Vertica Database 2.0.5-0

SQL Reference Manual

Copyright© 2006, 2007 Vertica Systems, Inc.

Date of Publication: 1/8/2008

CONFIDENTIAL

Copyright Notice

Copyright© 2006 - 2007 Vertica Systems, Inc. and its licensors. All rights reserved.

Vertica Systems, Inc.

Three Dundee Park Drive, Suite 102

Andover, MA 01810-3723

Phone: (978) 475-1070

Fax: (978) 475-6855

E-Mail: info@vertica.com

Web site: http://www.vertica.com
(http://www.vertica.com)

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license. Vertica Systems, Inc. software contains proprietary information, as well as trade secrets of Vertica Systems, Inc., and is protected under international copyright law. Reproduction, adaptation, or translation, in whole or in part, by any means — graphic, electronic or mechanical, including photocopying, recording, taping, or storage in an information retrieval system — of any part of this work covered by copyright is prohibited without prior written permission of the copyright owner, except as allowed under the copyright laws.

This product or products depicted herein may be protected by one or more U.S. or international patents or pending patents.

Trademarks

Vertica™ and the Vertica Database™ are trademarks of Vertica Systems, Inc.

Adobe®, Acrobat®, and Acrobat® Reader® are registered trademarks of Adobe Systems Incorporated.

AMD™ is a trademark of Advanced Micro Devices, Inc. in the United States and other countries.

Fedora™ is a trademark of Red Hat, Inc.

Intel® is a registered trademark of Intel.

Linux® is a registered trademark of Linus Torvalds.

Microsoft® is a registered trademark of Microsoft Corporation.

Novell® is a registered trademark and SUSE™ is a trademark of Novell, Inc. in the United States and other countries.

Oracle® is a registered trademark of Oracle Corporation.

Red Hat® is a registered trademark of Red Hat, Inc.

VMware® is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions.

Other products mentioned may be trademarks or registered trademarks of their respective companies.

Open Source Software Acknowledgements

Boost

Boost Software License - Version 1.0 - August 17th, 2003

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.

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.

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 <http://sultan.da.ru>

Florent Rougon <flo@via.ecp.fr>

Copyright © 2000 Robb Shecter, Sultanbek Tezadov

Copyright © 2002, 2003, 2004 Florent Rougon

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 will be 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

On Vertica systems, complete source code of the Python dialog package and complete text of the GNU Lesser General Public License can be found on the Vertica Systems website at http://www.vertica.com/licenses/pythondialog-2.7.tar.bz2 http://www.vertica.com/licenses/pythondialog-2.7.tar.bz2

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).

Copyright (c) 1993-2006 Spread Concepts LLC. 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 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:

"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"

6. 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).

Contents

Technical Support

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

http://www.vertica.com/support (http://www.vertica.com/support)

You must be a registered user in order to access the page.

Before reporting a problem, please run the Diagnostics Utility described in the Troubleshooting Guide and attach the resulting .zip file.

About the Documentation

Where to Find the Vertica Documentation

Vertica Systems, Inc. recommends that you copy the Vertica documentation from the database server (any cluster host) to a client system on which you can use a browser and/or the Adobe Reader.

Database Server Systems

The Vertica Database Documentation Set is automatically installed in the **/opt/vertica/doc/** directory on all cluster hosts. If you have a browser and/or the Adobe Reader installed on a cluster host, you can access the documentation directly.

`/opt/vertica/doc/HTML/Master/index.htm`
`/opt/vertica/doc/JDBC/index.htm`
`/opt/vertica/doc/PDF/`*`book-name`*`.pdf`

Database Client Systems

To create a copy of the Vertica documentation on a client system, do one of the following:

- Download the documentation package (**.tar.gz** or **.zip**) from the Vertica Systems, Inc. Web site and extract the files to a directory on the client system, using the original pathnames
- Copy the documentation directories from a database server system to a convenient location in your client system. All cross-references within the HTML documentation are relative so there is no location dependency.

`<`*`your-location`*`>HTML/Master/index.htm`
`<`*`your-location`*`>JDBC/index.htm`
`<`*`your-location`*`>PDF/`*`book-name`*`.htm`

World Wide Web

You can read and/or download the Vertica documentation from the **Vertica Systems, Inc. Web site**:

http://www.vertica.com/v-zone/product_documentation http://www.vertica.com/v-zone/product_documentation.

You need a V-Zone login to access the Product Documentation page.

The documentation on the Vertica Systems, Inc. Web site is updated each time a new release is issued. If you are using an older version, refer to the documentation on your database server or client systems.

Reading the HTML Files

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

The Vertica documentation has been tested on the following browsers:

- Internet Explorer 7
- FireFox
- Opera
- Safari

Please report any script, image rendering, or text formatting problems to ***Technical Support*** (on page 13).

The Vertica documentation may contain links to Web sites of other companies or organizations that Vertica does not own or control. If any of these links are broken, please inform us.

Printing the PDF Files

The documentation files are supplied in **Adobe Acrobat™ PDF** document format for the purpose of making printed copies as needed. The documents are designed to be printed on standard 8½ x 11 paper using full duplex (two sided printing).

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

HTML links to the PDF files are provided here for browser access.

- Database Administrator's Guide
- Database Administrator's Guide (Advanced)
- Glossary of Terms
- Installation Guide
- Product Overview
- Quick Start
- Release Notes
- SQL Programmer's Guide
- SQL Reference Manual
- Troubleshooting Guide

Suggested Reading Paths

This section provides a suggested reading path for various types of users. Read the manuals listed under All Users first.

All Users

- Product Overview (basic concepts critical to understanding Vertica)
- Quick Start (step-by-step guide to getting Vertica up and running)
- Glossary of Terms (glossary of terms)

System Administrators

- Installation Guide (platform configuration and software installation)
- Release Notes (release-specific information)
- Troubleshooting Guide (general troubleshooting information)

Database Administrators

- Installation Guide (platform configuration and software installation)
- Database Administrator's Guide (database configuration, loading, security, and maintenance)
- Troubleshooting Guide (general troubleshooting information)

Application Developers

- SQL Programmer's Guide (connecting to a database, queries, transactions, etc.)
- SQL Reference Manual (Vertica-specific language information)
- Troubleshooting Guide (general troubleshooting information)

Where to Find Additional Information

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

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

Typographical Conventions

It is important to understand the terms and typographical conventions used in this document.

General Convention	Description
colored bold text	introduces new terms defined either in the text, the glossary, or both.
normal italic text	indicates emphasis and the titles of other documents.
UPPERCASE TEXT	indicates the name of an SQL command or keyword.
`monospace text`	indicates literal interactive or programmatic input/output.
`italic monospace text`	indicates user-supplied information in interactive or programmatic input/output.
`bold monospace text`	indicates literal interactive user input
↵	indicates the Return/Enter key; implicit on all user input that includes text

SQL Syntax Convention	Description
indentation	is an attempt to maximize readability; SQL is a free-form language.
backslash \	continuation character used to indicate text that is too long to fit on a single line.
braces { }	indicate required items.
brackets [ ]	indicate optional items.
ellipses ...	indicate an optional sequence of similar items.
vertical ellipses ⋮	indicate an optional sequence of similar items or that part of the text has been omitted for readability.
vertical line |	within braces or brackets, indicates a choice .

Preface

This document provides a reference description of the Vertica SQL database language.

Audience

This document is intended for anyone who uses Vertica. It assumes that you are familiar with the basic concepts and terminology of the SQL language and relational database management systems.

SQL Overview

SQL (Structured Query Language) is a widely-used, industry standard data definition and data manipulation language for relational databases.

Vertica Support for ANSI SQL Standards

Vertica SQL supports a subset of **ANSI SQL 99**. Over time, Vertica SQL will enlarge and eventually converge with ANSI SQL 99. For information about ANSI SQL 99 see:

- ***BNF Grammar for SQL-99*** (http://savage.net.au/SQL/sql-99.bnf.html)
- ***Index of /education/modules/dbms/SQL99***
  (http://www.ncb.ernet.in/education/modules/dbms/SQL99/)

Vertica Use of PostgreSQL

In addition to using vsql as an interactive front-end, Vertica SQL supports a subset of the PostgreSQL language definition. For information about PostgreSQL see:

- ***PostgreSQL 8.0.12 Documentation*** (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

Vertica Major Extensions to SQL

Vertica provides several extensions to SQL that allow you to use the unique aspects of its column store architecture:

- **AT EPOCH LATEST SELECT..**.

  runs a SQL query in snapshot isolation mode in which it does not hold locks or block other processes such as data loads.
- **AT TIME 'timestamp' SELECT...**

  runs historical queries against a snapshot of the database a specific date and time.
- **CONSTRAINT ... CORRELATION (column) REFERENCES (column)**

  captures Functional Dependencies that can be used by the Database Designer to produce more efficient projections.
- **COPY**

  is used for bulk loading data. It reads data from a text file and inserts tuples into the WOS (Write Optimized Store) or directly into the ROS (Read Optimized Store).
- **CREATE/DROP/ALTER PROJECTION**

  are used for manipulating projections as described in the Database Administrator's Guide (Advanced). CREATE PROJECTION commands are generated for you by the Database Designer as described in the Database Administrator's Guide.
- **SELECT Vertica Function**

executes special Vertica functions.

- **SET DATESTYLE**

  chooses the format in which date/time values are displayed.

- **SET SEARCH_PATH**

  specifies the order in which Vertica searches through multiple schemas when a SQL statement contains an unqualified table name.

- **SET TIMEZONE**

  specifies the TIMEZONE run-time parameter for the current session.

- **SHOW**

  displays run-time parameters for the current session.

Support for Historical Queries

Unlike most databases, the ***DELETE*** (page 158) command in Vertica does not actually delete data; it simply marks tuples as deleted. The ***UPDATE*** (page 190) command actually does an INSERT and a DELETE. This behavior is necessary for historical queries. You can control how much historical data is stored on disk.

Non-Standard Syntax and Semantics

In case of non-standard SQL syntax or semantics, Vertica SQL follows Oracle whenever possible. For Oracle SQL documentation, you will need a web account to access the library:

- ***Oracle Database 10g Documentation Library***
  (http://www.oracle.com/pls/db102/homepage)

Joins

Vertica supports only:

- standard inner equi-joins in the WHERE clause

primary-key/foreign-key (1:n) star and snowflake joins Vertica does not support:

- self-joins (use temporary tables for this purpose)
- outer joins
- compound keys

Transactions

Vertica supports conventional SQL transactions with standard ACID properties. Specifically:

- Vertica supports ANSI SQL 92 style implicit transactions. You do not need to execute a BEGIN or START TRANSACTION command.
- Vertica does not use a redo/undo log or two-phase commit.

- The ***COPY*** (page 143) command automatically commits itself and any current transaction. Vertica recommends that you ***COMMIT*** (page 142) or ***ROLLBACK*** (page 173) the current transaction before using COPY.

Session-Scoped Isolation Levels

Vertica supports a **subset of the standard SQL isolation levels and access modes** for a user session as described in SERIALIZABLE Isolation and READ COMMITTED Isolation. These modes determine what data a transaction can access when other transactions are running concurrently. You can change the default isolation level for a user session using the ***SET SESSION CHARACTERISTICS*** (page 185) command.

> Session-scoped isolation levels do not apply to temporary tables, which are scoped to the current transaction. They do not take table locks, are only visible to one user, and their contents are always visible regardless of advancing epochs and COMMITs.

Query-Scoped Isolation Levels

Snapshot Isolation (historical queries) are part of the Vertica query syntax:

```
[ AT EPOCH LATEST ] | [ AT TIME 'timestamp' ] SELECT ...
```

AT EPOCH LATEST allows queries to access all historical data up to but not including the current epoch . It does not hold locks and does not block write operations. This provides a **profound query performance advantage**. It does not apply to temporary tables.

Automatic Rollback

When an error occurs or a user session is disconnected, the current transaction automatically rolls back. This behavior may be different from other databases.

SQL Language Elements

Keywords and Reserved Words

Keywords are words that have a specific meaning in the SQL language. Although SQL is not case-sensitive with respect to keywords, they are generally shown in uppercase letters throughout this documentation for readability purposes.

Some SQL keywords are also reserved words that cannot be used in an identifier unless enclosed in double quote (") characters.

Keywords

ABORT	ABSOLUTE	ACCESS	ACTION	ADD
AFTER	AGGREGATE	ALL	ALSO	ALTER
ANALYSE	ANALYZE	AND	ANY	ARRAY
AS	ASC	ASSERTION	ASSIGNMENT	AT
AUTHORIZATION	BACKWARD	BEFORE	BEGIN	BETWEEN
BIGINT	BINARY	BIT	BLOCK_DICT	BLOCKDICT_COMP
BOOLEAN	BOTH	BY	CACHE	CALLED
CASCADE	CASE	CAST	CATALOGPATH	CHAIN
CHAR	CHARACTER	CHARACTERISTICS	CHECK	CHECKPOINT
CLASS	CLOSE	CLUSTER	COALESCE	COLLATE
COLUMN	COMMENT	COMMIT	COMMITTED	COMMONDELTA_COMP
CONSTRAINT	CONSTRAINTS	CONVERSION	CONVERT	COPY
CORRELATION	CREATE	CREATEDB	CREATEUSER	CROSS
CSV	CURRENT_DATABASE	CURRENT_DATE	CURRENT_TIME	CURRENT_TIMESTAMP
CURRENT_USER	CURSOR	CYCLE	DATA	DATABASE
DATAPATH	DAY	DEALLOCATE	DEC	DECIMAL
DECLARE	DEFAULT	DEFAULTS	DEFERRABLE	DEFERRED
DEFINER	DELETE	DELIMITER	DELIMITERS	DELTARANGE_COMP

DELTARANGE_COMP_SP	DELTAVAL	DESC	DETERMINES	DIRECT
DISTINCT	DISTVALINDEX	DO	DOMAIN	DOUBLE
DROP	EACH	ELSE	ENCODING	ENCRYPTED
END	EPOCH	ERROR	ESCAPE	EXCEPT
EXCEPTIONS	EXCLUDING	EXCLUSIVE	EXECUTE	EXISTS
EXPLAIN	EXTERNAL	EXTRACT	FALSE	FETCH
FIRST	FLOAT	FOR	FORCE	FOREIGN
FORWARD	FREEZE	FROM	FULL	FUNCTION
GLOBAL	GRANT	GROUP	HANDLER	HAVING
HOLD	HOUR	ILIKE	IMMEDIATE	IMMUTABLE
IMPLICIT	IN	INCLUDING	INCREMENT	INDEX
INHERITS	INITIALLY	INNER	INOUT	INPUT
INSENSITIVE	INSERT	INSTEAD	INT	INTEGER
INTERSECT	INTERVAL	INTO	INVOKER	IS
ISNULL	ISOLATION	JOIN	KEY	LANCOMPILER
LANGUAGE	LARGE	LAST	LATEST	LEADING
LEFT	LESS	LEVEL	LIKE	LIMIT
LISTEN	LOAD	LOCAL	LOCALTIME	LOCALTIMESTAMP
LOCATION	LOCK	MATCH	MAXVALUE	MERGEOUT
MINUTE	MINVALUE	MOBUF	MODE	MONTH
MOVE	MOVEOUT	MULTIALGORITHM_COMP	MULTIALGORITHM_COMP_SP	NAMES
NATIONAL	NATURAL	NCHAR	NEW	NEXT
NO	NOCREATEDB	NOCREATEUSER	NODE	NODES
NONE	NOT	NOTHING	NOTIFY	NOTNULL
NOWAIT	NULL	NULLIF	NUMERIC	OBJECT
OF	OFF	OFFSET	OIDS	OLD
ON	ONLY	OPERATOR	OPTION	OR
ORDER	OUT	OUTER	OVERLAPS	OVERLAY
OWNER	PARTIAL	PASSWORD	PLACING	POSITION
PRECISION	PREPARE	PRESERVE	PRIMARY	PRIOR
PRIVILEGES	PROCEDURAL	PROCEDURE	PROJECTION	QUOTE

READ	REAL	RECHECK	RECORD	RECOVER
REFERENCES	REFRESH	REINDEX	REJECTED	RELATIVE
RELEASE	RENAME	REPEATABLE	REPLACE	RESET
RESTART	RESTRICT	RETURNS	REVOKE	RIGHT
RLE	ROLLBACK	ROW	ROWS	RULE
SAVEPOINT	SCHEMA	SCROLL	SECOND	SECURITY
SEGMENTED	SELECT	SEQUENCE	SERIALIZABLE	SESSION
SESSION_USER	SET	SETOF	SHARE	SHOW
SIMILAR	SIMPLE	SITE	SITES	SMALLINT
SOME	SPLIT	STABLE	START	STATEMENT
STATISTICS	STDIN	STDOUT	STORAGE	STRICT
SUBSTRING	SYSID	TABLE	TABLESPACE	TEMP
TEMPLATE	TEMPORARY	TERMINATOR	THAN	THEN
TIME	TIMESTAMPK	TIMESTAMPTZ	TIMETZ	TO
TOAST	TRAILING	TRANSACTION	TREAT	TRIGGER
TRIM	TRUE	TRUNCATE	TRUSTED	TYPE
UNCOMMITTED	UNENCRYPTED	UNION	UNIQUE	UNKNOWN
UNLISTEN	UNSEGMENTED	UNTIL	UPDATE	USAGE
USER	USING	VACUUM	VALID	VALIDATOR
VALINDEX	VALUES	VARCHAR	VARYING	VERBOSE
VIEW	VOLATILE	WHEN	WHERE	WITH
WITHOUT	WORK	WRITE	YEAR	ZONE

Reserved Words

ALL	ANALYSE	ANALYZE
AND	ANY	ARRAY
AS	ASC	BOTH
CASE	CAST	CHECK
COLLATE	COLUMN	CONSTRAINT

CREATE	CURRENT_DATABASE	CURRENT_DATE
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER
DEFAULT	DEFERRABLE	DESC
DISTINCT	DO	ELSE
END	EXCEPT	FALSE
FOR	FOREIGN	FROM
GRANT	GROUP	HAVING
IN_P	INITIALLY	INTERSECT
INTO	LEADING	LIMIT
LOCALTIME	LOCALTIMESTAMP	NEW
NODE	NODES	NOT
NULL	OFF	OFFSET
OLD	ON	ONLY
OR	ORDER	PLACING
PRIMARY	REFERENCES	SELECT
SESSION_USER	SOME	TABLE
THEN	TO	TRAILING
TRUE_P	UNION	UNIQUE
USER	USING	WHEN
WHERE		

Identifiers

Identifiers (names) of objects such as tables, projections, users, etc., can be up to 128 bytes in length.

Unquoted Identifiers

Unquoted SQL identifiers must begin with one of the following:

- an alphabetic character (a-z including letters with diacritical marks and non-Latin letters)
- underscore (_)

Subsequent characters in an identifier can be:

- alphabetic
- numeric (0-9)
- dollar sign ($) (not allowed in identifiers according to the SQL standard and thus may cause application portability problems)

Unquoted identifiers are not case-sensitive. Thus, the unquoted identifiers ABC, ABc, and aBc are synonymous.

Quoted Identifiers

Identifiers enclosed in double quote (") characters can contain any character other than a double quote itself. (To include a double quote, write two double quotes.) This allows you to use names that would otherwise be invalid, such as ones that include only numeric characters ("123" for example) or contain space characters, punctuation marks, keywords, etc.

Quoted identifiers are case-sensitive. Thus, the quoted identifiers "ABC", "ABc", and "aBc" are distinct.

Constants

Numeric Constants

Syntax

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

Semantics

digits represents one or more numeric characters (0 through 9).

