODE_INVALID_PSTATEMENT_DEFINITION	
42V15	ERRCODE_INVALID_SCHEMA_DEFINITION	
42V16	ERRCODE_INVALID_TABLE_DEFINITION	

42V17	ERRCODE_INVALID_OBJECT_DEFINITION	
42V18	ERRCODE_INDETERMINATE_DATATYPE	
42V21	ERRCODE_UNDEFINED_PROJECTION	
42V22	ERRCODE_UNDEFINED_NODE	
42V23	ERRCODE_UNDEFINED_PERMUTATION	
42V24	ERRCODE_UNDEFINED_USER	
Class 44	**WITH CHECK OPTION Violation**	
44000	ERRCODE_WITH_CHECK_OPTION_VIOLATION	
Class 53	**Insufficient Resources**	**<Class53 Error Code Examples (page 385)>**
53000	ERRCODE_INSUFFICIENT_RESOURCES	
53100	ERRCODE_DISK_FULL	
53200	ERRCODE_OUT_OF_MEMORY	
53300	ERRCODE_TOO_MANY_CONNECTIONS	
Class 54	**Program Limit Exceeded**	**<Class54 Error Code Examples (page 385)>**
54000	ERRCODE_PROGRAM_LIMIT_EXCEEDED	
54001	ERRCODE_STATEMENT_TOO_COMPLEX	
54011	ERRCODE_TOO_MANY_COLUMNS	
54023	ERRCODE_TOO_MANY_ARGUMENTS	
Class 55	**Object Not In Prerequisite State**	**<Class55 Error Code Examples (page 386)>**
55000	ERRCODE_OBJECT_NOT_IN_PREREQUISITE_STATE	
55006	ERRCODE_OBJECT_IN_USE	
55V02	ERRCODE_CANT_CHANGE_RUNTIME_PARAM	

55V03	ERRCODE_LOCK_NOT_AVAILABLE	
Class 57	**Operator Intervention**	**<Class57 Error Code Examples (page 387)>**
57000	ERRCODE_OPERATOR_INTERVENTION	
57014	ERRCODE_QUERY_CANCELED	
57V01	ERRCODE_ADMIN_SHUTDOWN	
57V02	ERRCODE_CRASH_SHUTDOWN	
57V03	ERRCODE_CANNOT_CONNECT_NOW	
Class 58	**System Error**	**<Class58 Error Code Examples (page 387)>**
58030	ERRCODE_IO_ERROR	
58V01	ERRCODE_UNDEFINED_FILE	
58V02	0x02026205 ERRCODE_DUPLICATE_FILE	
Class V	**Vertica Error**	**<ClassV Error Code Examples (page 387)>**
V1001	ERRCODE_LOST_CONNECTIVITY	
V1002	ERRCODE_K_SAFETY_VIOLATION	
V1003	ERRCODE_CLUSTER_CHANGE	
V2001	ERRCODE_LICENSE_ISSUE	
V2002	ERRCODE_MOVEOUT_ABORTED	
VC001	ERRCODE_CONFIG_FILE_ERROR	
VC002	ERRCODE_LOCK_FILE_EXISTS	
VX001	ERRCODE_INTERNAL_ERROR	
VX002	ERRCODE_DATA_CORRUPTED	
VX003	ERRCODE_INDEX_CORRUPTED	

Class 01 Error Code Examples

```
NOTICE "Cannot set locks for shutdown"
NOTICE "Cannot shut down while users are connected"
NOTICE "Shutdown for site already in progress"
WARNING "cannot resolve address"
WARNING "using /tmp for catalog path"
WARNING "Projection <%s> is not available for query processing.
         Execute the select start_refresh() function to copy data into this
         projection
WARNING "Received no response from %s%s"
WARNING "Transaction commit with NO_DISTRIBUTE set.  "
WARNING "cannot begin transaction; transaction is already running"
WARNING "no privileges could be revoked for \"%s\""
WARNING "not all privileges could be revoked for \"%s\""
WARNING "no privileges were granted for \"%s\""
WARNING "not all privileges were granted for \"%s\""
```

Class 08 Error Code Examples

```
08000
FATAL "no socket created for listening"
FATAL "unsupported frontend protocol %u.%u: server supports %u.0 to %u.%u"

08006
COMMERROR "unexpected EOF on client connection"
ERROR "Received no response from %s%s"
FATAL "SSL initialization failure"
ROLLBACK "client has disconnected"
ROLLBACK "unexpected EOF on client connection"


08V01
COMMERROR "SSL SYSCALL error: EOF detected"
COMMERROR "SSL error: %s"
COMMERROR "SSL failed to send renegotiation request"
COMMERROR "SSL renegotiation failure"
COMMERROR "could not accept SSL connection: %s"
COMMERROR "could not accept SSL connection: EOF detected"
COMMERROR "could not initialize SSL connection: %s"COMMERROR "could not set SSL socket: %s"
COMMERROR "expected password response, got message type %d"
COMMERROR "incomplete message from client"
COMMERROR "invalid message length"
COMMERROR "invalid password packet size"
COMMERROR "unexpected EOF within message length word"
COMMERROR "unrecognized SSL error code: %d"
ERROR "bind message has %d parameter formats but %d parameters"
ERROR "bind message has %d result formats but query has %d columns"
ERROR "insufficient data left in message
ERROR "invalid CLOSE message subtype %d"
ERROR "invalid DESCRIBE message subtype %d"
ERROR "invalid message format"
ERROR "invalid string in message"
ERROR "no data left in message"
FATAL "Incomplete startup packet"
FATAL "SSL negotiation failure"
FATAL "incomplete startup packet"
FATAL "invalid frontend message type %d"
FATAL "invalid length (%u) of startup packet"
FATAL "invalid startup packet layout: expected terminator as last byte"
ROLLBACK "COPY: Unexpected message type 0x%02X reading from stdin"
```