Notes

- At least one digit must be before or after the decimal point, if one is used.
- At least one digit must follow the exponent marker (e), if one is present.

- There cannot be any spaces or other characters embedded in the constant.
- Leading plus or minus signs are not actually considered part of the constant; they are unary operators applied to the constant.
- A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type INTEGER if its value fits; otherwise it is presumed to be DOUBLE PRECISION.
- In most cases a numeric constant is automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it as described in ***Data Type Coercion Operators (CAST)*** (page 39).

Examples

```
42
3.5
4.
.001
5e2
1.925e-3
```

String Constants (Dollar-Quoted)

The standard syntax for specifying string constants can be difficult to understand when the desired string contains many single quotes or backslashes. To allow more readable queries in such situations, Vertica SQL provides "dollar quoting." Dollar quoting is not part of the SQL standard, but it is often a more convenient way to write complicated string literals than the standard-compliant single quote syntax. It is particularly useful when representing string constants inside other constants.

Syntax

$$*characters*$$

Semantics

characters is an arbitrary sequence of UTF8 characters bounded by paired dollar signs ($$).

Dollar-quoted string content is treated as a literal. Single quote, backslash, and dollar sign characters have no special meaning within a dollar-quoted string.

Notes

- A dollar-quoted string that follows a keyword or identifier must be separated from it by whitespace; otherwise the dollar quoting delimiter would be taken as part of the preceding identifier.
- The string functions not handle multibyte UTF-8 sequences correctly. They treat each byte as a character.

Examples

```
=> SELECT $$Fred's\n car$$;
     ?column?
------------------
 Fred's\n car
(1 row)
```

String Constants (Standard)

Syntax

'*characters*'

Semantics

characters is an arbitrary sequence of UTF8 characters bounded by single quotes (').

Using Single Quotes in a String

The SQL standard way of writing a single-quote character within a string constant is to write two adjacent single quotes. for example:

```
'Chester''s gorilla'
```

Vertica SQL also allows single quotes to be escaped with a backslash (\). For example:

```
'Chesters\'s gorilla'
```

C-Style Backslash Escapes

Vertica SQL also supports the following C-style backslash escapes. Any other character following a backslash is taken literally.

- `\\` is a backslash
- `\b` is a backspace
- `\f` is a form feed
- `\n` is a newline
- `\r` is a carriage return
- `\t` is a tab
- `\`*xxx*, where `xxx` is an octal number representing a byte with the corresponding code. (It is your responsibility that the byte sequences you create are valid characters in the server character set encoding. The character with the code zero cannot be in a string constant.)

Notes

- Vertica supports the UTF-8 character set.

- The string functions not handle multibyte UTF-8 sequences correctly. They treat each byte as a character.

Examples

```
=> SELECT 'This is a string';
     ?column?
------------------
 This is a string
(1 row)
```

Date/Time Constants

Date or time literal input must be enclosed in single quotes. Input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. Vertica is more flexible in handling date/time input than the SQL standard requires.The exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones are described in ***Date/Time Expressions*** (page 47).

Time Zones

Vertica attempts to be compatible with the SQL standard definitions for time zones. However, the SQL standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the ***DATE*** (page 63) type does not have an associated time zone, the ***TIME*** (page 64) type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset may vary through the year with daylight-saving time boundaries.
- Vertica assumes your local time zone for any data type containing only date or time.
- The default time zone is specified as a constant numeric offset from UTC. It is therefore not possible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, Vertica recommends using date/time types that contain both date and time when using time zones. We recommend *not* using the type `TIME WITH TIME ZONE` (though it is supported by Vertica for legacy applications and for compliance with the SQL standard).

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900's, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules.

Daylight saving time starts on March 11 at 2:00 am in 2007.

Vertica currently supports daylight-savings rules over the time period 1902 through 2038 (corresponding to the full range of conventional Unix system time). Times outside that range are taken to be in "standard time" for the selected time zone, no matter what part of the year they fall in.

Example	Description
PST	Pacific Standard Time
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST
zulu	Military abbreviation for UTC
z	Short form of zulu

Day of the Week Names

The following tokens are recognized as names of days of the week.

Day	Abbreviations
SUNDAY	SUN
MONDAY	MON
TUESDAY	TUE, TUES
WEDNESDAY	WED, WEDS
THURSDAY	THU, THUR, THURS
FRIDAY	FRI
SATURDAY	SAT

Month Names

The following tokens are recognized as names of months.

Month	Abbreviations
JANUARY	JAN

FEBRUARY	FEB
MARCH	MAR
APRIL	APR
MAY	
JUNE	JUN
JULY	JUL
AUGUST	AUG
SEPTEMBER	SEP, SEPT
OCTOBER	OCT
NOVEMBER	NOV
DECEMBER	DEC

Date/Time Field Modifiers

The following tokens serve various modifier purposes.

Identifier	Description
AM	Time is before 12:00
AT	Ignored
JULIAN, JD, J	Next field is Julian Day
ON	Ignored
PM	Time is on or after 12:00
T	Next field is time

Operators

Boolean Operators

Syntax

```
[ AND | OR | NOT ]
```

Semantics

SQL uses a three-valued Boolean logic where the null value represents "unknown."

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Notes

- The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result. However, the order of evaluation of subexpressions is not defined. When it is essential to force evaluation order, use a ***CASE*** (page 45) construct.
- Do not confuse Boolean operators with the ***Boolean-predicate*** (page 51) or the ***Boolean*** (page 59) data type, which can have only two values: true and false.

Comparison Operators

Comparison operators are available for all data types where comparison makes sense. All comparison operators are binary operators that return values of True, False, or NULL.

Syntax and Semantics

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

Notes

The != operator is converted to <> in the parser stage. It is not possible to implement != and <> operators that do different things.

- The comparison operators return NULL (signifying "unknown") when either operand is null.

Data Type Coercion Operators (CAST)

Type coercion (casting) passes an expression value to an input conversion routine for a specified data type, resulting in a constant of the indicated type.

Syntax

```
CAST ( expression AS data-type )
expression::data-type
data-type 'string'
```

Semantics

expression	is an expression of any type
data-type	converts the value of *expression* to one of the following data types: ***BOOLEAN*** (on page 59) ***CHARACTER (CHAR)*** (on page 61) ***CHARACTER VARYING (VARCHAR)*** (on page 61) ***Date/Time Types*** (see "Date/Time" on page 63) ***DOUBLE PRECISION (FLOAT)*** (on page 70) ***INTEGER (BIGINT)*** (on page 73)

Notes

- Type coercion format of *data-type* '*string*' can only be used to specify the data type of a quoted string constant
- The explicit type cast can be omitted if there is no ambiguity as to the type the constant must be. For example, when a constant is assigned directly to a column it is automatically coerced to the column's data type.

Examples

```
=> SELECT CAST((2 + 2) AS VARCHAR);
 varchar
---------
 4
(1 row)

=> SELECT (2 + 2)::VARCHAR;
 varchar
---------
 4
(1 row)

=> SELECT '2.2' + 2;
ERROR:  invalid input syntax for integer: "2.2"
=> SELECT FLOAT '2.2' + 2;
 ?column?
----------
      4.2
(1 row)
```

Date/Time Operators

Syntax

```
[ + | - | * | / ]
```

Semantics

+	addition
-	subtraction
*	multiplication
/	division

Notes

- The operators described below that take `TIME` or `TIMESTAMP` inputs actually come in two variants: one that takes `TIME WITH TIME ZONE` or `TIMESTAMP WITH TIME ZONE`, and one that takes `TIME WITHOUT TIME ZONE` or `TIMESTAMP WITHOUT TIME ZONE`. For brevity, these variants are not shown separately.

- The + and * operators come in commutative pairs (for example both DATE + INTEGER and INTEGER + DATE); only one of each such pair is shown.

Example	Result Type	Result
DATE '2001-09-28' + INTEGER '7'	DATE	'2001-10-05'
DATE '2001-09-28' + INTERVAL '1 HOUR'	TIMESTAMP	'2001-09-28 01:00:00'
DATE '2001-09-28' + TIME '03:00'	TIMESTAMP	'2001-09-28 03:00:00'
INTERVAL '1 DAY' + INTERVAL '1 HOUR'	INTERVAL	'1 DAY 01:00:00'
TIMESTAMP '2001-09-28 01:00' + INTERVAL '23 HOURS'	TIMESTAMP	'2001-09-29 00:00:00'
TIME '01:00' + INTERVAL '3 HOURS'	TIME	'04:00:00'
- INTERVAL '23 HOURS'	INTERVAL	'-23:00:00'
DATE '2001-10-01' - DATE '2001-09-28'	INTEGER	'3'
DATE '2001-10-01' - INTEGER '7'	DATE	'2001-09-24'
DATE '2001-09-28' - INTERVAL '1 HOUR'	TIMESTAMP	'2001-09-27 23:00:00'
TIME '05:00' - TIME '03:00'	INTERVAL	'02:00:00'
TIME '05:00' - INTERVAL '2 HOURS'	TIME	'03:00:00'
TIMESTAMP '2001-09-28 23:00' - INTERVAL '23 HOURS'	TIMESTAMP	'2001-09-28 00:00:00'
INTERVAL '1 DAY' - INTERVAL '1 HOUR'	INTERVAL	'1 DAY -01:00:00'
TIMESTAMP '2001-09-29 03:00' - TIMESTAMP '2001-09-27 12:00'	INTERVAL	'1 DAY 15:00:00'
900 * INTERVAL '1 SECOND'	INTERVAL	'00:15:00'
21 * INTERVAL '1 DAY'	INTERVAL	'21 DAYS'
DOUBLE PRECISION '3.5' * INTERVAL '1 HOUR'	INTERVAL	'03:30:00'
INTERVAL '1 HOUR' / DOUBLE PRECISION '1.5'	INTERVAL	'00:40:00'

Mathematical Operators

Mathematical operators are provided for many data types.

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division (integer division truncates results)	4 / 2	2
%	modulo (remainder)	5 % 4	1
^	exponentiation	2.0 ^ 3.0	8
|/	square root	|/ 25.0	5
||/	cube root	||/ 27.0	3
!	factorial	5 !	120
!!	factorial (prefix operator)	!! 5	120
@	absolute value	@ -5.0	5
&	bitwise AND	91 & 15	11
|	bitwise OR	32 | 3	35

#	bitwise XOR	17 # 5	20
~	bitwise NOT	~1	-2
<<	bitwise shift left	1 << 4	16
>>	bitwise shift right	8 >> 2	2

Notes

- The bitwise operators work only on integer data types, whereas the others are available for all numeric data types.
- Vertica supports the use of the factorial operators on positive and negative floating point (***DOUBLE PRECISION*** (page 70)) numbers as well as integers. For example:

```
=> select 4.98!;
     ?column?
------------------
 115.978600750905
(1 row)
```

  Factorial is defined as z! = gamma(z+1) for all complex numbers z. See the ***Handbook of Mathematical Functions*** http://www.math.sfu.ca/~cbm/aands/ (1964) Section 6.1.5.

String Concatenation Operators

To concatenate two strings on a single line, use the concatenation operator (two consecutive vertical bars).

Syntax

string || *string*

Semantics

string	is an expression of type CHAR or VARCHAR

Notes

- Two consecutive strings within a single SQL statement on separate lines are concatenated.

Examples

```
> SELECT 'auto' || 'mobile';
 ?column?
----------
 automobile
(1 row)

> SELECT 'auto'
> 'mobile';
 ?column?
----------
 automobile
```

```
(1 row)
```

Expressions

Operator Precedence

The following table shows operator precedence in decreasing (high to low) order.

When an expression includes more than one operator, Vertica Systems, Inc. recommends that you specify the order of operation using parentheses, rather than relying on operator precedence.

Operator/Element	Associativity	Description
`.`	left	table/column name separator
`::`	left	typecast
`[ ]`	left	array element selection
`-`	right	unary minus
`^`	left	exponentiation
`* / %`	left	multiplication, division, modulo
`+ -`	left	addition, subtraction
`IS`		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
`IN`		set membership
`BETWEEN`		range containment
`OVERLAPS`		time interval overlap
`LIKE`		string pattern matching
`< >`		less than, greater than
`=`	right	equality, assignment
`NOT`	right	logical negation
`AND`	left	logical conjunction

OR	left	logical disjunction

Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order. To force evaluation in a specific order, use a ***CASE*** (page 45) construct. For example, this is an untrustworthy way of trying to avoid division by zero in a WHERE clause:

```
SELECT x, y
WHERE x <> 0 AND y/x > 1.5;
```

But this is safe:

```
SELECT x, y
WHERE
  CASE
    WHEN x <> 0 THEN y/x > 1.5
    ELSE false
  END;
```

A CASE construct used in this fashion defeats optimization attempts, so it should only be done when necessary. (In this particular example, it would doubtless be best to sidestep the problem by writing `y > 1.5*x` instead.)

Aggregate Expressions

An aggregate expression represents the application of an ***aggregate function*** (page 76) across the rows or groups of rows selected by a query. The syntax of an aggregate expression is one of the following (using AVG as an example):

`AVG (`*`expression`*`)`

invokes the aggregate across all input rows for which the given expression yields a non-null value.

`AVG (ALL `*`expression`*`)`

is the same as `AVG(`*`expression`*`)`, since ALL is the default.

`AVG (DISTINCT `*`expression`*`)`

invokes the AVG function across all input rows for all distinct, non-null values of the expression.

where *expression* is any value expression that does not itself contain an aggregate expression.

An aggregate expression only can appear in the select list or HAVING clause of a SELECT statement. It is forbidden in other clauses, such as WHERE, because those clauses are evaluated before the results of aggregates are formed.

CASE Expressions

The CASE expression is a generic conditional expression that can be used wherever an expression is valid. It is similar to case and if/then/else statements in other languages.

Syntax (Form 1)

```
CASE
  WHEN condition THEN result
  [ WHEN condition THEN result ]...
  [ ELSE result ]
END
```

Semantics

condition	is an expression that returns a boolean (true/false) result. If the result is false, subsequent WHEN clauses are evaluated in the same manner.
result	specifies the value to return when the associated *condition* is true.
ELSE *result*	If no *condition* is true then the value of the CASE expression is the result in the ELSE clause. If the ELSE clause is omitted and no condition matches, the result is null.

Syntax (Form 2)

```
CASE expression
 WHEN value THEN result
 [ WHEN value THEN result ]...
 [ ELSE result ]
END
```

Semantics

expression	is an expression that is evaluated and compared to all the *value* specifications in the WHEN clauses until one is found that is equal.
value	specifies a value to compare to the *expression*.
result	specifies the value to return when the *expression* is equal to the specified *value*.
ELSE *result*	specifies the value to return when the *expression* is not equal to any *value*; if no ELSE clause is specified, the value returned is null.

Notes

- The data types of all the result expressions must be convertible to a single output type.

Examples

```
SELECT * FROM test;
 a
---
 1
 2
 3

SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other

SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

Special Example

A CASE expression does not evaluate subexpressions that are not needed to determine the result. You can use this behavior to avoid division-by-zero errors:

```
SELECT x
  FROM T1
  WHERE
    CASE WHEN x <> 0 THEN y/x > 1.5
    ELSE false
  END;
```

Column References

Syntax

[*tablename*.] *columnname*

Semantics

tablename	is one of:

	• the name of a table • an alias for a table defined by means of a FROM clause in a query
columnname	is the name of a column that must be unique across all the tables being used in a query

Notes

- There are no space characters in a column reference.

Comments

A comment is an arbitrary sequence of characters beginning with two consecutive hyphen characters and extending to the end of the line. For example:

```
-- This is a standard SQL comment
```

A comment is removed from the input stream before further syntax analysis and is effectively replaced by white space.

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

where the comment begins with `/*` and extends to the matching occurrence of `*/`. These block comments nest, as specified in the SQL standard but unlike C, so that one can comment out larger blocks of code that may contain existing block comments.

Date/Time Expressions

Vertica uses an internal heuristic parser for all date/time input support. Dates and times are input as strings, and are broken up into distinct fields with a preliminary determination of what kind of information may be in the field. Each field is interpreted and either assigned a numeric value, ignored, or rejected. The parser contains internal lookup tables for all textual fields, including months, days of the week, and time zones.

The date/time type inputs are decoded using the following procedure.

- Break the input string into tokens and categorize each token as a string, time, time zone, or number.
- If the numeric token contains a colon (:), this is a time string. Include all subsequent digits and colons.
- If the numeric token contains a dash (-), slash (/), or two or more dots (.), this is a date string which may have a text month.
- If the token is numeric only, then it is either a single field or an ISO 8601 concatenated date (e.g., 19990113 for January 13, 1999) or time (e.g., 141516 for 14:15:16).
- If the token starts with a plus (+) or minus (-), then it is either a time zone or a special field.
- If the token is a text string, match up with possible strings.

- Do a binary-search table lookup for the token as either a special string (e.g., today), day (e.g., Thursday), month (e.g., January), or noise word (e.g., at, on).
- Set field values and bit mask for fields. For example, set year, month, day for today, and additionally hour, minute, second for now.
- If not found, do a similar binary-search table lookup to match the token with a time zone.
- If still not found, throw an error.
- When the token is a number or number field:
- If there are eight or six digits, and if no other date fields have been previously read, then interpret as a "concatenated date" (e.g., 19990118 or 990118). The interpretation is YYYYMMDD or YYMMDD.
- If the token is three digits and a year has already been read, then interpret as day of year.
- If four or six digits and a year has already been read, then interpret as a time (HHMM or HHMMSS).
- If three or more digits and no date fields have yet been found, interpret as a year (this forces yy-mm-dd ordering of the remaining date fields).
- Otherwise the date field ordering is assumed to follow the DateStyle setting: mm-dd-yy, dd-mm-yy, or yy-mm-dd. Throw an error if a month or day field is found to be out of range.
- If BC has been specified, negate the year and add one for internal storage. (There is no year zero in the Gregorian calendar, so numerically 1 BC becomes year zero.)
- If BC was not specified, and if the year field was two digits in length, then adjust the year to four digits. If the field is less than 70, then add 2000, otherwise add 1900.

Tip: Gregorian years AD 1-99 can be entered by using 4 digits with leading zeros (e.g., 0099 is AD 99).

Month Day Year Ordering

For some formats, ordering of month, day, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields. See Date/Time Run-Time Parameters for information about output styles such as `POSTGRES`.

Special Date/Time Values

Vertica supports several special date/time values for convenience, as shown below. All of these values need to be written in single quotes when used as constants in SQL statements.

The values `INFINITY` and `-INFINITY` are specially represented inside the system and are displayed the same way. The others are simply notational shorthands that are converted to ordinary date/time values when read. (In particular, `NOW` and related strings are converted to a specific time value as soon as they are read.)

String	Valid Data Types	Description
epoch	DATE, TIMESTAMP	1970-01-01 00:00:00+00 (UNIX SYSTEM TIME ZERO)
INFINITY	TIMESTAMP	Later than all other time stamps
-INFINITY	TIMESTAMP	Earlier than all other time stamps
NOW	DATE, TIME, TIMESTAMP	Current transaction's start time NOW is not the same as the ***NOW*** (on page 89) function.
TODAY	DATE, TIMESTAMP	Midnight today
TOMORROW	DATE, TIMESTAMP	Midnight tomorrow
YESTERDAY	DATE, TIMESTAMP	Midnight yesterday
ALLBALLS	TIME	00:00:00.00 UTC

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type:

CURRENT_DATE (page 84)

CURRENT_TIME (page 84)

CURRENT_TIMESTAMP (page 85)

LOCALTIME (page 88)

LOCALTIMESTAMP (page 88)

The latter four accept an optional precision specification. (See ***Date/Time Functions*** (page 82).) Note however that these are SQL functions and are not recognized as data input strings.

NULL Value

NULL is a reserved keyword used to indicate that a data value is unknown.

Be very careful when using NULL in expressions. NULL is not greater than, less than, equal to, or not equal to any other expression. Use the ***Boolean-predicate*** (on page 51) for determining whether or not an expression value is NULL.

Notes

- NULL appears last (largest) in ascending order.

Numeric Expressions

Notes

- The result of NaN > 1 is 'Unknown', which is the same as NULL or not true.
- Vertica follows the IEEE specification for floating point.

Predicates

BETWEEN-predicate

The special `BETWEEN` predicate is available as a convenience.

Syntax

a `BETWEEN` *x* `AND` *y*

Semantics

a BETWEEN *x* AND *y* is equivalent to

a `>=` *x* `AND` *a* `<=` *y*

Similarly,

a `NOT BETWEEN` *x* `AND` *y*

is equivalent to

a < *x* OR *a* > *y*

Boolean-predicate

The Boolean predicate retrieves rows where the value of an expression is true, false, or unknown (null).

Syntax

```
expression IS [NOT] TRUE
expression IS [NOT] FALSE
expression IS [NOT] UNKNOWN
```

Semantics

A Boolean predicate always return true or false, never a null value, even when the operand is null. A null input is treated as the value UNKNOWN.

Notes

- Do not confuse the boolean-predicate with ***Boolean Operators*** (on page 38) or the ***Boolean*** (page 59) data type, which can have only two values: true and false.
- `IS UNKNOWN` and `IS NOT UNKNOWN` are effectively the same as the ***NULL-predicate*** (page 57), except that the input expression does not have to be a single column value. To check a single column value for NULL, use the ***NULL-predicate*** (page 57).

column-value-predicate

Syntax

column-name comparison-op constant-expression

Semantics

column-name	is a single column of one the tables specified in the ***FROM clause*** (page 176).
comparison-op	is one of the ***comparison operators*** (on page 38).
constant-expression	is a constant value of the same data type as the *column-name*

Notes

- To check a column value for NULL, use the ***NULL-predicate*** (page 57).

Examples

```
Dimension.column1 = 2
Dimension.column2 = 'Seafood'
```

IN-predicate

Syntax

column-expression [NOT] IN (*list-expression*)

Semantics

column-expression — is a single column of one the tables specified in the ***FROM clause*** (page 176).

list-expression — is a comma-separated list of constant values matching the data type of the *column-expression*

Examples

```
x IN (5, 6, 7)
x in (select a from table)        not allowed
(x, y) in ((5,6), (7,8))          not allowed
```

join-predicate

Vertica supports only equi-joins based on a primary key-foreign key relationship between the joined tables. See Adding Join Constraints in the Database Administrator's Guide for information about defining primary and foreign keys.

Syntax

column-reference (see "Column References" on page 46) = *column-reference* (see "Column References" on page 46)

Semantics

column-reference	refers to a column of one the tables specified in the ***FROM clause*** (page 176).

Notes

- Self-joins are not allowed.

LIKE-predicate

The LIKE predicate retrieves rows where the string value of a column matches a specified pattern. The pattern can contain one or more wildcard characters, which match all valid characters. ILIKE is equivalent to LIKE except that the match is case-insensitive (non-standard extension).

Syntax

string { LIKE | ILIKE } *pattern* [ESCAPE *escape-character*]

string NOT { LIKE | ILIKE } *pattern* [ESCAPE *escape-character*]

Semantics

string	(CHAR or VARCHAR) is the column value to be compared to the *pattern*.
NOT	returns true if LIKE returns false, and vice versa (equivalent to NOT *string* LIKE *pattern*).
pattern	specifies a string containing wildcard characters. **underscore (_)** matches any single character. **percent sign (%)** matches any string of zero or more characters.
ESCAPE	specifies an *escape-character*. A null escape character ('') disables the escape mechanism. To match the escape character itself, use two consecutive escape characters.
escape-character	when preceding an underscore or percent sign character in the *pattern*, causes that character to be treated as a literal rather than a wildcard. The default escape character is the backslash (/) character.

Notes

- LIKE requires the entire string expression to match the pattern. To match a sequence of characters anywhere within a string, the pattern must start and end with a percent sign.
- The LIKE predicate does not ignore trailing "white space" characters. If the data values that you want to match have unknown numbers of trailing spaces, tabs, etc., terminate each LIKE predicate pattern with the percent sign wildcard character.
- To use a backslash character as a literal, specify a different escape character and use two backslashes. For example, if the escape character is circumflex (^), you would use ^\\ to specify a literal backslash. (Alternative: simply use four backslashes.)
- The use of a column data type other than character or character varying (implicit string conversion) is not supported and not recommended.
- Error messages caused by the LIKE predicate may refer to it by the following symbols instead of the actual keywords:

~~	LIKE
~~*	ILIKE

!~~	NOT LIKE
!~~*	NOT ILIKE

Examples

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

NULL-predicate

Syntax

column-name IS [NOT] NULL

Semantics

column-name	is a single column of one the tables specified in the ***FROM clause*** (page 176).

Examples

```
a IS NULL
b IS NOT NULL
```

See Also

NULL Value (page 49)

SQL Data Types

Vertica supports a subset of the PostgreSQL data types.

Implicit Data Type Coercion

When there is no ambiguity as to the data type of an expression value, it is implicitly coerced to match the expected data type. For example:

```
=> SELECT 2 + '2';
 ?column?
----------
        4
(1 row)
```

The quoted string constant '2' is implicitly coerced into an INTEGER value so that it can be the operand of an arithmetic operator (addition).

```
=> SELECT 2 + 2 || 2;
 ?column?
----------
 42
(1 row)
```

The result of the arithmetic expression 2 + 2 and the INTEGER constant 2 are implicitly coerced into a VARCHAR values so that they can be concatenated.

BOOLEAN

Vertica provides the standard SQL type BOOLEAN, which has two states: true and false. The third state in SQL boolean logic is unknown, which is represented by the NULL value.

Syntax

```
BOOLEAN
```

Semantics

Valid literal data values for input are:

TRUE	't'	'true'	'y'	'yes'	'1'
FALSE	'f'	'false'	'n'	'no'	'0'

Notes

- Do not confuse the BOOLEAN data type with ***Boolean Operators*** (on page 38) or the ***Boolean-predicate*** (on page 51).
- The keywords TRUE and FALSE are preferred (and SQL-compliant).

- All other values must be enclosed in single quotes.
- Boolean values are output using the letters t and f.

CHARACTER (CHAR)

The CHARACTER type is a fixed-length, blank padded string.

Syntax

```
[ CHARACTER | CHAR ] (n)
```

Semantics

n	specifies the length of the string. The default length is one (1). The maximum length is 4000.

Notes

- On input, strings are truncated if necessary to the specified number of characters.
- Remember to include the extra bytes required for multibyte characters in the column width declaration.
- String literals in SQL statements must be enclosed in single quotes.
- Vertica does not support the PostgreSQL TEXT type. Use VARCHAR instead.
- Due to Vertica's use of compression, the cost of over-estimating the length of these fields is incurred mostly at load time and during sorts.
- CHAR fields are padded on the right with spaces as needed.
- No embedded NULs are allowed.
- In CHAR fields NUL characters are handled as ordinary characters.
- NULL appears last (largest) in ascending order.

CHARACTER VARYING (VARCHAR)

The CHARACTER VARYING type is a variable-length string.

Syntax

```
[ CHARACTER VARYING | VARCHAR ] (n)
```

Semantics

n	specifies the maximum length of the string. The default length is 80. The maximum length is 4000.

Notes

- On input, strings are truncated if necessary to the specified number of characters.
- Remember to include the extra bytes required for multibyte characters in the column width declaration.
- String literals in SQL statements must be enclosed in single quotes.

- Vertica does not support the PostgreSQL TEXT type. Use VARCHAR instead.
- Due to Vertica's use of compression, the cost of over-estimating the length of these fields is incurred mostly at load time and during sorts.
- VARCHAR values terminate at their first NUL byte if any.
- A VARCHAR(n) field is processed internally as a NUL-padded string of maximum length n.
- No embedded NULs are allowed.
- NULL appears last (largest) in ascending order.

Date/Time

Vertica supports the full set of SQL date and time types. In most cases, a combination of `DATE`, `TIME`, `TIMESTAMP WITHOUT TIME ZONE`, and `TIMESTAMP WITH TIME ZONE` should provide a complete range of date/time functionality required by any application. However, in compliance with the SQL standard, Vertica also supports the `TIME WITH TIMEZONE` data type.

Notes

- Vertica uses Julian dates for all date/time calculations. They can correctly predict and calculate any date more recent than 4713 BC to far into the future, based on the assumption that the length of the year is 365.2425 days.
- All date/time types are stored in eight bits.
- A date/time value of NULL appears first (smallest) in ascending order.
- All the date/time data types accept the special literal value `NOW` to specify the current date and time. For example:
  ```
  SELECT TIMESTAMP 'NOW';
  ```

DATE

DATE consists of a month, day, and year. See ***SET DATESTYLE*** (page 183) for information about ordering.

Syntax

`DATE`

Semantics

Low Value	High Value	Resolution
4713 BC	32767 AD	1 DAY

Example	Description
January 8, 1999	unambiguous in any `datestyle` input mode
1999-01-08	ISO 8601; January 8 in any mode (recommended format)
1/8/1999	January 8 in `MDY` mode; August 1 in `DMY` mode
1/18/1999	January 18 in `MDY` mode; rejected in other modes
01/02/03	January 2, 2003 in `MDY` mode February 1, 2003 in `DMY` mode

	February 3, 2001 in YMD mode
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in YMD mode, else error
08-Jan-99	January 8, except error in YMD mode
Jan-08-99	January 8, except error in YMD mode
19990108	ISO 8601; January 8, 1999 in any mode
990108	ISO 8601; January 8, 1999 in any mode
1999.008	year and day of year
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

TIME

TIME consists of a time of day with or without a time zone.

Syntax

TIME [(*p*)] [{ WITH | WITHOUT } TIME ZONE] | TIMETZ [***AT TIME ZONE*** (on page 65)]

Semantics

p	(precision) specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range 0 to 6.
WITH TIME ZONE	specifies that valid values must include a time zone
WITHOUT TIME ZONE	specifies that valid values do not include a time zone (default). If a time zone is specified in the input it is silently ignored.

TIMETZ	is the same as TIME WITH TIME ZONE with no precision

Limits

Data Type	Low Value	High Value	Resolution
TIME [*p*]	00:00:00.00	23:59:59.99	1 MS / 14 digits
TIME [*p*] WITH TIME ZONE	00:00:00.00+12	23:59:59.99-12	1 ms / 14 digits

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	same as 04:05; AM does not affect value
04:05 PM	same as 16:05; input hour must be <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	time zone specified by name

AT TIME ZONE

The AT TIME ZONE construct converts TIME and TIMEZONE types to different time zones.

Syntax

timestamp AT TIME ZONE *zone*

Semantics

timestamp	TIMESTAMP	converts UTC to local time in given time zone
	TIMESTAMP WITH TIME ZONE	converts local time in given time zone to UTC
	TIME WITH TIME ZONE	converts local time across time zones
zone	is the desired time zone specified either as a text string (for example: 'PST') or as an interval (for example: INTERVAL '-08:00'). In the text case, the available zone names are abbreviations.	

Examples

The local time zone is PST8PDT.

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Result: 2001-02-16 19:38:40-05
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Result: 2001-02-16 18:38:40
```

The first example takes a zone-less time stamp and interprets it as MST time (UTC- 7) to produce a UTC time stamp, which is then rotated to PST (UTC-8) for display. The second example takes a time stamp specified in EST (UTC-5) and converts it to local time in MST (UTC-7).

TIMESTAMP

TIMESTAMP consists of a date and a time with or without a time zone and with or without an historical epoch (AD or BC).

Syntax

```
TIMESTAMP [ (p) ] [ { WITH | WITHOUT } TIME ZONE ] | TIMESTAMPTZ
[ AT TIME ZONE (on page 65) ]
```

Semantics

p	(precision) specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range 0 to 6.
WITH TIME ZONE	specifies that valid values must include a time zone. All TIMESTAMP WITH TIMEZONE values are stored internally in UTC. They are converted to local time in the zone specified by the timezone configuration parameter before being displayed to the client.
WITHOUT TIME ZONE	specifies that valid values do not include a time zone (default). If a time zone is specified in the input it is silently ignored.
TIMESTAMPTZ	is the same as TIMESTAMP WITH TIME ZONE with no precision

Limits

Data Type	Low Value	High Value	Resolution
`TIMESTAMP [ (p) ] [ WITHOUT TIME ZONE ]`	4713 BC	5874897 AD	1 MS / 14 digits
`TIMESTAMP [ (p) ] WITH TIME ZONE`	4713 BC	5874897 AD	1 MS / 14 digits

Notes

- AD/BC can appear before the time zone, but this is not the preferred ordering.
- The SQL standard differentiates `TIMESTAMP WITHOUT TIME ZONE` and `TIMESTAMP WITH TIME ZONE` literals by the existence of a "+"; or "-". Hence, according to the standard,
  `TIMESTAMP '2004-10-19 10:23:54'`
  is a `TIMESTAMP WITHOUT TIME ZONE`, while
  `TIMESTAMP '2004-10-19 10:23:54+02'`
  is a `TIMESTAMP WITH TIME ZONE`. Vertica differs from the standard by requiring that `TIMESTAMP WITH TIME ZONE` literals be explicitly typed:
  `TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'`
- If a literal is not explicitly indicated as being of `TIMESTAMP WITH TIME ZONE`, Vertica silently ignores any time zone indication in the literal. That is, the resulting date/time value is derived from the date/time fields in the input value, and is not adjusted for time zone.
- For `TIMESTAMP WITH TIME ZONE`, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's timezone parameter, and is converted to UTC using the offset for the `TIMEZONE` zone.
- When a `TIMESTAMP WITH TIME ZONE` value is output, it is always converted from UTC to the current `TIMEZONE` zone, and displayed as local time in that zone. To see the time in another time zone, either change `TIMEZONE` or use the `AT TIME ZONE` construct (see ***AT TIME ZONE*** (page 65)).
- Conversions between `TIMESTAMP WITHOUT TIME ZONE` and `TIMESTAMP WITH TIME ZONE` normally assume that the `TIMESTAMP WITHOUT TIME ZONE` value should be taken or given as `TIMEZONE` local time. A different zone reference can be specified for the conversion using `AT TIME ZONE`.

Examples

```
1999-01-08 04:05:06

1999-01-08 04:05:06 -8:00
January 8 04:05:06 1999 PST
```

INTERVAL

The INTERVAL type measures the difference between two points in time. It is represented internally as a number of microseconds and printed out as up to:

- 60 seconds
- 60 minutes
- 24 hours
- 30 days
- 12 months
- as many years as necessary.

All the fields are either positive or negative.

Syntax

```
INTERVAL [ (p) ]
```

Semantics

p	(precision) specifies the number of fractional digits retained in the seconds field in the range 0 to 6. The default is the precision of the input literal.

Notes

- INTERVAL values can be written with the following syntax:
  `[@] quantity unit [quantity unit...] [direction]`
  Where: *quantity* is a number (possibly signed); *unit* is `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals of these units; *direction* can be `ago` or empty. The at sign (@) is optional. The amounts of different units are implicitly added up with appropriate sign accounting.
- Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example:
  `'1 12:59:10'` is read the same as `'1 day 12 hours 59 min 10 sec'`

Low Value	High Value	Resolution
-178000000 YEARS	178000000 YEARS	1 MS / 14 DIGITS

Examples

```
=> SELECT TIMESTAMP 'Apr 1, 07' - TIMESTAMP 'Mar 1, 07';
 ?column?
```

```
----------
 1 mon
(1 row)

=> SELECT TIMESTAMP 'Mar 1, 07' - TIMESTAMP 'Feb 1, 07';
 ?column?
----------
 1 mon
(1 row)

=> SELECT TIMESTAMP 'Feb 1, 07' + INTERVAL '30 days';
      ?column?
---------------------
 2007-03-01 00:00:00
(1 row)
```

DOUBLE PRECISION (FLOAT)

Vertica supports the numeric data type DOUBLE PRECISION, which is the IEEE-754 8-byte floating point type along with most of the usual floating point operations.

Syntax

```
[ DOUBLE PRECISION | FLOAT | FLOAT8 ]
```

Semantics

On a machine whose floating-point arithmetic does not follow IEEE 754, these values probably do not work as expected.

Double precision is an inexact, variable-precision numeric type. In other words, some values cannot be represented exactly and are stored as approximations. Thus, input and output operations involving double precision may show slight discrepancies.

- For exact numeric storage and calculations (money for example), use INTEGER.
- Floating point calculations depend on the behavior of the underlying processor, operating system, and compiler.
- Comparing two floating-point values for equality may or may not work as expected.

Values

COPY accepts floating-point data in the following format:

1. optional leading white space
2. an optional plus ("+") or minus sign ("-")
3. a decimal number, a hexadecimal number, an infinity, a NAN, or a null value

A **decimal number** consists of a non-empty sequence of decimal digits possibly containing a radix character (decimal point, locale dependent, usually "."), optionally followed by a decimal exponent. A decimal exponent consists of an "E" or "e", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits, and indicates multiplication by a power of 10.

A **hexadecimal number** consists of a "0x" or "0X" followed by a non-empty sequence of hexadecimal digits possibly containing a radix character, optionally followed by a binary exponent. A binary exponent consists of a "P" or "p", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits, and indicates multiplication by a power of 2. At least one of radix character and binary exponent must be present.

An **infinity** is either "INF" or "INFINITY", disregarding case.

A **NAN** (Not A Number) is "NAN" (disregarding case) optionally followed by a sequence of characters enclosed in parentheses. The character string specifies the value of NAN in an implementation-dependent manner. (The Vertica internal representation of NAN is 0xfff8000000000000LL on x86 machines.)

When writing infinity or NAN values as constants in a SQL statement, enclose them in single quotes. For example:

```
UPDATE table SET x = 'Infinity'
```

> Note: Vertica follows the IEEE definition of NaNs (IEEE 754). (The SQL standards do not specify how floating point works in detail.)
>
> IEEE defines NaNs as a set of floating point values where each one is not equal to anything, even to itself. A NaN is not greater than and at the same time not less than anything, even itself. In other words, comparisons always return false whenever a NaN is involved.
>
> However, for the purpose of sorting data, NaN values must be placed somewhere in the result. The value generated 'NaN' appears in the context of a floating point number matches the NaN value generated by the hardware. For example, Intel hardware generates (0xfff8000000000000LL), which is technically a Negative, Quiet, Non-signaling NaN.
>
> Vertica uses a different NaN value to represent floating point NULL (0x7ffffffffffffffeLL). This is a Positive, Quiet, Non-signaling NaN and is reserved by Vertica.

The load file format of a **null value** is user defined, as described in the ***COPY*** (page 143) command. The Vertica internal representation of a null value is 0x7fffffffffffffffLL. The interactive format is controlled by the vsql printing option null. For example:

```
\pset null '(null)'
```

The default option is not to print anything.

Rules

- -0 == +0
- 1/0 = Infinity
- 0/0 == Nan
- NaN != anything (even NaN)

To search for NaN column values, use the following predicate:

```
... WHERE column != column
```

This is necessary because `WHERE` *column* = `'Nan'` cannot be true by definition.