Class 0A Error Code Examples

```
ERROR "%s is not a table. DML not supported"
ERROR "%s.%s is not a table. DML not supported"
ERROR "Aggregate function %s (%llu) is not supported"
ERROR "ArrayRef is not supported"
ERROR "COPY FROM does not support BINARY option"
ERROR "COPY FROM does not support CVS option"
ERROR "COPY FROM does not support OIDS option"
ERROR "CREATE table AS SELECT... is not supported"
ERROR "CSV mode not supported. COPY HEADER available only in CSV mode"
ERROR "CSV mode not supported. COPY escape available only in CSV mode"
ERROR "CSV mode not supported. COPY force not null available only in CSV mode"
ERROR "CSV mode not supported. COPY force quote available only in CSV mode"
ERROR "CSV mode not supported. COPY quote available only in CSV mode"
ERROR "Cannot execute query."
ERROR "Cannot perform requested delete operation"
ERROR "CoalesceExpr is not supported"
ERROR "CoerceToDomain is not supported"
ERROR "CoerceToDomainValue is not supported"
ERROR "Column type int2 is not supported"
ERROR "Column type int4 is not supported"
ERROR "Complex expression in the ON clause is not supported."
ERROR "ConvertRowtypeExpr is not supported"
ERROR "DML on projection is not supported"
ERROR "Executing when OPT:PLAN_ALL_SITES_ACTIVE option is set"
ERROR "Expr is not supported"
ERROR "Expression not supported in query"
ERROR "FieldSelect is not supported"
ERROR "FieldStore is not supported"
ERROR "Function %s can't be used as a case expression"
ERROR "Function %s can't be used in a WHEN clause"
ERROR "Function %s can't be used in a boolean"
ERROR "Function %s can't be used in another function"
ERROR "Function %s can't be used with an operator"
ERROR "Group By, Order By, Aggregates, Having & limits not allowed in update/delete"
ERROR "INSTEAD NOTHING rules on SELECT are not implemented"
ERROR "Join expression not supported in where/having clause when Joins specified in From clause"
ERROR "LIMIT clause is not supported for expressions"
ERROR "Non-Boolean functions in WHERE clause"
ERROR "Not a Star or Snow-Flake Query"
ERROR "Not a Star or Snow-Flake Query; a join column appears more than once in join expressions"
ERROR "Not a Star or Snow-Flake Query; a non-lossless relationship found"
ERROR "Not a Star or Snow-Flake Query; dimension table not a star or snowflake"
ERROR "Not a Star or Snow-Flake Query; no fact table found"
ERROR "Not a Star or Snow-Flake Query; there are multiple fact tables"
ERROR "NullIfExpr is not supported"
ERROR "ORDER BY on a UNION/INTERSECT/EXCEPT result must be on one of the result columns"
ERROR "Only a relation is allowed in the FROM clause"
ERROR "Only inner joins or only outer joins are supported"
ERROR "Operator %s (%llu) is not supported"
ERROR "ROW syntax is not supported"
ERROR "RowExpression is not supported"
ERROR "SELECT FOR UPDATE cannot be applied to NEW or OLD"
ERROR "SELECT FOR UPDATE cannot be applied to a function"
ERROR "SELECT FOR UPDATE cannot be applied to a join"
ERROR "SELECT FOR UPDATE is not allowed with DISTINCT clause"
ERROR "SELECT FOR UPDATE is not allowed with GROUP BY clause"
ERROR "SELECT FOR UPDATE is not allowed with UNION/INTERSECT/EXCEPT"
ERROR "SELECT FOR UPDATE is not allowed with aggregate functions"
ERROR "SQL Feature not supported"
ERROR "Set Operators in query  is not supported"
ERROR "SetToDefault is not supported"
ERROR "Subqueries in UPDATE/DELETE is not supported"
ERROR "Subquery is not supported"
ERROR "There is an inner table that is not joining on its primary key; so outer join not supported"
ERROR "Type %s (%llu) is not supported"
```

```
ERROR "Unsupported join between segmented table %s and replicated table %s. Table %s is not replicated
on all nodes."
ERROR "Unsupported join between segmented table and unreplicated table"
ERROR "Unsupported join/aggregate two non-alike segmented projections %s and %s"
ERROR "Update is disallowed on Primary/Foreign Keys columns. Use Delete followed by Insert instead"
ERROR "VALINDEX column must be the first column in ORDER BY list"
ERROR "\"E\" is not supported"
ERROR "\"TZ\"/\"tz\" not supported"
ERROR "argument of %s must not contain subqueries"
ERROR "cannot accept a value of type any"
ERROR "cannot accept a value of type anyarray"
ERROR "cannot accept a value of type anyelement"
ERROR "cannot accept a value of type internal"
ERROR "cannot accept a value of type language_handler"
ERROR "cannot accept a value of type opaque"
ERROR "cannot accept a value of type trigger"
ERROR "cannot assign to system column \"%s\""
ERROR "cannot compare rows of zero length"
ERROR "cannot convert relation containing dropped columns to view"
ERROR "cannot delete from a view"
ERROR "cannot display a value of type any"
ERROR "cannot display a value of type anyelement"
ERROR "cannot display a value of type internal"
ERROR "cannot display a value of type language_handler"
ERROR "cannot display a value of type opaque"
ERROR "cannot display a value of type trigger"
ERROR "cannot insert into a view"
ERROR "cannot set a subfield to DEFAULT"
ERROR "cannot set an array element to DEFAULT"
ERROR "cannot update a view"
ERROR "cannot use subquery in EXECUTE parameter"
ERROR "cannot use subquery in SEGMENTED BY expression"
ERROR "command %s is not supported"
ERROR "conditional UNION/INTERSECT/EXCEPT statements are not implemented"
ERROR "cross-database references are not implemented: %s"
ERROR "cross-database references are not implemented: \"%s.%s.%s\""
ERROR "dynamic load not supported"
ERROR "event qualifications are not implemented for rules on SELECT"
ERROR "for SELECT DISTINCT, ORDER BY expressions must appear in select list"
ERROR "input of anonymous composite types is not implemented"
ERROR "interval units \"%s\" not supported"
ERROR "multiple actions for rules on SELECT are not implemented"
ERROR "operator %s is not supported for row expressions"
ERROR "option %s not recognized"
ERROR "replicate_catalog has been shut off"
ERROR "rule actions on NEW are not implemented"
ERROR "rule actions on OLD are not implemented"
ERROR "rules on SELECT must have action INSTEAD SELECT"
ERROR "segmentation expression must have integer type"
ERROR "set-valued function called in context that cannot accept a set"
ERROR "timestamp units \"%s\" not supported"
ERROR "timestamp with time zone units \"%s\" not "
ERROR "timestamp with time zone units \"%s\" not supported"
ERROR "unsupported COPY command clause."
ERROR "unsupported expression in IN clause"
ERROR "vertica does not support GRANT / REVOKE ON FUNCTION"
ERROR "vertica does not support GRANT / REVOKE ON LANGUAGE"
ERROR "vertica does not support GRANT / REVOKE ON TABLESPACE"
FATAL "conversion between %s and %s is not supported"
ROLLBACK "%s not supported"
ROLLBACK "'VALID UNTIL' option is not supported"
ROLLBACK "ADD COLUMN over temporary tables is not supported"
ROLLBACK "ALTER TABLE can specify at most one ADD COLUMN clause"
ROLLBACK "ALTER TABLE cannot specify both ADD COLUMN and ADD CONSTRAINT clauses"
ROLLBACK "CREATEDB option is not supported"
ROLLBACK "CREATEUSER option is not supported"
ROLLBACK "Column %s has the NOT NULL constraint set and has no default value defined"
ROLLBACK "Constraints cannot be altered on tables with projections"
```

```
ROLLBACK "One to one unique joins must be between tables on the same site"
ROLLBACK "Only inner joins are allowed in the projection defining query"
ROLLBACK "Only temporary table's projection can be pinned"
ROLLBACK "Prepared statements are currently unsupported."
ROLLBACK "Site issuing the query cannot be marked as down"
ROLLBACK "Support for UPDATE/DELETE is not enabled"
ROLLBACK "Support for whatever compression you said doesn't exist yet"
ROLLBACK "User groups are not supported"
ROLLBACK "default expression must be a constant"
ROLLBACK "user \"%s\" does not exist"
```

Class 0L Error Code Examples

```
0LV01
ERROR "New %s"
ERROR "grant options can only be granted to users"
ERROR "grant options cannot be granted back to your own grantor"
ERROR "invalid privilege type %s for database"
ERROR "invalid privilege type %s for relation"
ERROR "invalid privilege type %s for schema"
ERROR "invalid privilege type %s for sequence"
```

Class 22 Error Code Examples

```
22000
ERROR "Test Error @%s"
ERROR "Test Error from @%s"
ERROR "invalid Datum pointer"

22001
ERROR "%d-byte value too long for type %s(%d)"
ERROR "date '%s' too long for type %s(%d)"
ERROR "float '%s' too long for type %s(%d)"
ERROR "integer '%s' too long for type %s(%d)"
ERROR "interval '%s' too long for type %s(%d)"
ERROR "padded length (%lld) exceeds the %d byte limit"
ERROR "result (%d characters) exceeds the field width (%d characters)"
ERROR "result exceeds field width"
ERROR "time '%s' too long for type %s(%d)"
ERROR "timestamp '%s' too long for type %s(%d)"
ERROR "timestamptz '%s' too long for type %s(%d)"
ERROR "timetz '%s' too long for type %s(%d)"
ERROR "value too long for type character varying(%d)"
ERROR "value too long for type character(%d)"

22003
ERROR "\"%s\" is out of range for type double precision"
ERROR "int8 out of range"
ERROR "value \"%s\" is out of range for 8-bit integer"
ERROR "value \"%s\" is out of range for type int8"
ERROR "value \"%s\" is out of range for type integer"
ERROR "value \"%s\" is out of range for type smallint"

22004
ERROR "ACL arrays must not contain null values"
ERROR "Cannot set a NOT NULL column to a NULL value in INSERT/UPDATE statement"

22007
ERROR "AM/PM hour must be between 1 and 12"
ERROR "cannot calculate day of year without year information"
ERROR "inconsistent use of year %04lld and \"BC\""
ERROR "invalid AM/PM string"
ERROR "invalid format specification for an interval value"
ERROR "invalid input syntax for type %s: \"%s\""
ERROR "invalid value for %s"
```

```
22008
ERROR "cannot subtract infinite timestamps"
ERROR "date/time field value out of range: \"%s\""
ERROR "interval out of range"
ERROR "timestamp out of range"
ERROR "timestamptz out of range"

22009
ERROR "time zone displacement out of range: \"%s\""

2200B
ERROR "conflicting or redundant options"
22011
ERROR "negative substring length not allowed"

22012
ERROR "division by zero"

22015
ERROR "interval field value out of range: \"%s\""
ERROR "interval is too large (%lld months)"

22019
ERROR "COPY delimiter must be a single character"

22021
ERROR "Unicode characters greater than or equal to 0x10000 are not supported"
ERROR "invalid byte sequence for encoding \"%s\": 0x%s"
22023
ERROR "ACL array contains wrong data type"
ERROR "ACL arrays must be one-dimensional"
ERROR "COPY delimiter must not appear in the NULL specification"
ERROR "Incorrect statement ID for session"
ERROR "NULL string and record_terminator can not be the same value"
ERROR "No running statement, that session is idle"
ERROR "SET %s takes only one argument"
ERROR "Unknown session ID"
ERROR "\"interval\" time zone is too big"
ERROR "\"time with time zone\" units \"%s\" not recognized"
ERROR "\"time\" units \"%s\" not recognized"
ERROR "cannot calculate week number without year information"
ERROR "conflicting \"datestyle\" specifications"
ERROR "could not convert to time zone \"%s\""
ERROR "delimiter and record_terminator can not be the same value"
ERROR "exceptions and rejected_data can not be the same filename"
ERROR "input file and exceptions can not be the same filename."
ERROR "input file and rejected_data can not be the same filename"
ERROR "interval time zone \"%s\" must not specify month"
ERROR "interval time zone must not specify month"
ERROR "interval units \"%s\" not recognized"
ERROR "interval(%d) precision must be between %d and %d"
ERROR "invalid destination encoding name \"%s\""
ERROR "invalid encoding number: %d"
ERROR "invalid interval value for time zone: day not allowed"
ERROR "invalid interval value for time zone: month not allowed"
ERROR "invalid list syntax for parameter \"datestyle\""
ERROR "invalid source encoding name \"%s\""
ERROR "invalid value for parameter \"%s\": \"%s\""
ERROR "time zone \"%s\" appears to use leap seconds"
ERROR "time zone \"%s\" not recognized"
ERROR "timestamp units \"%s\" not recognized"
ERROR "timestamp with time zone units \"%s\" not recognized"
ERROR "timestamp(%d) precision must be between %d and %d"
ERROR "unrecognized \"datestyle\" key word: \"%s\""
ERROR "unrecognized privilege type: \"%s\""
ERROR "unrecognized time zone name: \"%s\""
ERROR "unsupported format code: %d"
FATAL "invalid list syntax for \"listen_addresses\""
```

```
ROLLBACK "%s is a directory."
WARNING "@INCLUDE without filename in time zone file \"%s\", line %d"
WARNING "Could not open %s file, %s is a directory"
WARNING "invalid number for time zone offset in time zone file \"%s\", line %d"
WARNING "invalid syntax in time zone file \"%s\", line %d"
WARNING "invalid time zone file name \"%s\""
WARNING "missing time zone abbreviation in time zone file \"%s\", line %d"
WARNING "missing time zone offset in time zone file \"%s\", line %d"
WARNING "time zone abbreviation \"%s\" is multiply defined"
WARNING "time zone abbreviation \"%s\" is too long (maximum %d characters) in time zone file \"%s\",
line %d"
WARNING "time zone file recursion limit exceeded in file \"%s\""
WARNING "time zone offset %d is not a multiple of 900 sec (15 min) in time zone file \"%s\", line %d"
WARNING "time zone offset %d is out of range in time zone file \"%s\", line %d"
22025
ERROR "invalid escape string"

22V02
ERROR "\"%s\" is not a number"
ERROR "\"\" is not a valid input syntax for type double precision"
ERROR "invalid input syntax for integer: \"%s\""
ERROR "invalid input syntax for type boolean: \"%s\""
ERROR "invalid input syntax for type bytea"
ERROR "invalid input syntax for type double precision: \"%s\""
ERROR "malformed record literal: \"%s\""

220V03
ERROR "incorrect binary data format in bind parameter %d"
22V04
ROLLBACK "COPY from stdin failed: %s"
ROLLBACK "COPY: Input record %lld has been rejected (%s)"

22V05
WARNING "ignoring unconvertible %s character 0x%04x"
WARNING "ignoring unconvertible UTF-8 character 0x%04x"

22V21
ROLLBACK "Can't run historical queries at epochs prior to the Ancient History Mark"
```