Sort Order (Ascending)

- NaN
- -Inf
- numbers

- +Inf
- Null

Notes

- Vertica does not support REAL (FLOAT4) or NUMERIC.
- NULL appears last (largest) in ascending order.

INTEGER (BIGINT)

Vertica supports the numeric data type INTEGER, a signed eight-byte (64-bit) data type.

Syntax

```
[ INTEGER | INT | BIGINT ]
```

Semantics

The difference between integer and bigint is in the way that vsql performs input and output. INTEGER is handled as a 32-bit data type and BIGINT is handled as a 64-bit data type.

Notes

The JDBC type INTEGER is 4-bytes, and is not supported by Vertica; use BIGINT instead.

- The range of values is -2^63+1 to 2^63-1.
- 2^63 = 9,223,372,036,854,775,808 (19 digits).
- The value -2^63 is reserved to represent NULL.
- Vertica does not support the SQL/JDBC types NUMERIC, SMALLINT, or TINYINT.
- Vertica does not check for overflow (positive or negative) except in the aggregate function ***SUM*** (page 80)(). If you encounter overflow when using SUM, use ***SUM_FLOAT*** (page 81)() which converts to floating point.
- NULL appears first (smallest) in ascending order.
- Vertica does not have an explicit four-byte (32-bit integer) type. Vertica's encoding and compression automatically eliminate extra space.

SQL Functions

Aggregate Functions

Aggregate functions summarize data over groups of rows from a query result set. The groups are specified using the ***GROUP BY*** (see "GROUP BY Clause" on page 178) clause. They are allowed only in the select list and in the ***HAVING*** (see "HAVING Clause" on page 179) and ***ORDER BY*** (see "ORDER BY Clause" on page 180) clauses of a ***SELECT*** (page 174) statement (as described in ***Aggregate Expressions*** (page 44)).

Notes

- Except for COUNT, these functions return a null value when no rows are selected. In particular, SUM of no rows returns NULL, not zero.
- In some cases you can replace an expression that includes multiple aggregates with an single aggregate of an expression. For example SUM(x) + SUM(y) can be expressed as as SUM(x+y) (where x and y are NOT NULL).
- Vertica does not support nested aggregate functions.

AVG

AVG computes the average (arithmetic mean) of an expression over a group of rows. It returns a DOUBLE PRECISION value for a floating-point expression. Otherwise, the return value is the same as the expression data type.

Syntax

```
AVG ( [ ALL | DISTINCT ] expression )
```

Semantics

`ALL`	invokes the aggregate function for all rows in the group (default)
`DISTINCT`	invokes the aggregate function for all distinct non-null values of the expression found in the group
`expression`	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

COUNT

COUNT returns the number of rows in each group of the result set for which the expression is not null. The return value is a BIGINT.

Syntax

```
COUNT ( [ ALL | DISTINCT ] expression )
```

Semantics

ALL	invokes the aggregate function for all rows in the group (default)
DISTINCT	invokes the aggregate function for all distinct non-null values of the expression found in the group
expression	(any data type) contains at least one ***column reference*** (see "Column References" on page 46)

Examples

Query	Result
SELECT COUNT (DISTINCT x) FROM foo;	the number of distinct values in the x column of table foo
SELECT COUNT (DISTINCT x+y) FROM foo;	all distinct values of evaluating the expression x+y for all tuples of foo. An equivalent query is: SELECT COUNT(x+y) FROM foo GROUP BY foo;
SELECT x, COUNT (DISTINCT y) FROM foo GROUP BY x;	Each distinct x value in table foo The number of distinct values of y in all tuples with the specific distinct x value.
SELECT x, COUNT (DISTINCT x) FROM foo GROUP BY x;	Each distinct x value in table foo The constant "1" The DISTINCT keyword is redundant if all members of the SELECT list are present in the GROUP BY list as well.
SELECT x, COUNT (DISTINCT y), SUM(z) FROM foo GROUP BY x;	Each distinct x value The number of distinct y values for all tuples with the specific x value The sum of all the z values in all tuples with the specific x value
SELECT x, COUNT (DISTINCT y), COUNT (DISTINCT z) FROM foo GROUP BY x;	Each distinct x value The number of distinct y values for all tuples with the specific x value. The number of distinct z values in all tuples with the specific x value.
SELECT x, COUNT (DISTINCT y), COUNT (DISTINCT z), SUM (q), COUNT (r) FROM foo GROUP BY x;	Each distinct x value The number of distinct y values for all tuples with the specific x value. The number of distinct z values in all tuples with the specific x value. The sum of all q values in tuples with the specific x value

	The number of r values in tuples with the specific x value

COUNT(*)

COUNT(*) returns the number of rows in each group of the result set. The return value is a BIGINT.

Syntax

```
COUNT ( * )
```

Semantics

*	indicates that the count does not apply to any specific column or expression in the select list

Notes

- COUNT(*) requires a ***FROM Clause*** (page 176).

MAX

MAX returns the greatest value of an expression over a group of rows. The return value is the same as the expression data type.

Syntax

```
MAX ( [ ALL | DISTINCT ] expression )
```

Semantics

`ALL | DISTINCT`	are meaningless in this context
expression	(any numeric, string, or date/time type) contains at least one ***column reference*** (see "Column References" on page 46)

MIN

MIN returns the smallest value of an expression over a group of rows. The return value is the same as the expression data type.

Syntax

```
MIN ( [ ALL | DISTINCT ] expression )
```

Semantics

`ALL | DISTINCT`	are meaningless in this context

expression	(any numeric, string, or date/time type) contains at least one ***column reference*** (see "Column References" on page 46)

STDDEV

The non-standard function STDDEV is provided for compatibility with other databases. It is semantically identical to STDDEV_SAMP.

STDDEV_SAMP evaluates the statistical sample standard deviation for each member of the group.

```
STDDEV_SAMP(expression) = SQRT(VAR_SAMP(expression))
```

Syntax

```
STDDEV_SAMP(expression)
```

Semantics

expression	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

Examples

```
SELECT STDDEV_SAMP(b) FROM aggtest;
   stddev_samp
------------------
 57282.1641693241
```

(1 row)

STDDEV_POP

STDDEV_POP evaluates the statistical population standard deviation for each member of the group.

```
STDDEV_POP(expression) = SQRT(VAR_POP(expression))
```

Syntax

```
STDDEV_POP ( expression )
```

Semantics

expression	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

Examples

```
SELECT STDDEV_POP(b) FROM aggtest;
    stddev_pop
```

```
------------------
 64088.0121129418
(1 row)
```

STDDEV_SAMP

STDDEV_SAMP evaluates the statistical sample standard deviation for each member of the group.

`STDDEV_SAMP(`*`expression`*`) = SQRT(VAR_SAMP(`*`expression`*`))`

Syntax

`STDDEV_SAMP(`*`expression`*`)`

Semantics

expression	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

Examples

```
SELECT STDDEV_SAMP(b) FROM aggtest;
   stddev_samp
------------------
 57282.1641693241
(1 row)
```

SUM

SUM computes the sum of an expression over a group of rows. It returns a DOUBLE PRECISION value for a floating-point expression. Otherwise, the return value is the same as the expression data type.

Syntax

`SUM ( [ ALL | DISTINCT ] `*`expression`*` )`

Semantics

ALL	invokes the aggregate function for all rows in the group (default)
DISTINCT	invokes the aggregate function for all distinct non-null values of the expression found in the group
expression	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

Notes

- If you encounter overflow when using SUM, use ***SUM_FLOAT()*** (page 81) which converts to floating point.

SUM_FLOAT

SUM_FLOAT computes the sum of an expression over a group of rows. It returns a DOUBLE PRECISION value for the expression, regardless of the expression type.

Syntax

```
SUM_FLOAT ( [ ALL | DISTINCT ] expression )
```

Semantics

ALL	invokes the aggregate function for all rows in the group (default)
DISTINCT	invokes the aggregate function for all distinct non-null values of the expression found in the group
expression	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

VAR_POP

STDDEV_POP evaluates the statistical population variance for each member of the group. This is defined as the sum of squares of the difference of *expression* from the mean of *expression,* divided by the number of rows remaining.

Syntax

```
STDEV_POP ( expression )
```

Semantics

expression	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

Examples

```
SELECT VAR_POP(b) FROM aggtest;
     var_pop
------------------
 4107273296.58857
(1 row)
```

VAR_SAMP

STDDEV_SAMP evaluates the sample variance for each row of the group. This is defined as the sum of squares of the difference of *expression* from the mean of *expression,* divided by the number of rows remaining minus 1 (one).

Syntax

```
STDEV_SAMP ( expression )
```

Semantics

expression	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

Examples

```
SELECT VAR_SAMP(b) FROM aggtest;
    var_samp
-----------------
 3281246331.9214
(1 row)
```

VARIANCE

The non-standard function VARIANCE is provided for compatibility with other databases. It is semantically identical to VAR_SAMP.

STDDEV_SAMP evaluates the sample variance for each row of the group. This is defined as the sum of squares of the difference of *expression* from the mean of *expression,* divided by the number of rows remaining minus 1 (one).

Syntax

```
STDEV_SAMP ( expression )
```

Semantics

expression	(INTEGER, BIGINT, DOUBLE PRECISION, or INTERVAL) contains at least one ***column reference*** (see "Column References" on page 46)

Examples

```
SELECT VAR_SAMP(b) FROM aggtest;
    var_samp
-----------------
 3281246331.9214
```

(1 row)

Date/Time Functions

Documentation Notes

Functions that take TIME or TIMESTAMP inputs come in two variants:

- TIME WITH TIME ZONE or TIMESTAMP WITH TIME ZONE

TIME WITHOUT TIME ZONE or TIMESTAMP WITHOUT TIME ZONE For brevity, these variants are not shown separately.

The + and * operators come in commutative pairs (for example both DATE + INTEGER and INTEGER + DATE); we show only one of each such pair.

Daylight Savings Time Considerations

When adding an INTERVAL value to (or subtracting an INTERVAL value from) a TIMESTAMP WITH TIME ZONE value, the days component advances (or decrements) the date of the TIMESTAMP WITH TIME ZONE by the indicated number of days. Across daylight saving time changes (with the session time zone set to a time zone that recognizes DST), this means INTERVAL '1 day' does not necessarily equal INTERVAL '24 hours'.

For example, with the session time zone set to CST7CDT:

TIMESTAMP WITH TIME ZONE '2005-04-02 12:00-07' + INTERVAL '1 day'

produces

TIMESTAMP WITH TIME ZONE '2005-04-03 12:00-06'

while adding INTERVAL '24 hours' to the same initial TIMESTAMP WITH TIME ZONE produces

TIMESTAMP WITH TIME ZONE '2005-04-03 13:00-06',

as there is a change in daylight saving time at 2005-04-03 02:00 in time zone CST7CDT.

Date/Time Functions in Transactions

`CURRENT_TIMESTAMP` and related functions return the start time of the current transaction; their values do not change during the transaction. The intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp. However, `TIMEOFDAY( )` returns the wall-clock time and advances during transactions.

AGE

AGE returns an INTERVAL value representing the difference between two TIMESTAMP values.

Syntax

`AGE ( expression1 [ , expression2 ] )`

Semantics

expression1	(TIMESTAMP) specifies the beginning of the INTERVAL

expression2	(TIMESTAMP) specifies the end of the INTERVAL. The default is the ***CURRENT_DATE*** (page 84)

Examples

```
> SELECT AGE(TIMESTAMP '2001-04-10', TIMESTAMP '1957-06-13'); age
-------------------------
 43 years 9 mons 27 days
(1 row)

> SELECT AGE(TIMESTAMP '1957-06-13');
           age
-------------------------
 50 years 4 mons 13 days
(1 row)
```

CURRENT_DATE

CURRENT_DATE returns a value of type DATE representing today's date.

Syntax

```
CURRENT_DATE
```

Examples

```
> SELECT CURRENT_DATE;
    date
------------
 2007-10-26
(1 row)
```

CURRENT_TIME

CURRENT_TIME returns a value of type TIME WITH TIME ZONE representing the time of day.

Syntax

```
CURRENT_TIME [ ( precision ) ]
```

Semantics

precision	(INTEGER) causes the result to be rounded to the specified number of fractional digits in the seconds field.

Notes

- This function returns the start time of the current transaction; the value does not change during the transaction. The intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp.

Examples

```
> SELECT CURRENT_TIME;
       timetz
--------------------
 16:58:58.349569-04
(1 row)
```

CURRENT_TIMESTAMP

CURRENT_TIMESTAMP returns a value of type TIMESTAMP WITH TIME ZONE representing today's date and time of day.

Syntax

CURRENT_TIMESTAMP [(*precision*)]

Semantics

precision	(INTEGER) causes the result to be rounded to the specified number of fractional digits in the seconds field.

Notes

- This function returns the start time of the current transaction; the value does not change during the transaction. The intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp.

Examples

```
> SELECT CURRENT_TIMESTAMP;
          timestamptz
-------------------------------
 2007-10-26 16:58:58.349569-04
(1 row)
```

DATE_PART

The DATE_PART function is modeled on the traditional Ingres equivalent to the SQL-standard function EXTRACT:

Syntax

DATE_PART(*field* , *source*)

Semantics

field	a single-quoted string value that specifies the field to extract.
source	a ***date/time*** (page 63) expression

Notes

- The *field* parameter values are the same as EXTRACT (page 87).

Example

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

DATE_TRUNC

The function DATE_TRUNC is conceptually similar to the TRUNC function for numbers. The return value is of type `timestamp` or `interval` with all fields that are less significant than the selected one set to zero (or one, for day and month).

Syntax

```
DATE_TRUNC('field', source)
```

Semantics

field is a string constant that selects the precision to which truncate the input value. Valid values for *field* are:

```
century          milliseconds
day              minute
decade           month
hour             second
microseconds     week
millennium       year
```

source is a value expression of type TIMESTAMP or INTERVAL. (Values of type DATE and TIME are cast automatically, to TIMESTAMP or INTERVAL respectively.)

Examples

```
SELECT DATE_TRUNC('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00
SELECT DATE_TRUNC('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00
```

EXTRACT

The EXTRACT function retrieves subfields such as year or hour from date/time values and returns values of type `double precision (page 70).` It is primarily intended for computational processing rather than formatting date/time values for display.

Syntax

```
EXTRACT (field FROM source)
```

Semantics

field	is an identifier or string that selects what field to extract from the source value.
source	an expression of type DATE, TIMESTAMP, TIME, or INTERVAL. (Expressions of type DATE are cast to TIMESTAMP.)

ISFINITE

ISFINITE tests for the special TIMESTAMP constant INFINITY and returns a value of type BOOLEAN;

Syntax

```
ISFINITE(timestamp)
```

Semantics

timestamp	an expression of type TIMESTAMP

Examples

```
> SELECT ISFINITE(TIMESTAMP '2001-02-16 21:28:30'); isfinite
----------
 t
(1 row)

> SELECT ISFINITE(TIMESTAMP 'INFINITY');
 isfinite
----------
 f
(1 row)
```

LOCALTIME

LOCALTIME returns a value of type TIME representing the time of day.

Syntax

```
LOCALTIME [ ( precision ) ]
```

Semantics

precision	causes the result to be rounded to the specified number of fractional digits in the seconds field.

Notes

- This function returns the start time of the current transaction; the value does not change during the transaction. The intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp.

Examples

```
> SELECT LOCALTIME;
      time
-----------------
 16:58:58.349569
(1 row)
```

LOCALTIMESTAMP

LOCAL_TIMESTAMP returns a value of type TIMESTAMP representing today's date and time of day.

Syntax

```
LOCALTIMESTAMP [ ( precision ) ]
```

Semantics

precision	causes the result to be rounded to the specified number of fractional digits in the seconds field.

Notes

- This function returns the start time of the current transaction; the value does not change during the transaction. The intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp.

Examples

```
> SELECT LOCALTIMESTAMP;
         timestamp
----------------------------
 2007-10-26 16:58:58.349569
(1 row)
```

NOW

`NOW()` is equivalent to `CURRENT_TIMESTAMP` (page 85) except that it does not accept a precision parameter. It returns a value of type TIMESTAMP WITH TIME ZONE representing today's date and time of day.

Syntax

```
NOW()
```

Notes

- This function returns the start time of the current transaction; the value does not change during the transaction. The intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp.

Examples

```
> SELECT NOW();
              now
------------------------------
 2007-10-26 14:58:58.349569-06
(1 row)
```

OVERLAPS

The SQL `OVERLAPS` operator returns true when two time periods overlap, false when they do not overlap..

Syntax

```
( start, end ) OVERLAPS ( start, end )
```

(*start*, *length*) OVERLAPS (*start*, *length*)

Semantics

start	a DATE, TIME, or TIME STAMP value that specifies the beginning of a time period.
end	a DATE, TIME, or TIME STAMP value that specifies the end of a time period.
interval	an INTERVAL value that specifies the length of the time period.

Examples

```
SELECT (DATE '2001-02-16', DATE '2001-12-21')
  OVERLAPS (DATE '2001-10-30', DATE '2002-10-30');
Result: true
SELECT (DATE '2001-02-16', INTERVAL '100 days')
  OVERLAPS (DATE '2001-10-30', DATE '2002-10-30');
Result: false
```

TIMEOFDAY

TIMEOFDAY() returns a text string representing the time of day.

Syntax

```
TIMEOFDAY()
```

Notes

- TIMEOFDAY() returns the wall-clock time and advances during transactions.

Examples

```
> SELECT TIMEOFDAY();
              timeofday
-------------------------------------
 Fri Oct 26 18:10:45.637605 2007 EDT
(1 row)
```

Formatting Functions

These formatting functions provide a powerful set of tools for converting various data types (DATE/TIME, INTEGER, FLOATING POINT) to formatted strings and for converting from formatted strings to specific data types. These functions all follow a common calling convention: the **first argument is the value** to be formatted and the **second argument is a template** that defines the output or input format.

Exception: The `TO_TIMESTAMP` function can take a single `double precision` argument.

TO_CHAR

TO_CHAR converts various date/time and numeric values into text strings.

Syntax

`TO_CHAR ( expression, pattern )`

Semantics

expression	(TIMESTAMP, INTERVAL, INTEGER, DOUBLE PRECISION) specifies the value to convert
pattern	(CHAR or VARCHAR) specifies an output pattern string using the ***Template Patterns for Date/Time Formatting*** (page 96) and/or ***Template Patterns for Numeric Formatting*** (page 99).

Notes

- Ordinary text is allowed in TO_CHAR templates and is output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains pattern key words. For example, in '"Hello Year "YYYY', the YYYY is replaced by the year data, but the single Y in Year is not.
- TO_CHAR's day of the week numbering (see the 'D' ***template pattern*** (page 96)) is different from that of the ***EXTRACT*** (page 87) function
- Given an INTERVAL type, TO_CHAR formats HH and HH12 as hours in a single day, while HH24 can output hours exceeding a single day, for example, >24
- To use a double quote character in the output, precede it with a double backslash. This is necessary because the backslash already has a special meaning in a string constant. For example: `'\\"YYYY Month\\"'`
- TO_CHAR does not support the use of V combined with a decimal point. For example: 99.9V99 is not allowed.