Class 26 Error Code Examples

```
26000
ERROR "Cannot issue this command in a read-only transaction"
ERROR "Incorrect number of parameters for prepared statement %s"
ERROR "Prepared statement %s does not exist"
ERROR "Select statement of the insert doesn't have a from clause"
ERROR "unnamed prepared statement does not exist"
```

Class 28 Error Code Examples

```
28000
ERROR "conflicting or redundant options"
ERROR "option \"%s\" not recognized"
FATAL "Invalid username or password"
FATAL "invalid password packet size"
FATAL "no Vertica user name specified in startup packet"
ROLLBACK "conflicting or redundant options"
ROLLBACK "current user cannot be dropped"
ROLLBACK "session user cannot be dropped"
```

Class 42 Error Code Examples

```
42501
ERROR "Insufficient privilege: USAGE on SCHEMA '%s' not granted for current user"
ERROR "must be superuser to COPY to or from a file"
```

```
ERROR "permission denied"
ERROR "permission denied: \"%s\" is a system catalog"
ROLLBACK "must be owner of conversion %s"
ROLLBACK "must be owner of database %s"
ROLLBACK "must be owner of function %s"
ROLLBACK "must be owner of language %s"
ROLLBACK "must be owner of operator %s"
ROLLBACK "must be owner of operator class %s"
ROLLBACK "must be owner of relation %s"
ROLLBACK "must be owner of schema %s"
ROLLBACK "must be owner of sequence %s"
ROLLBACK "must be owner of tablespace %s"
ROLLBACK "must be owner of type %s"
ROLLBACK "must be superuser to create users"
ROLLBACK "must be superuser to drop users"
ROLLBACK "permission denied for conversion %s"
ROLLBACK "permission denied for database %s"
ROLLBACK "permission denied for function %s"
ROLLBACK "permission denied for language %s"
ROLLBACK "permission denied for operator %s"
ROLLBACK "permission denied for operator class %s"
ROLLBACK "permission denied for relation %s"
ROLLBACK "permission denied for schema %s"
ROLLBACK "permission denied for sequence %s"
ROLLBACK "permission denied for tablespace %s"
ROLLBACK "permission denied for type %s"

42601
ERROR "A site name can be specified only once in a create projection, site %s appears
       more than once"
ERROR "All columns in select list must be columns used by projection"
ERROR "Bad epoch range"
ERROR "CREATE TABLE AS specifies too many column names"
ERROR "CREATE VIEW specifies more column "
ERROR "Duplicate columns are not allowed in create table statement"
ERROR "Duplicate columns in select list of projection not allowed"
ERROR "Duplicate tables in projection not allowed"
ERROR "End epoch number out of range"
ERROR "Epoch number out of range"
ERROR "Epoch time out of range"
ERROR "Group by is not allowed in a projection"
ERROR "INSERT ... SELECT may not specify INTO"
ERROR "INSERT ... SELECT may not specify a virtual table (ie %s)"
ERROR "INSERT ... SELECT may not specify a virtual table"
ERROR "INSERT has more expressions than target columns"
ERROR "INSERT has more target columns than expressions"
ERROR "INTO is only allowed on first SELECT of UNION/INTERSECT/EXCEPT"
ERROR "Invalid hint identifier"
ERROR "Invalid predicate in projection-select. Only PK=FK equijoins are allowed."
ERROR "Join in From clause without ON clause is not supported"
ERROR "No columns specified in select list"
ERROR "Not a Star or Snow-Flake Query"
ERROR "Number of columns in the PROJECTION statement must be the same the number of columns in the
       SELECT statement"
ERROR "Only columns are allowed in SELECT list of projection"
ERROR "Only inner joins are allowed in a projection defining query"
ERROR "Only tables are allowed in FROM clause of projection"
ERROR "Projections can only be sorted in ascending order"
ERROR "SELECT * with no tables specified is not valid"
ERROR "SELECT DISTINCT ON is not standard SQL, use just SELECT DISTINCT"
ERROR "Site \"%s\" does not exist"
ERROR "Sort key should be in the target list"
ERROR "Start epoch number out of range"
ERROR "The foreign key in this constraint has already been defined as a foreign key for relation \"%s\"
"
ERROR "Unsupported From clause expression"
ERROR "Unsupported Join in From clause"
ERROR "Unsupported SET option %s"
```

```
ERROR "Unsupported SHOW option %s"
ERROR "Unsupported transaction option %s"
ERROR "Virtual tables are not allowed in FROM clause of projection"
ERROR "\"0\" must be ahead of \"PR\""
ERROR "\"9\" must be ahead of \"PR\""
ERROR "a column definition list is only allowed for functions returning \"record\""
ERROR "a column definition list is required for functions returning \"record\""
ERROR "arguments of row IN must all be row expressions"
ERROR "cannot insert into system column \"%s\""
ERROR "cannot insert multiple commands into a prepared statement"
ERROR "cannot use \"PR\" and \"S\"/\"PL\"/\"MI\"/\"SG\" together"
ERROR "cannot use \"S\" and \"MI\" together"
ERROR "cannot use \"S\" and \"PL\" together"
ERROR "cannot use \"S\" and \"PL\"/\"MI\"/\"SG\"/\"PR\" together"
ERROR "cannot use \"S\" and \"SG\" together"
ERROR "cannot use \"V\" and decimal point together"
ERROR "column alias list for \"%s\" has too many entries"
ERROR "conflicting NULL/NOT NULL declarations for column \"%s\" of table \"%s\""
ERROR "constraint \"%s\" for relation \"%s\" already exists"
ERROR "constraint declared INITIALLY DEFERRED must be DEFERRABLE"
ERROR "each %s query must have the same number of columns"
ERROR "improper %%TYPE reference (too few dotted names): %s"
ERROR "improper %%TYPE reference (too many dotted names): %s"
ERROR "improper qualified name (too many dotted names): %s"
ERROR "improper relation name (too many dotted names): %s"
ERROR "misplaced DEFERRABLE clause"
ERROR "misplaced INITIALLY DEFERRED clause"
ERROR "misplaced INITIALLY IMMEDIATE clause"
ERROR "misplaced NOT DEFERRABLE clause"
ERROR "multiple DEFERRABLE/NOT DEFERRABLE clauses not allowed"
ERROR "multiple INITIALLY IMMEDIATE/DEFERRED clauses not allowed"
ERROR "multiple assignments to same column \"%s\""
ERROR "multiple decimal points"
ERROR "multiple default values specified for column \"%s\" of table \"%s\""
ERROR "non-integer constant in %s"
ERROR "not unique \"S\""
ERROR "schema name may not be qualified"
ERROR "subquery in FROM may not have SELECT INTO"
ERROR "subquery in FROM must have an alias"
ERROR "unequal number of entries in row expression"
ERROR "unequal number of entries in row expressions"
ERROR "wrong number of parameters for prepared statement \"%s\""
ROLLBACK "Add Column driver: Unrecognized command type"
WARNING "Invalid projection name in hint"
WARNING "Invalid site name in hint"

42602
ERROR "invalid name syntax"
ROLLBACK "user ID %llu is already assigned"
ROLLBACK "user \"%s\" already exists"
ROLLBACK "user \"%s\" does not exist"
WARNING "DEPRECATED syntax. Segment expression "

42622
ERROR "encoding name too long"
ERROR "identifier \"%s\" is %d bytes long. Maximum limit is %d bytes."
ROLLBACK "Cannot open FileColumn because path is too long %s"

42701
ERROR "column %s specified more than once"
ERROR "column \"%s\" appears twice in primary key constraint"
ERROR "column \"%s\" appears twice in unique constraint"
ERROR "column \"%s\" specified more than once"
ERROR "column name \"%s\" appears more than once in USING clause"
ROLLBACK "Duplicate column name"
ROLLBACK "Duplicate projection column name (projection: %s)"

42702
ERROR "%s \"%s\" is ambiguous"
```

```
ERROR "column reference \"%s\" is ambiguous"
ERROR "common column name \"%s\" appears more than once in left table"
ERROR "common column name \"%s\" appears more than once in right table"

42703
ERROR "cannot assign to field \"%s\" of column \"%s\" because there is no such column in
       data type %s"
ERROR "cannot assign to system column \"%s\""
ERROR "column %s does not exist"
ERROR "column %s.%s does not exist"
ERROR "column \"%s\" does not exist"
ERROR "column \"%s\" does not exist;\n\tvertica does not support 'SELECT <table_name> FROM
      <table_name>'"
ERROR "column \"%s\" named as primary key does not exist"
ERROR "column \"%s\" named in key does not exist"
ERROR "column \"%s\" not found in data type %s"
ERROR "column \"%s\" of relation \"%s\" does not exist"
ERROR "column \"%s\" specified in USING clause does not exist in left table"
ERROR "column \"%s\" specified in USING clause does not exist in right table"
ERROR "could not identify column \"%s\" in record data type"
ROLLBACK "column %s does not exist in table\n"


42704
ERROR "Node %s does not exist"
ERROR "could not find array type for data type %s"
ERROR "invalid user ID: %llu"
ERROR "no value found for parameter %d"
ERROR "no value found for parameter \"%s\""
ERROR "rule \"%s\" for relation \"%s\" does not exist"
ERROR "type %s is only a shell"
ERROR "type \"%s\" does not exist"
ERROR "type \"%s\" is only a shell"
ERROR "type with OID %llu does not exist"
ERROR "user \"%s\" does not exist"
ROLLBACK "projection \"%s\" does not exist"
ROLLBACK "relation \"%s\" does not exist"
ROLLBACK "site \"%s\" does not exist"42710
ERROR "rule \"%s\" for relation \"%s\" already exists"
ROLLBACK "a table named \"%s\" exists"
ROLLBACK "relation \"%s\" already exists"
ROLLBACK "site \"%s\" already exists"
ROLLBACK "unrecognized drop object type: %d"

42710
ERROR "rule \"%s\" for relation \"%s\" already exists"
ROLLBACK "a table named \"%s\" exists"
ROLLBACK "relation \"%s\" already exists"
ROLLBACK "site \"%s\" already exists"


42712
ERROR "table name \"%s\" specified more than once"

42725
ERROR "function %s is not unique"
ERROR "operator is not unique: %s"

42803
ERROR "SEGMENTED BY expression may not contain aggregate functions"
ERROR "aggregate function calls may not be nested"
ERROR "aggregates not allowed in GROUP BY clause"
ERROR "aggregates not allowed in JOIN conditions"
ERROR "aggregates not allowed in WHERE clause"
ERROR "argument of %s must not contain aggregates"
ERROR "cannot use aggregate function in EXECUTE parameter"
ERROR "cannot use aggregate function in function expression in FROM"
ERROR "column \"%s.%s\" must appear in the GROUP BY clause or be used in an aggregate
```

```
       function"
ERROR "rule WHERE condition may not contain aggregate functions"
ERROR "subquery uses ungrouped column \"%s.%s\" from outer query"

42804
ERROR "%s types %s and %s cannot be matched"
ERROR "IS DISTINCT FROM requires = operator to yield boolean"
ERROR "NULLIF requires = operator to yield boolean"
ERROR "argument of %s must be type boolean, not type %s"
ERROR "argument of %s must be type integer, not type %s"
ERROR "argument of %s must not return a set"
ERROR "arguments declared \"anyelement\" are not all alike"
ERROR "array assignment requires type %s"
ERROR "array assignment to \"%s\" requires type %s"
ERROR "array subscript must have type integer"
ERROR "cannot assign to field \"%s\" of column \"%s\" because its type %s is not a composite
       type"
ERROR "cannot subscript type %s because it is not an array"
ERROR "column \"%s\" is of type %s"
ERROR "could not determine anyarray/anyelement type because input has type \"unknown\""
ERROR "could not determine row description for function returning record"
ERROR "function \"%s\" in FROM has unsupported return type %s"
ERROR "index expression may not return a set"
ERROR "mismatched types in VALUES LESS THAN expressions"
ERROR "no column alias was provided"
ERROR "number of aliases does not match number of columns"
ERROR "parameter $%d of type %s cannot be coerced to the expected type %s"
ERROR "row comparison operator must not return a set"
ERROR "row comparison operator must yield type boolean, "
ERROR "subfield \"%s\" is of type %s"

42809
ERROR "%s(*) specified, but %s is not an aggregate function"
ERROR "DISTINCT specified, but %s is not an aggregate function"
ERROR "\"%s\" is not a projection"
ERROR "column notation .%s applied to type %s, "
ERROR "function %s(%s) is not an aggregate"
ERROR "inherited relation \"%s\" is not a table"
ERROR "op ANY/ALL (array) requires array on right side"
ERROR "op ANY/ALL (array) requires operator not to return a set"
ERROR "op ANY/ALL (array) requires operator to yield boolean"
ERROR "record type has not been registered"
ROLLBACK "COPY requires relation %s to be a Table"
ROLLBACK "COPY requires relation %s to be a Table, not a %s"

42830
ROLLBACK "foreign keys not specified"
ROLLBACK "incompatible data types between primary and foreign key columns: fk: %s, pk: %s"
ROLLBACK "number of primary and foreign keys must be the same"

42846
ERROR "%s could not convert type %s to %s"
ERROR "cannot cast type %s to %s"
ROLLBACK "column \"%s\" is of type %s but default expression is of type %s"

42883
ERROR "Function %s (%llu) is not supported"
ERROR "Meta-function %s (%llu) is not supported with FROM"
ERROR "Operator %s (%llu) is not supported"
ERROR "aggregate %s(%s) does not exist"
ERROR "aggregate %s(*) does not exist"
ERROR "function %s does not exist"
ERROR "function with OID %llu does not exist"
ERROR "no binary input function available for type %s"
ERROR "no binary output function available for type %s"
ERROR "no input function available for type %s"
ERROR "no output function available for type %s"
ERROR "operator does not exist: %s"
ERROR "operator requires run-time type coercion: %s"
```

```
LOG "default conversion function for encoding \"%s\" to \"%s\" does not exist"

42939
ROLLBACK "user name \"%s\" is reserved"

42V01
ERROR "Site \"%s\" does not exist"
ERROR "Table with name '%s' does not exist"
ERROR "missing FROM-clause entry for table \"%s\""
ERROR "missing FROM-clause entry in subquery for table \"%s\""
ERROR "relation \"%s.%s\" does not exist"
ERROR "relation \"%s\" does not exist"
ERROR "relation \"%s\" in FOR UPDATE clause not found in FROM clause"
ERROR "relation with OID %llu does not exist"
ERROR "schema \"%s\" does not exist"
ERROR "site \"%s\" does not exist"
ERROR "table \"%s\" does not exist"
NOTICE "adding missing FROM-clause entry for table \"%s\""
NOTICE "adding missing FROM-clause entry in subquery for table \"%s\""
ROLLBACK "Can't find table"
ROLLBACK "primary table \"%s\" does not exist"
ROLLBACK "table \"%s\" does not exist"


42V02
ERROR "there is no parameter $%d"
42V03
ERROR "cursor \"%s\" already exists"
WARNING "closing existing cursor \"%s\""

42V06
ERROR "schema \"%s\" already exists"
42V07
ERROR "location \"%s\" already exists for site %s"
ROLLBACK "a projection named \"%s\" exists"
ROLLBACK "a table named \"%s\" exists"
ROLLBACK "relation \"%s\" already exists"

42V08
ERROR "could not determine data type of parameter $%d"
ERROR "inconsistent types deduced for parameter

42V09
ERROR "table reference %llu is ambiguous"
ERROR "table reference \"%s\" is ambiguous"

42V10
ERROR "%s position %d is not in select list"
ERROR "JOIN/ON clause refers to \"%s\", which is not part of JOIN"
ERROR "UNION/INTERSECT/EXCEPT member statement may not refer to other relations of same
       query level"
ERROR "argument of %s must not contain variables"
ERROR "function expression in FROM may not refer to other relations of same query level"
ERROR "subquery in FROM may not refer to other relations of same query level"
ERROR "table \"%s\" has %d columns available but %d columns specified"
ERROR "too many column aliases specified for function %s"
42V11
ERROR "cannot specify both SCROLL and NO SCROLL"
42V13
ERROR "aggregates may not return sets"
42V15
ERROR "CREATE specifies a schema (%s) "
ERROR "Insufficient projections to answer query"
ERROR "No super projection found for table %s"

42V16
ERROR "column \"%s\" cannot be declared SETOF"
ERROR "multiple primary keys for table \"%s\" are not allowed"
```

```
ERROR "temporary tables may not specify a schema name"
ROLLBACK "Column \"%s\" from table \"%s\" in the SEGMENTED BY "
ROLLBACK "MATCH types other than SIMPLE (the default) are not supported for foreign
          key constraints"
ROLLBACK "ON DELETE actions other than NO ACTION are not supported for foreign key
          constraints"
ROLLBACK "ON UPDATE actions other than NO ACTION are not supported for foreign key
          constraints"
ROLLBACK "Table changed by another DDL statement"
ROLLBACK "constraint \"%s\" for relation \"%s\" already exists"
ROLLBACK "primary constraint for relation \"%s\" already exists"
ROLLBACK "primary keys not specified"
ROLLBACK "referenced primary key constraint does not exist"

42V17
ERROR "ON DELETE rule may not use NEW"
ERROR "ON INSERT rule may not use OLD"
ERROR "ON SELECT rule may not use NEW"
ERROR "ON SELECT rule may not use OLD"
ERROR "SELECT rule's target entry %d has different column name from \"%s\""
ERROR "SELECT rule's target entry %d has different size from column \"%s\""
ERROR "SELECT rule's target entry %d has different type from column \"%s\""
ERROR "SELECT rule's target list has too few entries"
ERROR "SELECT rule's target list has too many entries"
ERROR "catalog_table requested non-existent object type %s"
ERROR "rule WHERE condition may not contain references to other relations"
ERROR "rules with WHERE conditions may only have SELECT, INSERT, UPDATE, or DELETE actions"
ERROR "view rule for \"%s\" must be named \"%s\""

42V18
ERROR "could not determine data type of parameter $%d"
42V21
ERROR "projection \"%s\" does not exist"
42V22
ERROR "site \"%s\" does not exist"
42V23
ERROR "permutation \"%s\" does not exist"
```