Examples

Expression	Result

TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
TO_CHAR(CURRENT_TIMESTAMP, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
TO_CHAR(-0.1, '99.99')	' -.10'
TO_CHAR(-0.1, 'FM9.99')	'-.1'
TO_CHAR(0.1, '0.9')	' 0.1'
TO_CHAR(12, '9990999.9')	' 0012.0'
TO_CHAR(12, 'FM9990999.9')	'0012.'
TO_CHAR(485, '999')	' 485'
TO_CHAR(-485, '999')	'-485'
TO_CHAR(485, '9 9 9')	' 4 8 5'
TO_CHAR(1485, '9,999')	' 1,485'
TO_CHAR(1485, '9G999')	' 1 485'
TO_CHAR(148.5, '999.999')	' 148.500'
TO_CHAR(148.5, 'FM999.999')	'148.5'
TO_CHAR(148.5, 'FM999.990')	'148.500'
TO_CHAR(148.5, '999D999')	' 148,500'
TO_CHAR(3148.5, '9G999D999')	' 3 148,500'
TO_CHAR(-485, '999S')	'485-'
TO_CHAR(-485, '999MI')	'485-'
TO_CHAR(485, '999MI')	'485 '
TO_CHAR(485, 'FM999MI')	'485'
TO_CHAR(485, 'PL999')	'+485'
TO_CHAR(485, 'SG999')	'+485'

TO_CHAR(-485, 'SG999')	'-485'
TO_CHAR(-485, '9SG99')	'4-85'
TO_CHAR(-485, '999PR')	'<485>'
TO_CHAR(485, 'L999')	'DM 485
TO_CHAR(485, 'RN')	' CDLXXXV'
TO_CHAR(485, 'FMRN')	'CDLXXXV'
TO_CHAR(5.2, 'FMRN')	'V'
TO_CHAR(482, '999th')	' 482nd'
TO_CHAR(485, '"Good number:"999')	'Good number: 485'
TO_CHAR(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
TO_CHAR(12, '99V999')	' 12000'
TO_CHAR(12.4, '99V999')	' 12400'
TO_CHAR(12.45, '99V9')	' 125'

TO_DATE

TO_DATE converts a string value to a DATE type.

Syntax

TO_DATE (*expression* , *pattern*)

Semantics

expression	(CHAR or VARCHAR) specifies the value to convert
pattern	(CHAR or VARCHAR) specifies an output pattern string using the ***Template Patterns for Date/Time Formatting*** (page 96) and/or ***Template Patterns for Numeric Formatting*** (page 99).

Notes

- To use a double quote character in the output, precede it with a double backslash. This is necessary because the backslash already has a special meaning in a string constant. For example: '\\"YYYY Month\\"'

- TO_TIMESTAMP and TO_DATE skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template. For example TO_TIMESTAMP('2000 JUN', 'YYYY MON') is correct, but TO_TIMESTAMP('2000 JUN', 'FXYYYY MON') returns an error, because TO_TIMESTAMP expects one space only.
- The YYYY conversion from string to TIMESTAMP or DATE has a restriction if you use a year with more than four digits. You must use some non-digit character or template after YYYY, otherwise the year is always interpreted as four digits. For example (with the year 20000): to_date('200001131', 'YYYYMMDD') will be interpreted as a four-digit year; instead use a non-digit separator after the year, like to_date('20000-1131', 'YYYY-MMDD') or to_date('20000Nov31', 'YYYYMonDD').
- In conversions from string to TIMESTAMP or DATE, the CC field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y then the year is computed as (CC-1)*100+YY.

Examples

```
TO_DATE('05 Dec 2000', 'DD Mon YYYY')
```

See Also

Template Pattern Modifiers for Date/Time Formatting (page 98)

TO_TIMESTAMP

The `TO_TIMESTAMP` function converts a string value or a UNIX/POSIX epoch value to a `TIMESTAMP WITH TIME ZONE` type.

Syntax

```
TO_TIMESTAMP ( { expression, pattern | unix-epoch } )
```

Semantics

expression	(CHAR or VARCHAR) is the string to convert
pattern	(CHAR or VARCHAR) specifies an output pattern string using the ***Template Patterns for Date/Time Formatting*** (page 96) and/or ***Template Patterns for Numeric Formatting*** (page 99).
unix-epoch	(DOUBLE PRECISION) specifies some number of seconds elapsed since midnight UTC of January 1, 1970, not counting leap seconds. `INTEGER` values are implicitly cast to `DOUBLE PRECISION`.

Notes

- For more information about UNIX/POSIX time, see the ***Wikipedia*** http://en.wikipedia.org/wiki/Unix_time.

- Millisecond (MS) and microsecond (US) values in a conversion from string to TIMESTAMP are used as part of the seconds after the decimal point. For example TO_TIMESTAMP('12:3', 'SS:MS') is not 3 milliseconds, but 300, because the conversion counts it as 12 + 0.3 seconds. This means for the format SS:MS, the input values 12:3, 12:30, and 12:300 specify the same number of milliseconds. To get three milliseconds, one must use 12:003, which the conversion counts as 12 + 0.003 = 12.003 seconds.
- Here is a more complex example: TO_TIMESTAMP('15:12:02.020.001230', 'HH:MI:SS.MS.US') is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.
- To use a double quote character in the output, precede it with a double backslash. This is necessary because the backslash already has a special meaning in a string constant. For example: `'\\"YYYY Month\\"'`
- TO_TIMESTAMP and TO_DATE skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template. For example TO_TIMESTAMP('2000 JUN', 'YYYY MON') is correct, but TO_TIMESTAMP('2000 JUN', 'FXYYYY MON') returns an error, because TO_TIMESTAMP expects one space only.
- The YYYY conversion from string to TIMESTAMP or DATE has a restriction if you use a year with more than four digits. You must use some non-digit character or template after YYYY, otherwise the year is always interpreted as four digits. For example (with the year 20000): to_date('200001131', 'YYYYMMDD') will be interpreted as a four-digit year; instead use a non-digit separator after the year, like to_date('20000-1131', 'YYYY-MMDD') or to_date('20000Nov31', 'YYYYMonDD').
- In conversions from string to TIMESTAMP or DATE, the CC field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y then the year is computed as (CC-1)*100+YY.

Examples

```
TO_TIMESTAMP('05 Dec 2000', 'DD Mon YYYY')
TO_TIMESTAMP(200120400)
```

See Also

Template Pattern Modifiers for Date/Time Formatting (page 98)

TO_NUMBER

TO_NUMBER converts a string value to DOUBLE PRECISION.

Syntax

```
TO_NUMBER ( expression, pattern )
```

Semantics

expression	(CHAR or VARCHAR) specifies the string to convert
pattern	(CHAR or VARCHAR) specifies an output pattern string using the

	Template Patterns for Date/Time Formatting (page 96) and/or ***Template Patterns for Numeric Formatting*** (page 99).

Notes

- To use a double quote character in the output, precede it with a double backslash. This is necessary because the backslash already has a special meaning in a string constant. For example: `'\\"YYYY Month\\"'`

Examples

```
> SELECT TO_CHAR(1999, 'rn'), TO_NUMBER('mcmxcix', 'rn'); to_char     |
to_number
-----------------+-----------
         mcmxcix |      1999
```

Template Patterns for Date/Time Formatting

In an output template string (for `TO_CHAR`), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for anything other than `TO_CHAR`), template patterns identify the parts of the input data string to be looked at and the values to be found there.

Certain modifiers can be applied to any template pattern to alter its behavior as described in ***Template Pattern Modifiers for Date/Time Formatting*** (page 98).

Pattern	Description
`HH`	hour of day (01-12)
`HH12`	hour of day (01-12)
`HH24`	hour of day (00-23)
`MI`	minute (00-59)
`SS`	second (00-59)
`MS`	millisecond (000-999)
`US`	microsecond (000000-999999)
`SSSS`	seconds past midnight (0-86399)
`AM` or `A.M.` or `PM` or `P.M.`	meridian indicator (uppercase)
`am` or `a.m.` or `pm` or	meridian indicator (lowercase)

p.m.	
Y,YYY	year (4 and more digits) with comma
YYYY	year (4 and more digits)
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
IYYY	ISO year (4 and more digits)
IYY	last 3 digits of ISO year
IY	last 2 digits of ISO year
I	last digits of ISO year
BC or B.C. or AD or A.D.	era indicator (uppercase)
bc or b.c. or ad or a.d.	era indicator (lowercase)
MONTH	full uppercase month name (blank-padded to 9 chars)
Month	full mixed-case month name (blank-padded to 9 chars)
month	full lowercase month name (blank-padded to 9 chars)
MON	abbreviated uppercase month name (3 chars)
Mon	abbreviated mixed-case month name (3 chars)
mon	abbreviated lowercase month name (3 chars)
MM	month number (01-12)
DAY	full uppercase day name (blank-padded to 9 chars)
Day	full mixed-case day name (blank-padded to 9 chars)
day	full lowercase day name (blank-padded to 9 chars)
DY	abbreviated uppercase day name (3 chars)

Dy	abbreviated mixed-case day name (3 chars)
dy	abbreviated lowercase day name (3 chars)
DDD	day of year (001-366)
DD	day of month (01-31)
D	day of week (1-7; Sunday is 1)
W	week of month (1-5) (The first week starts on the first day of the month.)
WW	week number of year (1-53) (The first week starts on the first day of the year.)
IW	ISO week number of year (The first Thursday of the new year is in week 1.)
CC	century (2 digits)
J	Julian Day (days since January 1, 4712 BC)
Q	quarter
RM	month in Roman numerals (I-XII; I=January) (uppercase)
rm	month in Roman numerals (i-xii; i=January) (lowercase)
TZ	time-zone name (uppercase)
tz	time-zone name (lowercase)

Template Pattern Modifiers for Date/Time Formatting

Certain modifiers can be applied to any template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier.

Modifier	Description	Example
FM prefix	fill mode (suppress padding blanks and zeroes)	FMMonth
TH suffix	uppercase ordinal number suffix	DDTH
th suffix	lowercase ordinal number suffix	DDth

FX prefix	fixed format global option	FX Month DD Day

Notes

- FM suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern be fixed-width.

Template Patterns for Numeric Formatting

Pattern	Description
9	value with the specified number of digits
0	value with leading zeros
. (period)	decimal point
, (comma)	group (thousand) separator
PR	negative value in angle brackets
S	sign anchored to number (uses locale)
L	currency symbol (uses locale)
D	decimal point (uses locale)
G	group separator (uses locale)
MI	minus sign in specified position (if number < 0)
PL	plus sign in specified position (if number > 0)
SG	plus/minus sign in specified position
RN	roman numeral (input between 1 and 3999)
TH or th	ordinal number suffix
V	shift specified number of digits (see notes)
EEEE	scientific notation (not implemented yet)

Notes

- A sign formatted using SG, PL, or MI is not anchored to the number; for example, TO_CHAR(-12, 'S9999') produces ' -12', but TO_CHAR(-12, 'MI9999') produces '- 12'. Oracle does not allow the use of MI ahead of 9, but rather requires that 9 precede MI.

- 9 results in a value with the same number of digits as there are 9s. If a digit is not available it outputs a space.
- TH does not convert values less than zero and does not convert fractional numbers.
- PL, SG, and TH are PostgreSQL extensions.
- V effectively multiplies the input values by 10^*n*, where *n* is the number of digits following V. TO_CHAR does not support the use of V combined with a decimal point. For example: 99.9V99 is not allowed.

Mathematical Functions

Some of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with DOUBLE PRECISION (page 70) data may vary in accuracy and behavior in boundary cases depending on the host system.

ABS

ABS returns the absolute value of the argument. The return value has the same data type as the argument..

Syntax

ABS (*expression*)

Semantics

expression	is a value of type INTEGER or DOUBLE PRECISION

Examples

ABS(-17.4)

Result: 17.4

ACOS

ACOS returns a DOUBLE PRECISION value representing the trigonometric inverse cosine of the argument.

Syntax

ACOS (*expression*)

Semantics

expression	is a value of type DOUBLE PRECISION

ASIN

ASIN returns a DOUBLE PRECISION value representing the trigonometric inverse sine of the argument.

Syntax

```
ASIN ( expression )
```

Semantics

expression	is a value of type DOUBLE PRECISION

ATAN

ATAN returns a DOUBLE PRECISION value representing the trigonometric inverse tangent of the argument.

Syntax

```
ATAN ( expression )
```

Semantics

expression	is a value of type DOUBLE PRECISION

ATAN2

ATAN2 returns a DOUBLE PRECISION value representing the trigonometric inverse tangent of the arithmetic dividend of the arguments.

Syntax

```
ATAN2 ( quotient, divisor )
```

Semantics

quotient	is an expression of type DOUBLE PRECISION representing the quotient
divisor	is an expression of type DOUBLE PRECISION representing the divisor

CBRT

CBRT returns the cube root of the argument. The return value has the type DOUBLE PRECISION.

Syntax

```
CBRT ( expression )
```

Semantics

expression	is a value of type DOUBLE PRECISION

Examples

CBRT(27.0)

Result: 3

CEILING (CEIL)

CEIL returns the smallest integer not less than then argument. The return data type is the same as the argument.

Syntax

```
CEILING ( expression )
CEIL ( expression )
```

Semantics

expression	is a value of type INTEGER or DOUBLE PRECISION

Examples

```
CEIL(-42.8)
```

Result: `-42`

COS

COS returns a DOUBLE PRECISION value representing the trigonometric cosine of the argument.

Syntax

```
COS ( expression )
```

Semantics

expression	is a value of type DOUBLE PRECISION

COT

COT returns a DOUBLE PRECISION value representing the trigonometric cotangent of the argument.

Syntax

```
COT ( expression )
```

Semantics

expression	is a value of type DOUBLE PRECISION

DEGREES

DEGREES converts an expression from radians to degrees. The return value has the type DOUBLE PRECISION.

Syntax

```
DEGREES ( expression )
```

Semantics

expression	is a value of type DOUBLE PRECISION

Examples

```
DEGREES(0.5)
```

Result: `28.6478897565412`

EXP

EXP returns the exponential function, e to the power of a number. The return value has the same data type as the argument.

Syntax

```
EXP ( exponent )
```

Semantics

exponent	is an expression of type INTEGER or DOUBLE PRECISION

Examples

EXP(1.0)

Result: 2.71828182845905

FLOOR

FLOOR returns the largest integer not greater than argument. The return data type is the same as the argument.

Syntax

FLOOR (*expression*)

Semantics

expression	is an expression of type INTEGER or DOUBLE PRECISION.

Examples

FLOOR(-42.8)

Result: -43

HASH

HASH calculates a hash value over its arguments, producing a value in the range 0 <= x < 263.

Syntax

HASH (*expression* [,...)

Semantics

expression	is an expression of any data type. For the purpose of hash segmentation, each expression is a ***column reference*** (see "Column References" on page 46).

Notes

- When used for hash segmentation, each expression must refer to a column of the projection.

Examples

HASH (C1, C2)

LN

LN returns the natural logarithm of the argument. The return data type is the same as the argument.

Syntax

```
LN ( expression )
```

Semantics

expression	is an expression of type INTEGER or DOUBLE PRECISION

Examples

```
LN(2.0)
```

Result: `0.693147180559945`

LOG

LOG returns the logarithm to the specified base of the argument. The return data type is the same as the argument.

Syntax

```
LOG ( [ base, ] expression )
```

Semantics

base	specifies the base (default is base 10)
expression	is an expression of type INTEGER or DOUBLE PRECISION

Examples

```
LOG(100.0)
```

Result: 2

```
LOG(2.0, 64.0)
```

Result: `6.0000000000`

MOD

MOD (modulo) returns the remainder of a division operation. The return data type is the same as the arguments.

Syntax

```
MOD ( expression1, expression2 )
```

Semantics

expression1	specifies the dividend (INTEGER or DOUBLE PRECISION)
expression2	specifies the divisor (type same as dividend)

Notes

- The dividend is the quantity to be divided. For example:
  `6/2 = 3`
  the dividend is 6. The divisor is 2.

Examples

```
MOD(9,4)
```

Result: 1

PI

PI returns the constant pi (Π), the ratio of any circle's circumference to its diameter in Euclidean geometry The return type is DOUBLE PRECISION.

Syntax

```
PI()
```

Examples

```
PI()
```

Result: `3.14159265358979`

POWER

POWER returns a DOUBLE PRECISION value representing one number raised to the power of another number.

Syntax

```
POWER ( expression1, expression2 )
```

Semantics

expression1	is an expression of type DOUBLE PRECISION that represents the base
expression2	is an expression of type DOUBLE PRECISION that represents the exponent

Examples

```
POWER(9.0, 3.0)
```

Result: 729

RADIANS

RADIANS returns a DOUBLE PRECISION value representing an angle expressed in degrees converted to radians.

Syntax

```
RADIANS ( expression )
```

Semantics

expression	is an expression of type DOUBLE PRECISION representing degrees

Examples

```
RADIANS(45.0)
```

Result: `0.785398163397448`

ROUND

The ROUND function rounds off a value to a specified number of decimal places. Fractions greater than or equal to .5 are rounded up. Fractions less than .5 are rounded down (truncated).

Syntax

```
ROUND ( expression [ , decimal-places ] )
```

Semantics

expression	is an expression of type DOUBLE PRECISION
decimal-places	if positive, specifies the number of decimal places to display to the right of the

	decimal point; if negative, specifies the number of decimal places to display to the left of the decimal point.

Notes

- The ROUND function rounds off except in the case of a decimal constant with more than 15 decimal places. For example:

```
> SELECT ROUND(3.499999999999999); -- 15 decimal places round
-------
      3
(1 row)
```

  The internal integer representation used to compute the ROUND function causes the fraction to be evaluated precisely and it is thus rounded down. However:

```
> SELECT ROUND(3.4999999999999999); -- 16 decimal places round
-------
      4
(1 row)
```

  The internal floating point representation used to compute the ROUND function causes the fraction to be evaluated as 3.5, which is rounded up.

Examples

```
SELECT ROUND(3.14159, 3);
3.142
SELECT ROUND(1234567, -3);
1235000
SELECT ROUND(3.4999, -1);
 0
```

SIGN

SIGN returns a DOUBLE PRECISION value of -1, 0, or 1 representing the arithmetic sign of the argument.

Syntax

```
SIGN ( expression )
```

Semantics

expression	is an expression of type DOUBLE PRECISION

Examples

```
SIGN(-8.4)
```

Result: `-1`

SIN

SIN returns a DOUBLE PRECISION value representing the trigonometric sine of the argument.

Syntax

```
SIN ( expression )
```

Semantics

expression	is an expression of type DOUBLE PRECISION

SQRT

SQRT returns a DOUBLE PRECISION value representing the arithmetic square root of the argument.

Syntax

```
SQRT ( expression )
```

Semantics

expression	is an expression of type DOUBLE PRECISION

Examples

```
SQRT(2.0)
```

Result: `1.4142135623731`

TAN

TAN returns a DOUBLE PRECISION value representing the trigonometric tangent of the argument.

Syntax

```
TAN ( expression )
```

Semantics

expression	is an expression of type DOUBLE PRECISION

TRUNC

TRUNC returns a value representing the argument fully truncated (toward zero) or truncated to a specific number of decimal places.

Syntax

```
TRUNC ( expression [ , places ]
```

Semantics

expression	is an expression of type INTEGER or DOUBLE PRECISION that represents the number to truncate
places	is an expression of type INTEGER specifies the number of decimal places to return

Examples

```
TRUNC(42.8)
```

Result: 42

```
TRUNC(42.4382, 2)
```

Result: 42.43

String Functions

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of all the types CHARACTER and CHARACTER VARYING. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of the automatic padding when using the CHARACTER type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first. Some functions also exist natively for the bit-string types.

The string functions not handle multibyte UTF-8 sequences correctly. They treat each byte as a character.

BTRIM

BTRIM removes the longest string consisting only of specified characters from the start and end of a string.

Syntax

```
BTRIM ( expression [ , characters-to-remove ] )
```

Semantics

expression	(CHAR or VARCHAR) is the string to modify
characters-to-remove	(CHAR or VARCHAR) specifies the characters to remove. The default is the space character.