Class 53 Error Code Examples

```
53000
ERROR "Too many ROS containers exist for the "
ROLLBACK "Could not create thread for SubsessionHandler"
ROLLBACK "Could not create thread for recoverProjection"
ROLLBACK "Thread limit %d, but statement needs %lld threads"

53100
ROLLBACK "Could not write to %s: %s"
ROLLBACK "Unable to create catalog file %s"

53200
ERROR "Insufficient resources to execute localized plan [%s]"
ERROR "out of memory"
FATAL "out of memory"
LOG "out of memory"
ROLLBACK "Plan memory limit exhausted: %s"
ROLLBACK "Ran out of WOS memory during %s"
ROLLBACK "malloc of %zu bytes for %s failed"
```

Class 54 Error Code Examples

```
54000
ERROR "%d-byte varchar, oid = %lld"
ERROR "Function %s may give a %d-byte Varchar result; the limit is %d bytes"
ERROR "Unsupported access to virtual table"
ERROR "Unsupported virtual table query. Only a single table reference in FROM clause
```

```
       is allowed."
ERROR "target lists can have at most %d entries"
ERROR "timezone directory stack overflow"
FATAL "out of on_proc_exit slots"
ROLLBACK "Cannot prepare statement - too many prepared statements"
WARNING "line is too long in time zone file \"%s\", line %d"

54011
ERROR "number of columns (%d) exceeds limit (%d)"
ROLLBACK "A table can have at most %d columns"
ROLLBACK "a table/projection can only have up to %d columns -- adding one will exceed
         this limit"
ROLLBACK "a table/projection can only have up to %d columns -- attempt to create one
         with %d\n"
54023
ERROR "cannot pass more than %d arguments to a function"
ERROR "functions cannot have more than %d arguments"
```

Class 55 Error Code Examples

```
55000
ERROR "Cannot issue this command in a read-only transaction"
ERROR "No plan received at node"
ERROR "No transaction running on node"
ERROR "Node has not been set up for plan execution"
ERROR "Node not prepared to accept plan"
ERROR "System is not k-safe. DDL/DML is disallowed"
ERROR "\"%s\" is already a view"
ERROR "could not convert table \"%s\" to a view because it has child tables"
ERROR "could not convert table \"%s\" to a view because it has indexes"
ERROR "could not convert table \"%s\" to a view because it has triggers"
ERROR "could not convert table \"%s\" to a view because it is not empty"
ERROR "cursor can only scan forward"
ERROR "portal \"%s\" cannot be run"
FATAL "data directory \"%s\" has group or world access"
FATAL "data directory \"%s\" has wrong ownership"
ROLLBACK "Cannot Drop: %s %s depends on %s %s"
ROLLBACK "Error: Projection table no longer valid"
ROLLBACK "Query is directly referencing a projection. Unable to retrieve data from requested
projection because one or more sites containing its data are down."
55006
ERROR "Manual analyze statistics not supported"
ERROR "Manual mergeout not supported"
ERROR "Manual moveout not supported"
ERROR "Projection cannot be dropped because K-safety would be violated"
ROLLBACK "A DDL statement interfered with this statement"
ROLLBACK "The status of one or more nodes changed during query planning"

55V03
ERROR "Tuple Mover %s error S locking global catalog"
ERROR "Tuple Mover %s error S locking local catalog"
ERROR "Tuple Mover %s error X locking TMMergeOut lock for moveout"
ROLLBACK "%s error S locking epoch map for installNewCatalog"
ROLLBACK "%s error X locking global catalog for installNewCatalog"
ROLLBACK "%s error X locking local catalog for installNewCatalog"
ROLLBACK "Could not access local catalog due to locking timeout"
ROLLBACK "Could not lock file %s for reading."
ROLLBACK "Could not lock file %s for writing."
ROLLBACK "Error T locking projection anchor table for mergeout"
ROLLBACK "Error T locking projection anchor table for moveout"
ROLLBACK "Error getting epoch map sLock: %s"
ROLLBACK "Error getting table (%llx) sLock: %s"
ROLLBACK "Finalize error (%s) getting S lock on global catalog"
ROLLBACK "Locking failure: %s"
ROLLBACK "Tuple Mover %s error S locking epoch map"
ROLLBACK "Tuple Mover %s error S locking global catalog"
ROLLBACK "Tuple Mover %s error S locking local catalog"
```

```
ROLLBACK "Tuple Mover %s error X locking TMMergeOut lock for mergeout"
ROLLBACK "Tuple Mover %s error X locking local catalog"
ROLLBACK "Tuple Mover %s error locking for mergeout"
ROLLBACK "Tuple Mover %s error locking for moveout"
ROLLBACK "Txn %#llx: %s error %s"
ROLLBACK "Txn %#llx: %s error S locking epoch map for DDL"
ROLLBACK "Txn %#llx: %s error X locking local catalog for DDL"
ROLLBACK "analyze_stats: %s error S locking epoch map for commit"
WARNING "Could not lock file %s for writing."
```

Class 57 Error Code Examples

```
57014
ERROR "Execution canceled (prepare)"
ERROR "Execution canceled (start)"
ERROR "Execution canceled by operator"
ERROR "Execution canceled in EE"
ERROR "Node failure in %s"
ERROR "Operator intervention"
ERROR "Plan canceled prior to execute call"
ERROR "Processing aborted by peer"
ERROR "Statement abandoned due to subsequent DDL"
ERROR "analyze_statistics abandoned due to subsequent DDL"
FATAL "Session canceled by client"
ROLLBACK "Subsession interrupted"
ROLLBACK "Txn %#llx: %s %s"

57V03
FATAL "Shutdown in progress. No longer accepting connections"
FATAL "Site startup/recovery in progress. Not yet ready to accept connections"
ROLLBACK "Session manager cannot add an external session - disabled"
ROLLBACK "Session manager cannot add an internal session - disabled"
```

Class 58 Error Code Examples

```
58030
ERROR "Bad return from WaitForMultipleObjects: %i (%i)"
ERROR "Failed to create socket waiting event: %i"
ERROR "Failed to reset socket waiting event: %i"
FATAL "Failed to load netmsg.dll: %i"
FATAL "failed to enumerate network events: %i"
PANIC "Failure in catalog access; cannot proceed"
PANIC "Failure to roll back transaction; cannot proceed"
PANIC "Failure to roll back transaction; cannot proceed."
ROLLBACK "AddColumn: error writing data file %s"
ROLLBACK "Unable to write catalog file %s"
WARNING "getnameinfo_all() failed: %s"

58V01
ERROR "Invalid filename. Input filename is an empty string"
```

Class V Error Code Examples

```
V1001
ERROR "Connection to spread closed"
ERROR "Receive: Message receipt failed: %s"
ERROR "Receive: Unexpected end of stream: %s"
ERROR "Some nodes did not receive their plans"
ROLLBACK "Receive: open failed on node: %s (%s)"
ROLLBACK "Send: Connection not open [%s tag:%d plan %llu]"
ROLLBACK "Send: Open failed on node [%s] (%s)"

V1002
NOTICE "Cannot shutdown unsafe cluster with this command"
```

```
V1003
ERROR "A node has entered/left the spread group"
ERROR "A node has gone UP/DOWN"

V2001
NOTICE "Vertica license is in its grace period"
WARNING "License issue: %s"

V2002
ROLLBACK "A DDL interfered with moveout"
ROLLBACK "A DDL interfered with recover"
ROLLBACK "A DDL interfered with split"

VC001
FATAL "Cannot load configuration from %s"
FATAL "could not load server certificate file \"%s\": %s"
FATAL "unsafe permissions on private key file \"%s\""
LOG "authentication file token too long, skipping: \"%s\""

VC002
LOG "lock file \"%s\" already exists, %d"
VX001
ERROR "password encryption failed"
FATAL "Unhandled exception during recovery assessment"
FATAL "Unhandled exception during recovery startup assessment"
FATAL "could not get current working directory: %m"
FATAL "failed to create signal event: %d"
FATAL "failed to create signal handler thread"
FATAL "failed to create waitable timer: %i"
FATAL "failed to set console control handler"
FATAL "failed to set waitable timer: %i"
FATAL "findMySession: no session for thread id 0x%llx"
INTERNAL " file %s is not under management"
INTERNAL "AddColumn: internal error writing data file to %s"
INTERNAL "Asked to send %d, but sent %d"
INTERNAL "Attempt to access undefined argument %d"
INTERNAL "Attempt to send distributed calls"
INTERNAL "CALL_DISPATCH_ANY_THREAD is currently unsupported"
INTERNAL "CALL_DISPATCH_SINGLE_THREAD currently requires CAL_RETURN_ASYNCHRONOUS"
INTERNAL "CALL_USE_SESSION_NODES used without setting nodes"
INTERNAL "CALL_USE_SPECIFIED_GROUP requires CALL_RETURN_ASYNCHRONOUS"
INTERNAL "Cannot Begin Transaction when Transaction is already running"
INTERNAL "Caught an exception from EE operator constructor of type %d: %s"
INTERNAL "Caught an unknown exception from EE operator constructor of type %d"
INTERNAL "Caught exception '%s' in dispatchIncomingCallMessage"
INTERNAL "Caught unknown exception in dispatchIncomingCallMessage"
INTERNAL "Compression failed..."
INTERNAL "Corrupt callNodeSelection"
INTERNAL "Couldn't update this session's state"
INTERNAL "DTop: internal error writing data file to %s"
INTERNAL "Did not get the correct sum in "
INTERNAL "DistCalls does not support recursion;"
INTERNAL "EE Block queue corrupted"
INTERNAL "Error creating operator for plan node of type %d: not implemented"
INTERNAL "Error during recover projection"
INTERNAL "Exception decoding the call we just locally encoded"
INTERNAL "Got unexpected error code from spread %d"
INTERNAL "Internal error during data load operation"
INTERNAL "Invalid Execution Point 10"
INTERNAL "Invalid Execution Point 11"
INTERNAL "Invalid Execution Point 12"
INTERNAL "Invalid Execution Point 4"
INTERNAL "Invalid Execution Point 5"
INTERNAL "Invalid Execution Point 6"
INTERNAL "Invalid plan node type for operator DS: expected %d but got %d"
INTERNAL "Join: Invalid phase %d"
INTERNAL "Recover error: recoverProjection"
INTERNAL "Send: cannot execute in undistributed plan %llu"
```

```
INTERNAL "Unable to serialize message;"
INTERNAL "Unknown compression algorithm"
INTERNAL "VEval: unhandled Boolean type %d"
INTERNAL "VEval: unhandled boolean test type %d"
INTERNAL "VEval: unhandled evaluateExpr oid %u"
INTERNAL "VEval: unhandled null check data type %d"
INTERNAL "VEval::VEval unhandled expression type %d"
INTERNAL "aggregate function %llu called as normal function"
INTERNAL "bogus ContainsOids value: %d"
INTERNAL "bogus InhOption value: %d"
INTERNAL "bogus resno %d in targetlist"
INTERNAL "can't happen"
INTERNAL "compile plan already compiled."
INTERNAL "compile plan node of type %d (operator %s) has NULL dest on output edge %d"
INTERNAL "compile plan node of type %d (operator %s) has NULL source on input edge %d."
INTERNAL "compile plan not yet compiled."
INTERNAL "could not create signal listener pipe for pid %d: error code %d"
INTERNAL "expected SELECT query from subquery in FROM"
INTERNAL "getMySessionID: no session for thread id 0x%llx"
INTERNAL "invalid ObjectType"
INTERNAL "invalid datetoken tables, please fix %s"
INTERNAL "invalid return code %d from operator %s"
INTERNAL "scalar array op %s (%llu) is not supported"
INTERNAL "sendCallToOne applies only to calls sent to specified nodes"
INTERNAL "too many arguments"
INTERNAL "unexpected parse analysis result for subquery in FROM"
INTERNAL "unhandled AclObjectKind value"
INTERNAL "unhandled AclResult value"
INTERNAL "unhandled GrantObjectType value"
INTERNAL "unrecognized GrantStmt.objtype: %d"
INTERNAL "unrecognized join type: %d"
INTERNAL "unrecognized node type: %d"
INTERNAL "unrecognized object kind: %d"
INTERNAL "unrecognized objkind: %d"
INTERNAL "unrecognized portal strategy: %d"
INTERNAL "unrecognized sortby_kind: %d"
INTERNAL "updateCheckpointEpoch called without a transaction"
NOTICE "Unknown win32 socket error code: %i"
PANIC "Message could not be deserialized: %s"
PANIC "Redundant bind of conflicting transaction "
PANIC "Unbind of conflicting transaction "
WARNING "Exception decoding the response we just locally encoded"

VX002
ROLLBACK "Delete: could not find a data row to delete (data integrity violation?)"
ROLLBACK "Error finalizing AddColumn"
ROLLBACK "Error finalizing DT; column data may be lost."
ROLLBACK "FileColumnReader: Decompression error in %s at offset %llu"
```

Index

D

E

F

G

H

I

Q

R

S

T

U

V

W

X

Z

Copyright Notice

Vertica, An HP Company
8 Federal Street
Billerica, MA 01821
Phone: (978) 600-1000
Fax: (978) 600-1001
E-Mail: info@vertica.com
Web site: http://www.vertica.com
(***http://www.vertica.com***)

Trademarks

Open Source Software Acknowledgments

Vertica makes no representations or warranties regarding any third party software. All third-party software is provided or recommended by Vertica on an AS IS basis.

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

ASMJIT

Copyright (c) 2008-2010, Petr Kobalicek <kobalicek.petr@gmail.com>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Boost

Boost Software License - Version 1.38 - February 8th, 2009

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

bzip2

This file is a part of bzip2 and/or libbzip2, a program and library for lossless, block-sorting data compression.

Daemonize

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the clapper.org nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Ganglia Open Source License

Copyright © 2001 by Matt Massie and The Regents of the University of California.

All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without written agreement is hereby granted, provided that the above copyright notice and the following two paragraphs appear in all copies of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

ICU (International Components for Unicode) License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright © 1995-2009 International Business Machines Corporation and others

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

All trademarks and registered trademarks mentioned herein are the property of their respective owners.

jQuery

Copyright © 2009 John Resig, http://jquery.com/

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Keepalived Vertica IPVS (IP Virtual Server) Load Balancer

Copyright © 2007 Free Software Foundation, Inc.

http://fsf.org/

The keepalived software contained in the `VerticaIPVSLoadBalancer-5.0.x-0.RHEL5.x86_64.rpm` software package is licensed under the GNU General Public License ("GPL"). You are entitled to receive the source code for such software. For no less than three years from the date you obtained this software package, you may download a copy of the source code for the software in this package licensed under the GPL at no charge by visiting http://www.vertica.com/licenses/keepalived-1.1.17.tar.gz ***http://www.vertica.com/licenses/keepalived-1.1.17.tar.gz***. You may download this source code so that it remains separate from other software on your computer system.

Lighttpd Open Source License

Copyright © 2004, Jan Kneschke, incremental

All rights reserved.

1. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
2. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
3. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
4. Neither the name of the 'incremental' nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

MersenneTwister.h

Copyright © 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,

Copyright © 2000 - 2009, Richard J. Wagner

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

MIT Kerberos

Copyright © 1985-2007 by the Massachusetts Institute of Technology.

Export of software employing encryption from the United States of America may require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting.

WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Furthermore if you modify this software you must label your software as modified software and not distribute it in such a fashion that it might be confused with the original MIT software. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

Individual source code files are copyright MIT, Cygnus Support, Novell, OpenVision Technologies, Oracle, Red Hat, Sun Microsystems, FundsXpress, and others.