Examples

```
BTRIM('xyxtrimyyx', 'xy')
```

Result: `trim`

CHARACTER_LENGTH

CHARACTER_LENGTH returns an INTEGER value representing the number of characters in a string. It strips the padding from CHAR expressions but not from VARCHAR expressions.

Syntax

```
[ CHAR_LENGTH | CHARACTER_LENGTH ] ( expression )
```

Semantics

expression	(CHAR or VARCHAR) is the string to measure

Notes

- CHARACTER_LENGTH is identical to ***LENGTH*** (page 112).
- See also ***OCTET_LENGTH*** (page 115).

Examples

```
> SELECT CHAR_LENGTH('1234  '::CHAR(10));
 char_length
-------------
           4
(1 row)
> SELECT CHAR_LENGTH('1234  '::VARCHAR(10));
 char_length
-------------
           6
(1 row)
> SELECT CHAR_LENGTH(NULL::CHAR(10)) IS NULL;
 ?column?
----------
 t
(1 row)
```

CLIENT_ENCODING

CLIENT_ENCODING returns a VARCHAR value representing the character set encoding of the client system.

Syntax

```
CLIENT_ENCODING()
```

Notes

- Vertica supports the UTF-8 character set.
- CLIENT_ENCODING returns the same value as the vsql meta-command \encoding and variable ENCODING

Examples

```
> SELECT CLIENT_ENCODING();
 client_encoding
-----------------
 UTF-8
(1 row)
```

LENGTH

LENGTH returns an INTEGER value representing the number of characters in a string. It strips the padding from CHAR expressions but not from VARCHAR expressions.

Syntax

```
LENGTH ( expression )
```

Semantics

expression	(CHAR or VARCHAR) is the string to measure

Notes

- LENGTH is identical to ***CHARACTER_LENGTH*** (page 111).
- See also ***OCTET_LENGTH*** (page 115).

Examples

```
> SELECT LENGTH('1234  '::CHAR(10));
 length
--------
      4
(1 row)
> SELECT LENGTH('1234  '::VARCHAR(10));
 length
--------
      6
(1 row)
> SELECT LENGTH(NULL::CHAR(10)) IS NULL;
 ?column?
----------
 t
(1 row)
```

LOWER

LOWER returns a VARCHAR value containing the argument converted to lower case letters.

Syntax

```
LOWER ( expression )
```

Semantics

expression	(CHAR or VARCHAR) is the string to convert

Examples

```
> SELECT LOWER('AbCdEfG');
  lower
----------
 abcdefg
(1 row)
```

LPAD

LPAD returns a VARCHAR value representing a string of a specific length filled on the left with specific characters.

Syntax

```
LPAD( expression, length [ , fill ] )
```

Semantics

expression	(CHAR OR VARCHAR) specifies the string to fill
length	(INTEGER) specifies the number of characters to return
fill	(CHAR OR VARCHAR) specifes the repeating string of characters with which to fill the output string. The default is the space character.

Notes

- If the string is already longer than the specified length it is truncated on the right.

Examples

```
> SELECT LPAD('database', 15, 'xzy');
      lpad
-----------------
 xzyxzyxdatabase
(1 row)
```

LTRIM

LTRIM returns a VARCHAR value representing a string with specific characters having been removed from the left side (beginning).

Syntax

```
LTRIM ( expression [ , characters ] )
```

Semantics

expression	(CHAR or VARCHAR) is the string to trim
characters	(CHAR or VARCHAR) specifies the characters to remove from the left side of *expression*. The default is the space character.

Examples

```
> SELECT LTRIM('zzzyyyyyyyxxxxxxxxtrim', 'xyz'); ltrim
-------
 trim
(1 row)
```

OCTET_LENGTH

LENGTH returns an INTEGER value representing the true number of bytes in a string.

Syntax

```
LENGTH ( expression )
```

Semantics

expression	(CHAR or VARCHAR) is the string to measure

Notes

- See also ***CHARACTER_LENGTH*** (page 111).

Examples

```
> SELECT OCTET_LENGTH('1234  '::CHAR(10));
 octet_length
--------------
           10
(1 row)
> SELECT OCTET_LENGTH('1234  '::VARCHAR(10));
 octet_length
--------------
            6
(1 row)
> SELECT OCTET_LENGTH(NULL::CHAR(10)) IS NULL;
 ?column?
----------
 t
(1 row)
```

OVERLAY

OVERLAY returns a VARCHAR value representing a string having had a substring replaced by another string.

Syntax

```
OVERLAY ( expression1 PLACING expression2 FROM position [ FOR extent ] )
```

Semantics

expression1	(CHAR or VARCHAR) is the string to process
expression2	(CHAR or VARCHAR) is the substring to overlay
position	(INTEGER) is the character position (counting from one) at which to begin the overlay
extent	(INTEGER) specifies the number of characters to replace with the overlay

Examples

```
=> \a
Output format is unaligned.

=> SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2);
overlay
1xxx56789
(1 row)

=> SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2 FOR 4);
overlay
1xxx6789
(1 row)

=> SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2 FOR 5);
overlay
1xxx789
(1 row)

=> SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2 FOR 6);
overlay
1xxx89
(1 row)
```

POSITION

POSITION returns an INTEGER values representing the location of a specified substring with a string (counting from one).

Syntax

```
POSITION ( substring IN string )
```

Semantics

substring	(CHAR or VARCHAR) is the substring to locate
string	(CHAR or VARCHAR) is the string in which to locate the substring

Notes

POSITION is identical to ***STRPOS*** (page 119) except for the order of the arguments.

Examples

```
=> SELECT POSITION('3' IN '1234');
 position
----------
        3
(1 row)
```

REPEAT

REPEAT returns a VARCHAR value containing a specified string repeated a specified number of times.

Syntax

```
REPEAT ( string , repetitions )
```

Semantics

string	(CHAR or VARCHAR) is the string to repeat
repetitions	(INTEGER) is the number of times to repeat the string

Examples

```
=> SELECT REPEAT ('1234', 5);
        repeat
----------------------
 12341234123412341234
(1 row)
```

REPLACE

Replace all occurrences in string of substring from with substring to

Syntax

```
REPLACE ( string , target , replacement )
```

Semantics

string	(CHAR OR VARCHAR) is the string to which to perform the replacement
target	(CHAR OR VARCHAR) is the string to replace
replacement	(CHAR OR VARCHAR) is the string with which to replace the target

Examples

```
> SELECT REPLACE('Documentation%20Library','%20',' '); replace
-----------------------
 Documentation Library
(1 row)
```

RPAD

RPAD returns a VARCHAR value representing a string of a specific length filled on the right with specific characters.

Syntax

```
RPAD ( expression , length [ , fill ] )
```

Semantics

expression	(CHAR OR VARCHAR) specifies the string to fill
length	(INTEGER) specifies the number of characters to return
fill	(CHAR OR VARCHAR) specifes the repeating string of characters with which to fill the output string. The default is the space character.

Notes

- If the string is already longer than the specified length it is truncated on the right.

Examples

```
> SELECT RPAD('database', 15, 'xzy');
      rpad
-----------------
 databasexzyxzyx
(1 row)
```

RTRIM

RTRIM returns a VARCHAR value representing a string with specific characters having been removed from the right side (end).

Syntax

```
RTRIM ( expression [ , characters ] )
```

Semantics

expression	(CHAR or VARCHAR) is the string to trim
characters	(CHAR or VARCHAR) specifies the characters to remove from the right side of *expression*. The default is the space character.

Examples

```
> SELECT RTRIM('trimzzzyyyyyyxxxxxxx', 'xyz'); ltrim
-------
```

```
 trim
(1 row)
```

STRPOS

STRPOS returns an INTEGER values representing the location of a specified substring with a string (counting from one).

Syntax

```
STRPOS ( string , substring )
```

Semantics

string	(CHAR or VARCHAR) is the string in which to locate the substring
substring	(CHAR or VARCHAR) is the substring to locate

Notes

- STRPOS is identical to ***POSITION*** (page 116) except for the order of the arguments.

Examples

```
=> SELECT STRPOS('1234','3');
 strpos
--------
      3
(1 row)
```

SUBSTR

SUBSTR returns a VARCHAR value representing a substring of a specified string.

Syntax

```
SUBSTR (string , position [ , extent ] )
```

Semantics

string	(CHAR or VARCHAR) is the string from which to extract a substring
position	(INTEGER) is the starting position of the substring (counting from one)
extent	(INTEGER) is the length of the substring to extract. The default is the end of the string.

Notes

- ***SUBSTRING*** (page 120) performs the same function as SUBSTR. The only difference is the syntax allowed.

Examples

```
=>  SELECT SUBSTR('123456789', 3, 2);
 substr
--------
 34
(1 row)

=>  SELECT SUBSTR('123456789', 3);
 substr
---------
 3456789
(1 row)
```

SUBSTRING

SUBSTRING returns a VARCHAR value representing a substring of a specified string.

Syntax

```
SUBSTRING (string , position [ , extent ] )
SUBSTRING (string FROM position [ FOR extent ] )
```

Semantics

string	(CHAR or VARCHAR) is the string from which to extract a substring
position	(INTEGER) is the starting position of the substring (counting from one)
extent	(INTEGER) is the length of the substring to extract. The default is the end of the string.

Notes

- ***SUBSTR*** (page 119) performs the same function as SUBSTRING. The only difference is the syntax allowed.

Examples

```
=>  SELECT SUBSTRING('123456789', 3, 2);
 substring
-----------
 34
(1 row)

=>  SELECT SUBSTRING('123456789' FROM 3 FOR 2); substring
-----------
 34
(1 row)
```

TO_HEX

TO_HEX returns a VARCHAR representing the hexadecimal equivalent of a number.

Syntax

```
TO_HEX ( number )
```

Semantics

number	(INTEGER) is the number to convert to hexadecimal

Examples

```
=> SELECT TO_HEX(123456789);
 to_hex
---------
 75bcd15
(1 row)
```

TRIM

TRIM combines the ***BTRIM*** (on page 111), ***LTRIM*** (on page 114), and ***RTRIM*** (on page 118) functions into a single function.

Syntax

```
TRIM ( [ [ LEADING | TRAILING | BOTH ] characters FROM ] expression )
```

Semantics

LEADING	removes the specified characters from the left side of the string
TRAILING	removes the specified characters from the right side of the string
BOTH	removes the specified characters from both sides of the string (default)
characters	(CHAR or VARCHAR) specifies the characters to remove from *expression*. The default is the space character.
expression	(CHAR or VARCHAR) is the string to trim

Examples

```
Retail_Schema=> SELECT '-' || TRIM(LEADING 'x' FROM 'xxdatabasexx') || '-';
   ?column?
--------------
 -databasexx-
(1 row)
Retail_Schema=> SELECT '-' || TRIM(TRAILING 'x' FROM 'xxdatabasexx') || '-';
```

```
   ?column?
--------------
 -xxdatabase-
(1 row)
Retail_Schema=> SELECT '-' || TRIM(BOTH 'x' FROM 'xxdatabasexx') || '-';
  ?column?
------------
 -database-
(1 row)
Retail_Schema=> SELECT '-' || TRIM('x' FROM 'xxdatabasexx') || '-';
  ?column?
------------
 -database-
(1 row)
Retail_Schema=> SELECT '-' || TRIM(LEADING FROM '  database  ') || '-';
   ?column?
--------------
 -database  -
(1 row)
Retail_Schema=> SELECT '-' || TRIM('  database  ') || '-';
  ?column?
------------
 -database-
(1 row)
```

UPPER

UPPER returns a VARCHAR value containing the argument converted to upper case letters.

Syntax

```
UPPER ( expression )
```

Semantics

expression	(CHAR or VARCHAR) is the string to convert

Examples

```
> SELECT UPPER('AbCdEfG');
  upper
----------
 ABCDEFG
(1 row)
```

System Information Functions

CURRENT_DATABASE

CURRENT_DATABASE returns a VARCHAR value containing the name of the database to which you are connected.

Syntax

```
CURRENT_DATABASE()
```

Examples

```
=> SELECT CURRENT_DATABASE;
 current_database
------------------
 RETAIL_SCHEMA
(1 row)
```

CURRENT_SCHEMA

CURRENT_SCHEMA returns the name of the current schema.

Syntax

```
CURRENT_SCHEMA()
```

Examples

```
QATESTDB=> SELECT CURRENT_SCHEMA();
 current_schema
----------------
 public
(1 row)
```

CURRENT_USER

CURRENT_USER returns a VARCHAR containing the name of the user who initiated the current database connection.

Syntax

```
CURRENT_USER
```

Notes

- This function must be called without trailing parentheses.
- CURRENT_USER is useful for permission checking

- CURRENT_USER is equivalent to ***SESSION_USER*** (page 124) and ***USER*** (page 125).

Examples

```
=> SELECT CURRENT_USER;
 current_user
--------------
 dbadmin
(1 row)
```

HAS_TABLE_PRIVILEGE

HAS_TABLE_PRIVILEGE returns a true/false value indicating whether or not a user can access a table in a particular way.

Syntax

```
HAS_TABLE_PRIVILEGE ( [ user, ] table, privilege )
```

Semantics

user	specifies the name of a database user. The default is the ***CURRENT_USER*** (on page 123).	
table	specifies the name of a table in the logical schema.	
privilege	SELECT	Allows the user to SELECT from any column of the specified table.
	INSERT	Allows the user to INSERT tuples into the specified table and to use the ***COPY*** (page 143) command to load the table.
	UPDATE	Allows the user to UPDATE tuples in the specified table.
	DELETE	Allows DELETE of a row from the specified table.
	REFERENCES	To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

Notes

- All arguments must be quoted ***string constants*** (page 33).

Examples

```
SELECT HAS_TABLE_PRIVILEGE('mytable', 'SELECT');
```

SESSION_USER

SESSION_USER is equivalent to ***CURRENT_USER*** (on page 123).

USER

USER is equivalent to ***CURRENT_USER*** (on page 123).

VERSION

VERSION returns a VARCHAR containing a Vertica node's version information

Syntax

```
VERSION()
```

Examples

```
=> SELECT VERSION();
                version
----------------------------------------
 Vertica_Database v2.0.0-20080104190005
(1 row)
```

Vertica Functions

ADVANCE_EPOCH

ADVANCE_EPOCH manually closes the current epoch and begins a new epoch. Use ADVANCE_EPOCH immediately before using ***ALTER PROJECTION*** (page 138) MOVEOUT.

Syntax

```
SELECT ADVANCE_EPOCH()
```

ANALYZE_STATISTICS

ANALYZE_STATISTICS collects and aggregates data samples and storage information from all nodes on which a projection is stored, then writes statistics into the catalog so that they can be used by the query optimizer. Without these statistics, the query optimizer would assume uniform distribution of data values and equal storage usage for all projections.

Syntax

```
SELECT ANALYZE_STATISTICS ( 'projection' )
```

Semantics

projection	specifies the name of the projection.

DISPLAY_LICENSE

DISPLAY_LICENSE returns license information.

description

Syntax

```
SELECT DISPLAY_LICENSE()
```

Examples

```
=> SELECT DISPLAY_LICENSE();
                  display_license
-----------------------------------------------------
 Vertica Systems, Inc.
1/1/2008
12/31/2008
30
50TB

(1 row)
```

DUMP_CATALOG

DUMP_CATALOG check for deadlocked clients and the resources they are waiting for.

Syntax

```
SELECT DUMP_CATALOG()
```

Notes

- Use DUMP_CATALOG if Vertica seems to be hung
- Send the output to ***Technical Support*** (on page 13).

DUMP_LOCKTABLE

DUMP_LOCKTABLE determines whether or not a lock has been released:

Syntax

```
SELECT DUMP_LOCKTABLE()
```

Notes

- Send the output to ***Technical Support*** (on page 13).

GET_PROJECTION_STATUS

GET_PROJECTION_STATUS returns information relevant to the status of a projection:

- the current K-Safety status of the database
- the number of nodes in the database
- whether or not the projection is segmented
- the number and names of buddy projections
- whether or not the projection is safe
- whether or not the projection is up-to-date

This function may be deprecated and replaced by a mechanism similar to the ***SQL Virtual Tables (Monitoring API)*** (page 191) in a future release.

Syntax

```
GET_PROJECTION_STATUS('projection');
```

Semantics

projection	is the name of the projection for which to display status

Notes

- You can use GET_PROJECTION_STATUS to monitor the progress of a projection data refresh (see ***ALTER PROJECTION*** (page 138)).

Examples

```
=> SELECT GET_PROJECTION_STATUS('t1_sp2');

                                    get_projection_status
-------------------------------------------------------------------------------------------------
 Current system K is 0.
# of Nodes: 1.
t1_sp2 [Segmented: No] [# of Buddies: 0] [No buddy projections] [Safe: Yes] [UptoDate: No]
```

GET_TABLE_PROJECTIONS

GET_TABLE_PROJECTIONS returns information relevant to the status of a table:

- the current K-Safety status of the database
- the number of nodes (sites) in the database
- the number of projections for which the specified table is the anchor table
- for each of those projections:
  - the projection's buddy projections
  - whether or not the projection is segmented
  - whether or not the projection is safe
  - whether or not the projection is up-to-date

Syntax

```
GET_TABLE_PROJECTIONS('table')
```

Semantics

table	is the name of the table for which to list projections

Notes

- You can use GET_TABLE_PROJECTIONS to monitor the progress of a projection data refresh (see ***ALTER PROJECTION*** (page 138)).

Examples

```
=> SELECT GET_TABLE_PROJECTIONS('t1');

                                    get_table_projections
-------------------------------------------------------------------------------------------------
 Current system K is 0.
# of Sites: 1.
Table t1 has 2 projections.

Projection Name: [Segmented] [# of Buddies] [Buddy Projections] [Safe] [UptoDate]
-------------------------------------------------------
t1_p1 [Segmented: No] [# of Buddies: 0] [No buddy projections] [Safe: Yes] [UptoDate: Yes]
t1_sp1 [Segmented: No] [# of Buddies: 0] [No buddy projections] [Safe: Yes] [UptoDate: Yes]
```

INSTALL_LICENSE

SELECT INSTALL_LICENSE(<FILENAME>) installs the license key in the global catalog.

description

Syntax

```
INSTALL_LICENSE( 'filename' )
```

Semantics

filename	specifies the absolute pathname of a valid license file.

Notes

- See Managing Your License Key in the Database Administrator's Guide for more information about license keys.

Examples

```
SELECT INSTALL_LICENSE('/tmp/vlicense.key');
```

MARK_DESIGN_KSAFE

Use MARK_DESIGN_KSAFE to **enable or disable automatic recovery** in case of a failure. Before enabling automatic recovery, MARK_DESIGN_KSAFE **queries the catalog** to determine whether or not a cluster's physical schema design meets the following requirements:

- Dimension tables are replicated on all nodes.
- Fact table superprojections are segmented with each segment on a different node.
- Each fact table projection has at least one "buddy" projection.
- Each segment of each fact table projection exists on two nodes.

  Two projections are considered to be buddies if they contain the same columns and have the same segmentation. They can have different sort orders.

MARK_DESIGN_KSAFE does not change the physical schema in any way.

Syntax

```
SELECT MARK_DESIGN_KSAFE(k)
```

Semantics

k	1 enables automatic recovery if the schema design meets requirements 0 disables automatic recovery

If you specify a *k* value of one (1), Vertica returns one of the following messages:

```
Marked design 1-safe
```

or

```
The schema does not meet requirements for K=1.
Fact table projection projection-name
has insufficient "buddy" projections..
```

The database's internal automatic recovery state persists across database restarts but is not checked at startup time.