Project Athena, Athena, Athena MUSE, Discuss, Hesiod, Kerberos, Moira, and Zephyr are trademarks of the Massachusetts Institute of Technology (MIT). No commercial use of these trademarks may be made without prior written permission of MIT.

“Commercial use” means use of a name in a product or other for-profit manner. It does NOT prevent a commercial firm from referring to the MIT trademarks in order to convey information (although in doing so, recognition of their trademark status should be given).

Portions of src/lib/crypto have the following copyright:

Copyright © 1998 by the FundsXpress, INC.

All rights reserved.

Export of this software from the United States of America may require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting.

WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of FundsXpress. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. FundsXpress makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

THIS SOFTWARE IS PROVIDED “AS IS” AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The implementation of the AES encryption algorithm in src/lib/crypto/aes has the following copyright:

> Copyright © 2001, Dr Brian Gladman <brg@gladman.uk.net>, Worcester, UK.
> All rights reserved.

LICENSE TERMS

The free distribution and use of this software in both source and binary form is allowed (with or without changes) provided that:

1. Distributions of this source code include the above copyright notice, this list of conditions and the following disclaimer.
2. Distributions in binary form include the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other associated materials.
3. The copyright holder's name is not used to endorse products built using this software without specific written permission.

DISCLAIMER

This software is provided 'as is' with no explicit or implied warranties in respect of any properties, including, but not limited to, correctness and fitness for purpose.

The implementations of GSSAPI mechglue in GSSAPI-SPNEGO in src/lib/gssapi, including the following files:

- lib/gssapi/generic/gssapi_err_generic.et
- lib/gssapi/mechglue/g_accept_sec_context.c
- lib/gssapi/mechglue/g_acquire_cred.c
- lib/gssapi/mechglue/g_canon_name.c
- lib/gssapi/mechglue/g_compare_name.c
- lib/gssapi/mechglue/g_context_time.c
- lib/gssapi/mechglue/g_delete_sec_context.c
- lib/gssapi/mechglue/g_dsp_name.c
- lib/gssapi/mechglue/g_dsp_status.c
- lib/gssapi/mechglue/g_dup_name.c
- lib/gssapi/mechglue/g_exp_sec_context.c
- lib/gssapi/mechglue/g_export_name.c
- lib/gssapi/mechglue/g_glue.c
- lib/gssapi/mechglue/g_imp_name.c

- lib/gssapi/mechglue/g_imp_sec_context.c
- lib/gssapi/mechglue/g_init_sec_context.c
- lib/gssapi/mechglue/g_initialize.c
- lib/gssapi/mechglue/g_inquire_context.c
- lib/gssapi/mechglue/g_inquire_cred.c
- lib/gssapi/mechglue/g_inquire_names.c
- lib/gssapi/mechglue/g_process_context.c
- lib/gssapi/mechglue/g_rel_buffer.c
- lib/gssapi/mechglue/g_rel_cred.c
- lib/gssapi/mechglue/g_rel_name.c
- lib/gssapi/mechglue/g_rel_oid_set.c
- lib/gssapi/mechglue/g_seal.c
- lib/gssapi/mechglue/g_sign.c
- lib/gssapi/mechglue/g_store_cred.c
- lib/gssapi/mechglue/g_unseal.c
- lib/gssapi/mechglue/g_userok.c
- lib/gssapi/mechglue/g_utils.c
- lib/gssapi/mechglue/g_verify.c
- lib/gssapi/mechglue/gssd_pname_to_uid.c
- lib/gssapi/mechglue/mglueP.h
- lib/gssapi/mechglue/oid_ops.c
- lib/gssapi/spnego/gssapiP_spnego.h
- lib/gssapi/spnego/spnego_mech.c

are subject to the following license:

Copyright © 2004 Sun Microsystems, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Npgsql-.Net Data Provider for Postgresql

Copyright © 2002-2008, The Npgsql Development Team

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE NPGSQL DEVELOPMENT TEAM BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE NPGSQL DEVELOPMENT TEAM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE NPGSQL DEVELOPMENT TEAM SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE NPGSQL DEVELOPMENT TEAM HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Open LDAP

The OpenLDAP Public License

Version 2.8, 17 August 2003

Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met:

1 Redistributions in source form must retain copyright statements and notices,

2 Redistributions in binary form must reproduce applicable copyright statements and notices, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution, and

3 Redistributions must contain a verbatim copy of this document.

The OpenLDAP Foundation may revise this license from time to time. Each revision is distinguished by a version number. You may use this Software under terms of this license revision or under the terms of any subsequent revision of the license.

THIS SOFTWARE IS PROVIDED BY THE OPENLDAP FOUNDATION AND ITS CONTRIBUTORS ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OPENLDAP FOUNDATION, ITS CONTRIBUTORS, OR THE AUTHOR(S) OR OWNER(S) OF THE SOFTWARE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The names of the authors and copyright holders must not be used in advertising or otherwise to promote the sale, use or other dealing in this Software without specific, written prior permission. Title to copyright in this Software shall at all times remain with copyright holders.

OpenLDAP is a registered trademark of the OpenLDAP Foundation.

Open SSL

OpenSSL License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.
5. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.
6. Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

PCRE LICENCE

PCRE is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.

Release 8 of PCRE is distributed under the terms of the "BSD" licence, as specified below. The documentation for PCRE, supplied in the "doc" directory, is distributed under the same terms as the software itself.

The basic library functions are written in C and are freestanding. Also included in the distribution is a set of C++ wrapper functions.

Perl Artistic License

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

"Package" refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

"Standard Version" refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

"Copyright Holder" is whoever is named in the copyright or copyrights for the package.

"You" is you, if you're thinking about copying or distributing this Package.

"Reasonable copying fee" is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

"Freely Available" means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

1 You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2 You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.

3 You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

4 place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.

1. use the modified Package only within your corporation or organization.
2. rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.
3. make other distribution arrangements with the Copyright Holder.

5 You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:

1. distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.
2. accompany the distribution with the machine-readable source of the Package with your modifications.

3. give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.
4. make other distribution arrangements with the Copyright Holder.

6 You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.

7 The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called "undump" or "unexec" methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.

8 C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.

9 Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.

10 The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The End

Pexpect

Copyright © 2010 Noah Spurrier

Credits: Noah Spurrier, Richard Holden, Marco Molteni, Kimberley Burchett, Robert Stone, Hartmut Goebel, Chad Schroeder, Erick Tryzelaar, Dave Kirby, Ids vander Molen, George Todd, Noel Taylor, Nicolas D. Cesar, Alexander Gattin, Geoffrey Marshall, Francisco Lourenco, Glen Mabey, Karthik Gurusamy, Fernando Perez, Corey Minyard, Jon Cohen, Guillaume Chazarain, Andrew Ryan, Nick Craig-Wood, Andrew Stone, Jorgen Grahn (Let me know if I forgot anyone.)

Free, open source, and all that good stuff.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

PHP License

The PHP License, version 3.01

Copyright © 1999 - 2009 The PHP Group. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, is permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name "PHP" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact group@php.net.
4. Products derived from this software may not be called "PHP", nor may "PHP" appear in their name, without prior written permission from group@php.net. You may indicate that your software works in conjunction with PHP by saying "Foo for PHP" instead of calling it "PHP Foo" or "phpfoo"
5. The PHP Group may publish revised and/or new versions of the license from time to time. Each version will be given a distinguishing version number.

- Once covered code has been published under a particular version of the license, you may always continue to use it under the terms of that version. You may also choose to use such covered code under the terms of any subsequent version of the license published by the PHP Group. No one other than the PHP Group has the right to modify the terms applicable to covered code created under this License.

6. Redistributions of any form whatsoever must retain the following acknowledgment:
"This product includes PHP software, freely available from <http://www.php.net/software/>".

THIS SOFTWARE IS PROVIDED BY THE PHP DEVELOPMENT TEAM ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PHP DEVELOPMENT TEAM OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the PHP Group.

The PHP Group can be contacted via Email at group@php.net.

For more information on the PHP Group and the PHP project, please see <http://www.php.net>.

PHP includes the Zend Engine, freely available at <http://www.zend.com>.

PostgreSQL

This product uses the PostgreSQL Database Management System(formerly known as Postgres, then as Postgres95)

Portions Copyright © 1996-2005, The PostgreSQL Global Development Group

Portions Copyright © 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Psqlodbc

The client drivers for Vertica Analytic Database use psqlodbc library, the Postgresql ODBC driver.

License:

This package is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This package is distributed in the hope that it is useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this package; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

The complete source code of the library and complete text of the GNU Lesser General Public License can be obtained by contacting Vertica support.

Python 2.7

This is the official license for the Python 2.7 release:

A. HISTORY OF THE SOFTWARE

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation, see http://www.zope.com). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL-compatible? (1)
0.9.0 thru 1.2		1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes

2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2002-2003	PSF	yes
2.3.3	2.3.2	2002-2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes
2.4.2	2.4.1	2005	PSF	yes
2.4.3	2.4.2	2006	PSF	yes
2.5	2.4	2006	PSF	yes
2.7	2.6	2010	PSF	yes

Footnotes:

1 GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

2 According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is "not incompatible" with the GPL.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

B. TERMS AND CONDITIONS FOR ACCESSING OR OTHERWISE USING PYTHON

PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

---

1 This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using this software ("Python") in source or binary form and its associated documentation.

2 Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006 Python Software Foundation; All Rights Reserved" are retained in Python alone or in any derivative version prepared by Licensee.

3 In the event Licensee prepares a derivative work that is based on or incorporates Python or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python.

4 PSF is making Python available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5 PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6 This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7 Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8 By copying, installing or otherwise using Python, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1 This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2 Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3 BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4 BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF

5 This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6 This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7 By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

---

1 This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2 Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013".

3 In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4 CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5 CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6 This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7 This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8 By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright (c) 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Python Dialog

The Administration Tools part of this product uses Python Dialog, a Python module for doing console-mode user interaction.

Upstream Author:

Peter Astrand <peter@cendio.se>

Robb Shecter <robb@acm.org>

Sultanbek Tezadov

Florent Rougon <flo@via.ecp.fr>

Copyright © 2000 Robb Shecter, Sultanbek Tezadov

Rsync

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

RRDTool Open Source License

Note: rrdtool is a dependency of using the ganglia-web third-party tool. RRDTool allows the graphs displayed by ganglia-web to be produced.

RRDTOOL - Round Robin Database Tool

A tool for fast logging of numerical data graphical display of this data.

GNU GPL License

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

FLOSS License Exception

(Adapted from http://www.mysql.com/company/legal/licensing/foss-exception.html)

I want specified Free/Libre and Open Source Software ("FLOSS") applications to be able to use specified GPL-licensed RRDtool libraries (the "Program") despite the fact that not all FLOSS licenses are compatible with version 2 of the GNU General Public License (the "GPL").

As a special exception to the terms and conditions of version 2.0 of the GPL:

You are free to distribute a Derivative Work that is formed entirely from the Program and one or more works (each, a "FLOSS Work") licensed under one or more of the licenses listed below, as long as:

1. You obey the GPL in all respects for the Program and the Derivative Work, except for identifiable sections of the Derivative Work which are not derived from the Program, and which can reasonably be considered independent and separate works in themselves
2. All identifiable sections of the Derivative Work which are not derived from the Program, and which can reasonably be considered independent and separate works in themselves

- are distributed subject to one of the FLOSS licenses listed below, and
- the object code or executable form of those sections are accompanied by the complete corresponding machine-readable source code for those sections on the same medium and under the same FLOSS license as the corresponding object code or executable forms of those sections.

3 Any works which are aggregated with the Program or with a Derivative Work on a volume of a storage or distribution medium in accordance with the GPL, can reasonably be considered independent and separate works in themselves which are not derivatives of either the Program, a Derivative Work or a FLOSS Work.

If the above conditions are not met, then the Program may only be copied, modified, distributed or used under the terms and conditions of the GPL.

FLOSS License List

License name	Version(s)/Copyright Date
Academic Free License	2.0
Apache Software License	1.0/1.1/2.0
Apple Public Source License	2.0
Artistic license	From Perl 5.8.0
BSD license	"July 22 1999"
Common Public License	1.0
GNU Library or "Lesser" General Public License (LGPL)	2.0/2.1
IBM Public License, Version	1.0
Jabber Open Source License	1.0
MIT License (As listed in file MIT-License.txt)	-
Mozilla Public License (MPL)	1.0/1.1
Open Software License	2.0
OpenSSL license (with original SSLeay license)	"2003" ("1998")
PHP License	3.0
Python license (CNRI Python License)	-
Python Software Foundation License	2.1.1
Sleepycat License	"1999"
W3C License	"2001"
X11 License	"2001"
Zlib/libpng License	-
Zope Public License	2.0/2.1

Spread

This product uses software developed by Spread Concepts LLC for use in the Spread toolkit. For more information about Spread see http://www.spread.org (***http://www.spread.org***).

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer and request.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer and request in the documentation and/or other materials provided with the distribution.
3. All advertising materials (including web pages) mentioning features or use of this software, or software that uses this software, must display the following acknowledgment: "This product uses software developed by Spread Concepts LLC for use in the Spread toolkit. For more information about Spread see http://www.spread.org"
4. The names "Spread" or "Spread toolkit" must not be used to endorse or promote products derived from this software without prior written permission.
5. Redistributions of any form whatsoever must retain the following acknowledgment:
6. "This product uses software developed by Spread Concepts LLC for use in the Spread toolkit. For more information about Spread, see http://www.spread.org"
7. This license shall be governed by and construed and enforced in accordance with the laws of the State of Maryland, without reference to its conflicts of law provisions. The exclusive jurisdiction and venue for all legal actions relating to this license shall be in courts of competent subject matter jurisdiction located in the State of Maryland.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, SPREAD IS PROVIDED UNDER THIS LICENSE ON AN AS IS BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT SPREAD IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. ALL WARRANTIES ARE DISCLAIMED AND THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE CODE IS WITH YOU. SHOULD ANY CODE PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE COPYRIGHT HOLDER OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY CODE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY OTHER CONTRIBUTOR BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES FOR LOSS OF PROFITS, REVENUE, OR FOR LOSS OF INFORMATION OR ANY OTHER LOSS.

YOU EXPRESSLY AGREE TO FOREVER INDEMNIFY, DEFEND AND HOLD HARMLESS THE COPYRIGHT HOLDERS AND CONTRIBUTORS OF SPREAD AGAINST ALL CLAIMS, DEMANDS, SUITS OR OTHER ACTIONS ARISING DIRECTLY OR INDIRECTLY FROM YOUR ACCEPTANCE AND USE OF SPREAD.

Although NOT REQUIRED, we at Spread Concepts would appreciate it if active users of Spread put a link on their web site to Spread's web site when possible. We also encourage users to let us know who they are, how they are using Spread, and any comments they have through either e-mail (spread@spread.org) or our web site at (http://www.spread.org/comments).

SNMP

Various copyrights apply to this package, listed in various separate parts below. Please make sure that you read all the parts. Up until 2001, the project was based at UC Davis, and the first part covers all code written during this time. From 2001 onwards, the project has been based at SourceForge, and Networks Associates Technology, Inc hold the copyright on behalf of the wider Net-SNMP community, covering all derivative work done since then. An additional copyright section has been added as Part 3 below also under a BSD license for the work contributed by Cambridge Broadband Ltd. to the project since 2001. An additional copyright section has been added as Part 4 below also under a BSD license for the work contributed by Sun Microsystems, Inc. to the project since 2003.

Code has been contributed to this project by many people over the years it has been in development, and a full list of contributors can be found in the README file under the THANKS section.

Part 1: CMU/UCD copyright notice: (BSD like)

> Copyright © 1989, 1991, 1992 by Carnegie Mellon University
>
> Derivative Work - 1996, 1998-2000
>
> Copyright © 1996, 1998-2000 The Regents of the University of California
>
> All Rights Reserved

Permission to use, copy, modify and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of CMU and The Regents of the University of California not be used in advertising or publicity pertaining to distribution of the software without specific written permission.

CMU AND THE REGENTS OF THE UNIVERSITY OF CALIFORNIA DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL CMU OR THE REGENTS OF THE UNIVERSITY OF CALIFORNIA BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM THE LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Part 2: Networks Associates Technology, Inc copyright notice (BSD)

> Copyright © 2001-2003, Networks Associates Technology, Inc
>
> All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the Networks Associates Technology, Inc nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 3: Cambridge Broadband Ltd. copyright notice (BSD)

Portions of this code are copyright (c) 2001-2003, Cambridge Broadband Ltd.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of Cambridge Broadband Ltd. may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 4: Sun Microsystems, Inc. copyright notice (BSD)

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara,

California 95054, U.S.A. All rights reserved.

Use is subject to license terms below.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Sun Microsystems, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 5: Sparta, Inc copyright notice (BSD)

Copyright © 2003-2006, Sparta, Inc

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Sparta, Inc nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 6: Cisco/BUPTNIC copyright notice (BSD)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Cisco, Inc, Beijing University of Posts and Telecommunications, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 7: Fabasoft R&D Software GmbH & Co KG copyright notice (BSD)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of Fabasoft R&D Software GmbH & Co KG or any of its subsidiaries, brand or product names may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE

LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Tecla Command-line Editing

Copyright © 2000 by Martin C. Shepherd.

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

Webmin Open Source License

Copyright © Jamie Cameron

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the developer nor the names of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE DEVELOPER ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE DEVELOPER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

xerces

NOTICE file corresponding to section 4(d) of the Apache License,

Version 2.0, in this case for the Apache Xerces distribution.

This product includes software developed by The Apache Software Foundation (http://www.apache.org/).

Portions of this software were originally based on the following:

Software copyright © 1999, IBM Corporation., http://www.ibm.com.

zlib

This is used by the project to load zipped files directly by COPY command. www.zlib.net/

zlib.h -- interface of the 'zlib' general purpose compression library version 1.2.3, July 18th, 2005

Copyright © 1995-2005 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly jloup@gzip.org

Mark Adler madler@alumni.caltech.edu