To illustrate, consider the following sequence of events:

1. Your database has automatic recovery enabled.
2. You drop a table (and corresponding projections) in a running database.
3. You forget to MARK_DESIGN_KSAFE(0).
4. You shut down the database.
5. You start the database with automatic recovery enabled.
6. Recovery fails because the physical schema design no longer meets requirements.

Notes

- In a newly-created database, automatic recovery is disabled by default.
- If a database has had automatic recovery enabled, you must temporarily disable automatic recovery in order to create a new table.
- For information about monitoring K-Safety, see ***VT_SYSTEM*** (page 198) in the ***SQL Virtual Tables (Monitoring API)*** (page 191).

Examples

```
> SELECT MARK_DESIGN_KSAFE(1);
  mark_design_ksafe
----------------------
 Marked design 1-safe
(1 row)
```

If the physical schema design is not K-safe, messages indicate which projections do not have a buddy:

```
> SELECT MARK_DESIGN_KSAFE(1);
The given K value is not correct; the schema is 0-safe
Projection pp1 has 0 buddies, which is smaller that the given K of 1
Projection pp2 has 0 buddies, which is smaller that the given K of 1
        ⋮
(1 row)
```

SET_ATM_MODE

SET_ATM_MODE controls the behavior of the ATM while loading data.

Syntax

```
SELECT SET_ATM_MODE('mode')
```

Semantics

mode	is one of the following: NORMAL - Use this mode when using DML Commands (or ***COPY*** (page 143)) to load less than one percent (1%) of the fact table data daily. This is the default. BULKLOAD - Use this mode when using ***COPY ... DIRECT*** (page 143) to bulk load large volumes of data into the ROS.

Notes

- SET_ATM_MODE returns to the default NORMAL mode when the database starts up.
- Set the ATM mode to BULKLOAD within ten minutes of database startup so that it can take effect before a mergeout begins. A mergeout can take a long time to finish.

Examples

To set the automatic tuple mover to bulk load mode:

```
SELECT SET_ATM_MODE('BULKLOAD');
```

To return the automatic tuple mover to normal mode:

```
SELECT SET_ATM_MODE('NORMAL');
```

START_REFRESH

START_REFRESH transfers data to projections that are not able to participate in query execution due to missing or out-of-date data.

Syntax

```
START_REFRESH()
```

Notes

- All nodes must be up in order to start a refresh.
- A refresh can start when:
  - a new projection is created for tables that already have data
  - recovery has completed
  - the ***MARK_DESIGN_KSAFE*** (page 133) function is called

- If a refresh is already running the function has no effect.
- Shutting down the database ends the refresh.
- To monitor the refresh operation, examine the log files on each node.

SQL Commands

The Vertica documentation makes a distinction between SQL commands and statements. A command is an SQL keyword that specifies the imperative verb in a statement. A statement is a complete SQL command that can be executed. For example, SELECT is a command while SELECT C1 FROM T1 is a statement (with or without a semicolon).

ALTER PROJECTION

This command manually forces a mergeout or moveout operation.

See Understanding the Automatic Tuple Mover in the Database Administrator's Guide (Advanced) for an explanation of how to use this command.

Syntax

```
ALTER PROJECTION projection
  { MOVEOUT | MERGEOUT [ FROM start_epoch TO end_epoch ] }
```

Semantics

projection	specifies the name of a projection.
MOVEOUT	manually forces a moveout operation
MERGEOUT	combines two or more ROS containers into a single container.
start_epoch *end_epoch*	specifies the epochs to merge into a single ROS container. Epoch numbering begins at zero (0) at database creation time and is incremented each time an advance epoch occurs, either automatically or manually using ***SELECT ADVANCE_EPOCH*** (page 127). Epoch numbering does not restart and continues to advance throughout the life of the database. Epoch numbers, therefore, can become quite large.

Notes

- Use ALTER PROJECTION MOVEOUT immediately after using ***SELECT ADVANCE_EPOCH*** (page 127).
- This command may not be supported in future releases.

ALTER TABLE

Adds or drops a single column or a table constraint from the metadata of a table.

Syntax

```
ALTER TABLE table-name
  {
    ADD COLUMN column-definition (on page 153)
    | ADD table-constraint (on page 139)
    | DROP CONSTRAINT constraint-name
  }
```

Semantics

table-name	specifies the name of the table to be altered
ADD COLUMN *column-definition*	adds a new column to a table and to **all superprojections** of the table. A unique projection column name is generated in each superprojection. The column is populated according to the ***column-constraint*** (on page 154).
ADD *table-constraint*	adds a table constraint (see ***table-constraint*** (on page 139)) to a table that does not have any associated projections.
DROP CONSTRAINT *constraint-name*	drops the specified ***table-constraint*** (on page 139) from a table that does not have any associated projections.

Notes

- Adding a column to a table does not affect the K-Safety of the physical schema design.
- Cancelling a ALTER TABLE statement can cause unpredictable results. Vertica Systems, Inc. recommends that you allow the statement to finish, then use another ALTER TABLE statement to undo the changes.
- You cannot use ALTER TABLE ... ADD COLUMN on a temporary table.

Example

```
ALTER TABLE Product_Dimension
    ADD CONSTRAINT PK_Product_Dimension PRIMARY KEY (Product_Key);
```

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

table-constraint

Adds a join constraint or a constraint describing a functional dependency to the metadata of a table. See Adding Constraints in the Database Administrator's Guide.

Syntax

```
[ CONSTRAINT constraint_name ]
  { PRIMARY KEY ( column [ , ... ] )
    | FOREIGN KEY ( column [ , ... ] )
        REFERENCES table [ ( column [ , ... ] ) ]
    | CORRELATION ( column1 ) DETERMINES ( column2 )
  }
```

Semantics

CONSTRAINT *constraint-name*	optionally assigns a name to the constraint. Vertica recommends that you name all constraints.
PRIMARY KEY (*column* [, ...])	adds a referential integrity constraint defining one or more NOT NULL numeric columns as the primary key.
FOREIGN KEY (*column* [, ...])	adds a referential integrity constraint defining one or more NOT NULL numeric columns as a foreign key.
REFERENCES *table* [(*column* [, ...])	specifies the table to which the FOREIGN KEY constraint applies. If *column* is omitted, the default is the primary key of *table*.
CORRELATION	describes a functional dependency. Given a tuple and the set of values in *column1*, one can determine the corresponding value of *column2*.

Notes

- Use the ***ALTER TABLE*** (page 139) command to add a table constraint. The CREATE TABLE statement does not allow table constraints.
- You must define primary key and foreign key constraints in all tables that participate in joins. See Adding Join Constraints.

Examples

```
CORRELATION (Product_Description) DETERMINES (Category_Description)
```

The Retail Sales Example Database described in the Quick Start contains a table Product_Dimension in which products have descriptions and categories. For example, the description "Seafood Product 1" exists only in the "Seafood" category. You can define several similar correlations between columns in the Product Dimension table.

ALTER USER

ALTER USER changes a database user account

Syntax

```
ALTER USER name [ WITH [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password' ]
```

Semantics

name	specifies the name of the user to alter; names that contain special characters must be double-quoted.
ENCRYPTED	is the default.
password	is the password to assign to the user.

Notes

- User names are not case-sensitive.
- ALTER USER ... PASSWORD cannot be used in a database that was created without a superuser password.

  ```
  ERROR: authorization method for database name is TRUST; no password
     set
  ```

COMMIT

The COMMIT command ends the current transaction and makes all changes that occurred during the transaction permanent and visible to other users.

Syntax

```
COMMIT [ WORK | TRANSACTION ]
```

Semantics

`WORK` `TRANSACTION`	have no effect; they are optional keywords for readability.

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

COPY

The COPY command is designed for bulk loading data from a file on a cluster host into a Vertica database. It reads data from a delimited text file and inserts tuples either into the WOS (memory) or directly into the ROS (disk).

See ***LCOPY*** (page 170) to load from a data file on a client system.

Syntax

```
COPY table [ column [ ,...] ]
  FROM { 'file' | STDIN }
  [ WITH ]
  [ DELIMITER [ AS ] 'char' ]
  [ NULL [ AS ] 'string' ]
  [ RECORD TERMINATOR 'string' ]
  [ EXCEPTIONS 'pathname' ]
  [ REJECTED DATA 'pathname' ]
  [ ABORT ON ERROR ]
  [ DIRECT ]
```

Semantics

table	specifies the name of a schema table (not a projection). Vertica loads the data into all projections that include columns from the schema table. It does not delete or overwrite any existing data.
column	restricts the load to one or more specific columns in the table (all columns are loade by default). Table columns that are not in the column list are given their default values. If no default value is defined for a column, COPY inserts NULL. The data file must contain the same number of columns as the COPY command's column list. For example, in a table T1 with nine columns (C1 through C9), COPY T1 (C1, C6, C9) would load the three columns of data in each record to columns C1, C6, and C9 respectively.
FROM '*file*'	specifies the absolute pathname of the text file containing the data. The file must be accessible to the host on which the COPY statement runs. (You can use variables t construct the pathname as described in Using Load Scripts.)
STDIN	reads from the standard input instead of a file.
WITH AS	are for readability and have no effect.
DELIMITER '*char*'	specifies the single-character column delimiter in the text file. For example, comma is the delimiter commonly used in textual (CSV) data files. In data files, the number of delimited column values is significant; rows can begin and/or end with a delimiter or a column value. The default delimiter is the tab character. The example database data files use a different delimiter: vertical bar (|). Use the backslash character (\) to specify special (non-printing, control) characters

<table>
<tr><td></td><td>as the delimiter. For example:
'\t' = tab character (the default)
'\\' = backslash character (not a good choice)
If the delimiter character appears in string data values, you can use the backslash character to indicate that it is a literal (see Loading Character Data).</td></tr>
<tr><td>NULL 'string'</td><td>specifies the multi-character string that represents a null value such as 'NULL'. The null string is case-insensitive and must be the only value between the delimiters. Fo example, if the null string is NULL and the delimiter is the vertical bar (|):
|nuLL| indicates a null value
| nuLL | does not indicate a null value
The default null string is \N and \n (backslash uppercase en or backslash lowercase en). The example database data files use the default null string.
When you use the COPY command in a script, you must use a double-backslash in a null string that includes a backslash.
For example, the scripts used to load the example databases contain:
COPY ... NULL '\\n' ...
The example scripts specify the null string to demonstrate this requirement, in spite of the fact that it is the default null string.
The null string that you specify for the Database Designer does not require a double-backslash.</td></tr>
<tr><td>RECORD TERMINATOR 'string'</td><td>specifies the literal character string that indicates the end of a data file record. You can include non-printing characters and backslash characters in the string accordin to the following convention:
<table><tr><th>Sequence</th><th>Description</th><th>Abbreviation</th><th>ASCII Decimal</th></tr><tr><td>\0</td><td>Null character</td><td>NUL</td><td>0</td></tr><tr><td>\a</td><td>Bell</td><td>BEL</td><td>7</td></tr><tr><td>\b</td><td>Backspace</td><td>BS</td><td>8</td></tr><tr><td>\t</td><td>Horizontal Tab</td><td>HT</td><td>9</td></tr><tr><td>\n</td><td>Linefeed</td><td>LF</td><td>10</td></tr><tr><td>\v</td><td>Vertical Tab</td><td>VT</td><td>11</td></tr><tr><td>\f</td><td>Formfeed</td><td>FF</td><td>12</td></tr><tr><td>\r</td><td>Carriage Return</td><td>CR</td><td>13</td></tr><tr><td>\\</td><td>Backslash</td><td></td><td>92</td></tr></table></td></tr>
<tr><td>EXCEPTIONS 'pathname'</td><td>specifies the filename or absolute pathname in which to write messages indicating the input line number and the reason for each rejected data record. The default pathname is:
catalog-dir/CopyErrorLog/input-filename-copy-from-exceptions
where catalog-dir represents the directory in which the database catalog files are</td></tr>
</table>

	stored, and *input-filename* is the name of the data file. If copying from STDIN, the *input-filename* is STDIN.
REJECTED DATA '*pathname*'	specifies the filename or absolute pathname in which to write rejected rows. This fil can then be edited to resolve problems and reloaded. The default pathname is: *catalog-dir*/CopyErrorLog/*input-filename*-copy-from-rejected-data where *catalog-dir* represents the directory in which the database catalog files are stored, and *input-filename* is the name of the data file. If copying from STDIN, the *input-filename* is STDIN.
ABORT ON ERROR	stops the COPY command if a row is rejected and rolls back the command. No data is loaded.
DIRECT	specifies that the data should go directly to the ROS (Read Optimized Store. By default, data goes to the WOS (Write Optimized Store).

Although they both specify the same things, the syntax of the COPY command is different from the Database Designer input parameter syntax.

Notes

- The COPY command automatically commits itself and any current transaction. Vertica recommends that you ***COMMIT*** (page 142) or ***ROLLBACK*** (page 173) the current transaction before using COPY.
- You cannot use the same character in both the DELIMITER and NULL strings.
- NULL values are not allowed for columns with primary key or foreign key referential integrity constraints.
- String data in load files is considered to be all characters between the specified delimiters. Do not enclose character strings in quotes. In other words, quote characters are treated as ordinary data.
- Invalid input is defined as:
  - Missing columns (too few columns in an input line).
  - Extra columns (too many columns in an input line).
  - Empty columns for INTEGER or date/time data types. COPY does not use the default data values defined by the CREATE TABLE command.
  - Incorrect representation of data type. For example, non-numeric data in an integer column is invalid.
- Empty values (two consecutive delimiters) are accepted as valid input data for CHAR and VARCHAR data types. Empty columns are stored as an empty string (''), which is not equivalent to a null string.
- Cancelling a COPY statement rolls back all rows loaded by that statement.

Examples

```
COPY Store_Dimension
```

```
FROM :      input_file
      DELIMITER '|'
      NULL '\\n'
      RECORD TERMINATOR '\f'
      DIRECT;
```

CREATE PROJECTION

The CREATE PROJECTION command creates metadata for a projection in the Vertica catalog. It does not load data into physical storage unless the tables(s) over which the projection is defined already contain data. In that case, the new projection becomes available as soon as the database has automatically refreshed it. This process may take a long time, depending on how much data is in the table(s).

Use CREATE PROJECTION only when advised to do so by ***Technical Support*** (on page 13). Improper use can corrupt data and/or damage a database.

Syntax

```
CREATE PROJECTION projection-name
  ( projection-column [ ENCODING encoding-type (on page 148) ] [ , ... ] )
 AS SELECT table-column [ , ... ]
    FROM-clause (see "FROM Clause" on page 176)
    [ WHERE join-predicate (on page 54) [ AND join-predicate (on page 54) ]
...
    [ ORDER BY table-column [ , ... ] ]
    [ hash-segmentation-clause (on page 149) | range-segmentation-clause (on
page 150)
      | UNSEGMENTED { NODE node | ALL NODES } ]
```

Semantics

projection-name	specifies the name of the projection (see note below).
projection-column	specifies the name of a column in the projection. The data type is inferred from the corresponding column in the schema table (based on ordinal position). Different *projection-column* names can be used to distinguish multiple columns of the same name from different tables so that no aliases are needed.
encoding-type	specifies the ***type of encoding*** (see "encoding-type" on page 148) to use on the column. The Database Designer automatically chooses an appropriate encoding for each projection column.
SELECT *table-column*	specifies a list of schema table columns corresponding (in ordinal position) to the projection columns.
FROM-clause	specifies a list of schema tables containing the schema columns. (See ***FROM Clause*** (on page 176).) The first table in the list is the anchor table. The anchor table of a pre-join projection can be: • the fact table in a star schema • the central fact table in a snowflake schema • a dimension table in a snowflake schema that functions as a fact table (with a restriction)

WHERE *join-predicate*	specifies foreign-key = primary-key equijoins between the fact table and dimensio tables. Foreign key columns must be NOT NULL. No other predicates are allowed
ORDER BY *table-column*	specifies which columns to sort. Because all projection columns are sorted in ascending order in physical storage, CREATE PROJECTION does not allow you to specify ascending or descending.
hash-segmentation-clause	allows you to segment a projection based on a built-in hash function that provides even distribution of data across nodes, resulting in optimal query execution. See ***hash-segmentation-clause*** (on page 149).
range-segmentation-clause	allows you to segment a projection based on a known range of values stored in a specific column chosen to provide even distribution of data across a set of nodes, resulting in optimal query execution. See ***range-segmentation-clause*** (on page 150).
NODE *node*	creates an unsegmented projection on the specified node only. Dimension table projections must be UNSEGMENTED.
ALL NODES	creates a separate unsegmented projection on each node at the time the CREATE PROJECTION statement is executed (automatic replication). In order to do distributed quer execution, Vertica requires an exact, unsegmented copy of each dimension table superprojection on each node. See projection naming note below.

Unsegmented Projection Naming

CREATE PROJECTION ... UNSEGMENTED takes a snapshot of the nodes defined at execution time to generate a node list in a predictable order. Thus, replicated projections have the name:

projection-name_node-name

For example, if the nodes are named NODE01, NODE02, and NODE03 then:

```
CREATE PROJECTION ABC ... UNSEGMENTED ALL NODES
```

creates projections named ABC_NODE01, ABC_NODE02, and ABC_NODE03.

This naming convention may impact functions that provide information about projections, for example, GET_TABLE_PROJECTIONS or GET_PROJECTION_STATUS where you will need to provide the name ABC_NODE01 instead of just ABC. To view a list of the nodes in a database, use the View Database command in the Administration Tools.

Notes

- If no segmentation is specified, the default is UNSEGMENTED on the node where the CREATE PROJECTION was executed.

encoding-type

Type	Description	Use For
RLE	RLE (Run Length Encoding) replaces sequences of the same data values within a column by a	sorted columns in which the average number of repeated rows exceeds ten.

	single value and a count number.	
DELTAVAL	DELTAVAL (Delta Encoding) stores only the differences between sequential data values rather than the values themselves.	columns that store sorted integer or date data that has a narrow range of values.
NONE	automatic encoding (default)	all columns not suitable for RLE or DELTAVAL.

hash-segmentation-clause

Hash segmentation allows you to segment a projection based on a built-in hash function that provides even distribution of data across some or all of the nodes in a cluster, resulting in optimal query execution.

Hash segmentation is the preferred method of segmentation in Vertica 2.0 and later.

Syntax

```
SEGMENTED BY expression
  [ ALL NODES [ OFFSET offset ] | NODES node [ ,... ] ]
```

Semantics

SEGMENTED BY *expression*	can be a general SQL expression but there is no reason to use anything other than the built-in ***HASH*** (page 104) function with one or more NOT NULL columns as arguments. Choose columns that have a **large number of unique** data values and acceptable skew in the data distribution. Primary key columns that meet the criteria may be an excellent choice for hash segmentation.
ALL NODES	automatically distributes the data evenly across all nodes at the time the CREATE PROJECTION statement is executed. The ordering of the nodes is fixed.
OFFSET *offset*	is an integer that specifies the node within the ordered sequence on which to start the segmentation distribution, relative to 0. See example below.
NODES *node* [,...]	specifies a subset of the nodes in the cluster over which to distribute the data. You can use a specific node only once in any projection. For a list of the nodes in a database, use the View Database command in the Administration Tools.

Notes

- Hash segmentation is the preferred method of segmentation in Vertica 2.0 and later. The Database Designer uses hash segmentation by default.

- During INSERT or COPY to a segmented projection, if *expression* produces a value outside the expected range (a negative value for example), no error occurs, and the row is added to a segment of the projection.
- No OFFSET clause is equivalent to OFFSET 0.
- CREATE PROJECTION accepts the deprecated syntax `SITES` *node* for compatibility with previous releases.
- If you wish to use a different `SEGMENTED BY` expression, the following restrictions apply:
  - All leaf expressions must be either ***constants*** (on page 31) or ***column-references*** (see "Column References" on page 46) to a column in the SELECT list of the CREATE PROJECTION command
  - Aggregate functions are not allowed
  - The expression must return the same value over the life of the database.
  - The expression must return non-negative INTEGER values in the range 0 to 263.

Examples

```
CREATE PROJECTION ... SEGMENTED BY HASH(C1,C2,...) ALL NODES;
CREATE PROJECTION ... SEGMENTED BY HASH(C1,C2,...) ALL NODES OFFSET 1;
```

The example produces two hash-segmented buddy projections that form part of a K-Safe design. The projections can use different sort orders.

range-segmentation-clause

Range segmentation allows you to segment a projection based on a known range of values stored in a specific column chosen to provide even distribution of data across a set of nodes, resulting in optimal query execution.

Vertica Systems, Inc. recommends that you use Hash Segmentation instead of range segmentation.

Syntax

```
SEGMENTED BY expression
  NODE node VALUES LESS THAN value
    ⋮
  NODE node VALUES LESS THAN MAXVALUE
```

Semantics (Range Segmentation)

`SEGMENTED BY` *expression*	is a single ***column reference*** (see "Column References" on page 46) to a column in the SELECT list of the CREATE PROJECTION command. Choose a column that has: **INTEGER** or **FLOAT** data type a **known range** of data values an **even distribution** of data values

	a **large number of unique** data values Avoid columns that: are foreign **keys** are used in **query predicates** have a **date/time** data type have **correlations** with other columns due to functional dependencies. Segmenting on date/time data types is valid but guaranteed to produce temporal skew in the data distribution and is not recommended. If you choose this option, do not use TIME or TIMETZ because their range is only 24 hours.
`NODE` *node*	a symbolic name for a node. You can use a specific node only once in any projection. For a list of the nodes in a database, use the View Database command in the Administration Tools.
`VALUES LESS THAN` *value*	specifies that this segment can contain a range of data values *less than* the specified *value*, except that segments cannot overlap. In other words, the minimum value of the range is determined by the *value* of the previous segment (if any).
`MAXVALUE`	specifies a sub-range with no upper limit. In other words, it represents a value greater than the maximum value that can exist in the data. The maximum value depends on the data type of the segmentation column.

Notes

- The `SEGMENTED BY` *expression* syntax allows a general SQL expression but there is no reason to use anything other than a single ***column reference*** (see "Column References" on page 46) for range segmentation. If you wish to use a different expression, the following restrictions apply:
  - All leaf expressions must be either ***constants*** (on page 31) or ***column-references*** (see "Column References" on page 46) toa column in the SELECT list of the CREATE PROJECTION command
  - Aggregate functions are not allowed
  - The expression must return the same value over the life of the database.
- During INSERT or COPY to a segmented projection, if *expression* produces a value outside the expected range (a negative value for example), no error occurs, and the row is added to a segment of the projection.
- CREATE PROJECTION with range segmentation accepts the deprecated syntax `SITE` *node* for compatibility with previous releases.
- CREATE PROJECTION with range segmentation allows the `SEGMENTED BY` expression to be a single column-reference to a column in the *projection-column* list for compatibility with previous releases. This syntax is considered to be a deprecated feature and causes a warning message. See DEPRECATED syntax in the Troubleshooting Guide.

CREATE TABLE

Creates a table in the logical schema.

CREATE TABLE does not create a projection corresponding to the table. Every column in the table must exist in at least one projection before you can store data in the table.

Syntax

```
CREATE TABLE table_name ( column-definition (on page 153) [ , ... ] )
```

Semantics

table-name	specifies the name of the table to be created
column-definition	defines one or more columns. See ***column-definition*** (on page 153)

Notes

- CREATE TABLE does not allow table constraints, only column and correlation constraints.
- If a database has had automatic recovery enabled, you must temporarily disable automatic recovery in order to create a new table. In other words, you must:
  ```
  SELECT MARK_DESIGN_KSAFE(0)
  CREATE TABLE ...
  CREATE PROJECTION ...
  SELECT MARK_DESIGN_KSAFE(1)
  ```
- Cancelling a CREATE TABLE statement can cause unpredictable results. Vertica Systems, Inc. recommends that you allow the statement to finish, then use ***DROP TABLE*** (page 160).

Examples

```
CREATE TABLE Product_Dimension (
    Product_Key                    integer NOT NULL,
    Product_Description            varchar(128),
    SKU_Number                     char(32) NOT NULL,
    Category_Description           char(32),
    Department_Description         char(32) NOT NULL,
    Package_Type_Description       char(32),
    Package_Size                   char(32),
    Fat_Content                    integer,
    Diet_Type                      char(32),
    Weight                         integer,
    Weight_Units_of_Measure        char(32),
    Shelf_Width                    integer,
    Shelf_Height                   integer,
    Shelf_Depth                    integer
);
```

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

column-definition

A column definition specifies the name, data type, and constraints to be applied to a column.

Syntax

```
column-name data-type [ column-constraint (on page 154) [ ... ] ]
```

Semantics

column-name	specifies the name of a column to be created or added.
data-type	specifies one of the following data types: ***BOOLEAN*** (on page 59) ***CHARACTER (CHAR)*** (on page 61) ***CHARACTER VARYING (VARCHAR)*** (on page 61) ***Date/Time Types*** (see "Date/Time" on page 63) ***DOUBLE PRECISION (FLOAT)*** (on page 70) ***INTEGER (BIGINT)*** (on page 73)
column-constraint	specifies a ***column constraint*** (see "column-constraint" on page 154) to apply to the column.

column-constraint

Adds a referential integrity constraint to the metadata of a column. See Adding Join Constraints in the Database Administrator's Guide.

Syntax

```
[ CONSTRAINT constraint-name ]
  { [ NOT ] NULL
    | PRIMARY KEY
    | REFERENCES table-name [ ( column-name ) ]
    [ DEFAULT constant ]
  }
```

Semantics

CONSTRAINT *constraint-name*	optionally assigns a name to the constraint. Vertica recommends that you name all constraints.
NULL	(default) specifies that the column is allowed to contain null values.
NOT NULL	specifies that the column must receive a value during INSERT and UPDATE operations. If no DEFAULT value is specified and no value is provided, the INSERT or UPDATE statement returns an error because no default value exists.
PRIMARY KEY	adds a referential integrity constraint defining the column as the primary key.
REFERENCES	adds a referential integrity constraint defining the column as a foreign key. If column is omitted, the default is the primary key of table.
table-name	specifies the table to which the REFERENCES constraint applies.
column-name	specifies the column to which the REFERENCES constraint applies. If column is omitted, the default is the primary key of table-name.
DEFAULT *constant*	specifies a constant to be used by default in any INSERT operation that does not specify a value for the column. The data type of the constant must match the data type of the column. If no default value is specified, the default is null.

Notes

- You must define primary key and foreign key constraints in all tables that participate in joins. See Adding Constraints.
- You must specify NOT NULL constraints on columns that will be given PRIMARY and REFERENCES constraints.
- Vertica does not support expressions in the DEFAULT clause.

CREATE TEMPORARY TABLE

Creates a table and a corresponding superprojection consisting of persistent metadata and data that is materialized only until the current transaction ends. In other words, rows stored in a temporary table exist only in memory (WOS). They are never moved to disk (ROS).

The purpose of a temporary table is to perform complex queries in multiple steps: first get a result set, then query the result set, and so forth.

Syntax

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ]
TABLE table-name (
  { column-name data-type
    [ DEFAULT default ]
    [ NULL | NOT NULL ]
  } ...
)
[ NO PROJECTION ]
[ ON COMMIT { DELETE ROWS | PRESERVE ROWS } ]
```

Semantics

GLOBAL LOCAL	are ignored; temporary tables are always global (visible to all database users).
table-name	specifies the name of the temporary table to be created.
column-name	specifies the name of a column to be created in the new temporary table.
data-type	specifies one of the supported data types: ***BOOLEAN*** (on page 59) ***CHARACTER*** (see "CHARACTER (CHAR)" on page 61) ***CHARACTER VARYING*** (see "CHARACTER VARYING (VARCHAR)" on page 61) ***Date/Time Types*** (see "Date/Time" on page 63) ***DOUBLE PRECISION (FLOAT8)*** (see "DOUBLE PRECISION (FLOAT)" on page 70) ***INTEGER AND BIGINT*** (see "INTEGER (BIGINT)" on page 73)
DEFAULT *default*	assigns a default data value to be used in any INSERT operation that does not specify a value for the column. The data type of the default expression must match the data type of the column. If no default value is specified, the default

	is null.
NULL	(default) specifies that the column is allowed to contain null values.
NOT NULL	specifies that the column must receive a value during INSERT and UPDATE operations. If no DEFAULT value is specified and no value is provided, the INSERT or UPDATE statement returns an error because no default value exists.
NO PROJECTION	prevents the automatic creation of a superprojection for the temporary table using a default algorithm to choose sort order and compression.
DELETE ROWS PRESERVE ROWS	is ignored; rows are always deleted on commit.

Notes

- Unsegmented projections of temporary tables reside on the node that initiated the transaction.
- SELECT queries do not lock temporary tables.
- The COPY and INSERT SELECT commands do not auto-commit when applied to temporary tables.
- The scope of a temporary table in Vertica is the current transaction. This is not the same as the temporary table scope in PostgreSQL, which is the current session.
- You cannot query temporary tables using Snapshot Isolation. The isolation level (specified using ***SET SESSION CHARACTERISTICS*** (page 185)) must be READ COMMITED or READ UNCOMMITTED.
- You cannot query temporary tables while the database is being loaded.

CREATE USER

CREATE USER adds a name to the list of authorized database users.

Syntax

```
CREATE USER name [ WITH [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password' ]
```

Semantics

name	specifies the name of the user to create; names that contain special characters must be double-quoted.
ENCRYPTED	is the default.
password	is the password to assign to the user.

Notes

- User names are not case-sensitive.
- The following options are allowed but ignored:
  - SYSID uid
  - CREATEDB
  - NOCREATEDB
  - CREATEUSER
  - NOCREATEUSER
  - VALID UNTIL
- Other options, including IN GROUP, are not allowed.
- Newly-created users do not have access to schema PUBLIC by default. Make sure to GRANT USAGE ON SCHEMA PUBLIC to all users you create.

Examples

```
CREATE USER Fred;
GRANT USAGE ON SCHEMA PUBLIC to Fred;
```

DELETE

DELETE marks tuples as no longer valid in the current epoch. It does not delete data from disk storage.

Syntax

`DELETE FROM` *`table`* ***`WHERE clause`*** `(on page 177)`

Semantics

`table`	specifies the name of a table in the schema. You cannot delete tuples from a projection.

Notes

- DELETE marks tuples for deletion in the WOS. Thus, be aware of WOS Overload.
- In order to use the ***DELETE*** (page 158) or ***UPDATE*** (page 190) commands with a ***WHERE clause*** (page 177), a user must have both SELECT and DELETE privileges on the table.

Examples

```
DELETE FROM T
WHERE  C1 = C2 - C1;
DELETE FROM CUSTOMER
WHERE  STATE IN ('MA', 'NH');
```

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

DROP PROJECTION

Marks a projection to be dropped from the catalog and makes it unavailable to user queries. The next mergeout operation drops the associated data.

Syntax

```
DROP PROJECTION projection
```

Semantics

projection	specifies the name of the projection to drop.

Notes

- DROP PROJECTION fails if the projection is the only superprojection for the table and contains data. In that case, you must use DROP TABLE ... CASCADE to drop the projection.

DROP TABLE

Drops a table and optionally its associated projections.

Syntax

```
DROP TABLE table [ CASCADE ]
```

Semantics

table	specifies the name of a schema table
CASCADE	drops all projections that include the table. Warning: dropping a table and its associated projections is very likely to destroy the K-Safety of your physical schema design. Use this command only when absolutely necessary.

if you try to drop an table that has associated projections, you will get a message listing the projections. For example:

```
=> DROP TABLE d1;
NOTICE: Constraint - depends on Table d1
NOTICE: Projection d1p1 depends on Table d1
NOTICE: Projection d1p2 depends on Table d1
NOTICE: Projection d1p3 depends on Table d1
NOTICE: Projection f1d1p1 depends on Table d1
NOTICE: Projection f1d1p2 depends on Table d1
NOTICE: Projection f1d1p3 depends on Table d1
ERROR: DROP failed due to dependencies: Cannot drop Table d1 because other objects depend on it
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

Notes

- Cancelling a DROP TABLE statement can cause unpredictable results.
- Vertica Systems, Inc. recommends that you gain exclusive access to the database before using DROP TABLE. In other words, make sure that all other users have disconnected.

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

DROP USER

DROP USER removes a name from the list of authorized database users.

Syntax

```
DROP USER name [, ...]
```

Semantics

name	specifies the name of the user to drop.

Examples

```
DROP USER Fred
```

EXPLAIN

The EXPLAIN command outputs the query plan.

Syntax

```
EXPLAIN { SELECT... | INSERT... | UPDATE... }
```

Output

The EXPLAIN command is provided as a support feature and is not fully described here. For information on how to interpret the output, contact ***Technical Support*** (page 13).

- A compact human-readable representation of the query plan, laid out hierarchically. For example:

```
Vertica QUERY PLAN DESCRIPTION:
------------------------------
  ID:1 Cost:2.7 Card:-1
    Projection: P0
       ID:2 Cost:0.1 Card:-1
         DS: Value Idx
         ProjCol:c_state, Table Oid.Attr#:25424.4
         Pred: Y         Out: P
       ID:3 Cost:0.3 Card:-1
         DS: Position Filtered by ID:2
         ProjCol:c_gender, Table Oid.Attr#:25424.2
         Pred: Y         Out: P
       ID:4 Cost:0.3 Card:-1
         DS: Position Filtered by ID:3
         ProjCol:c_name, Table Oid.Attr#:25424.3
         Pred: Y         Out: P
       ID:5 Cost:1 Card:-1
         DS: Position Filtered by ID:4
         ProjCol:c_cid, Table Oid.Attr#:25424.1
         Pred: N         Out: V
       ID:6 Cost:1 Card:-1
         DS: Position Filtered by ID:4
         ProjCol:c_state, Table Oid.Attr#:25424.4
         Pred: N         Out: V
```

- A GraphViz format of the graph for display in a graphical format. Graphviz is a graph plotting utility with layout algorithms, etc. You can obtain a Fedora Core 4 RPM for GraphViz from:

  ```
  yum -y install graphviz
  ```

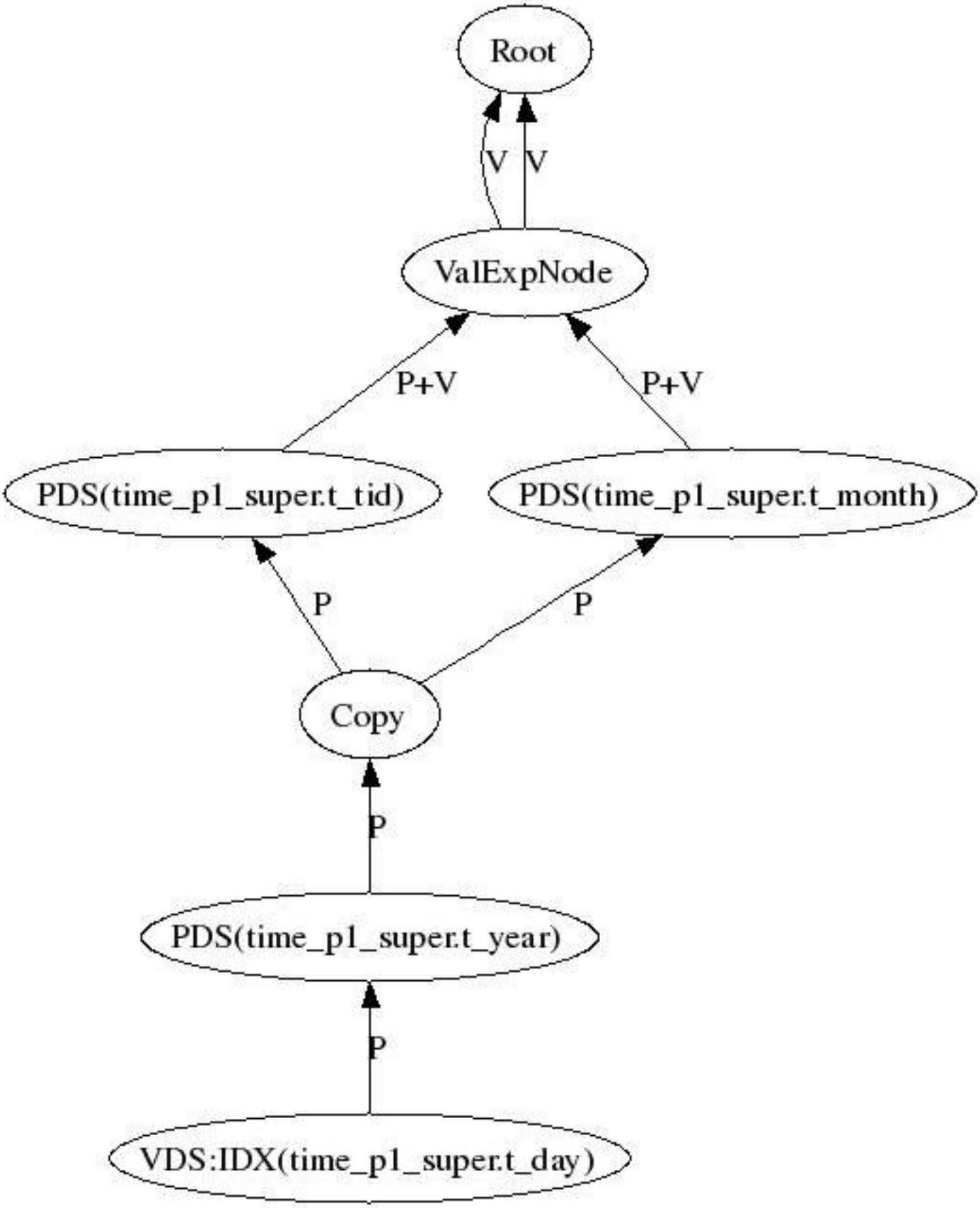
  A example of a GraphViz graph for a Vertica plan:

  ```
  digraph G {
  graph [rankdir=BT]
  0[label="Root"];
  1[label="ValExpNode"];
  ```

```
2[label="VDS:DVIDX(P0.c_state)"];
3[label="PDS(P0.c_gender)"];
4[label="PDS(P0.c_name)"];
5[label="Copy"];
6[label="PDS(P0.c_cid)"];
7[label="PDS(P0.c_state)"];
1->0 [label="V"];
1->0 [label="V"];
2->3 [label="P"];
3->4 [label="P"];
4->5 [label="P"];
5->6 [label="P"];
5->7 [label="P"];
6->1 [label="P+V"];
7->1 [label="P+V"]; }
```

- To create a picture of the plan, copy the output above to a file, in this example /tmp/x.txt:
  1. dot -Tps /tmp/x.txt > /tmp/x.ps
  2. ggv x.ps [evince x.ps works if you don't have ggv]
  3. Alternative: dot -Tps | ghostview - and paste in the digraph.
  4. Alternative: generate jpg using -Tjpg.
  5. To scale an image for printing (8.5"x11" in this example):
  6. Portrait: dot -Tps -Gsize="7.5,10" -Gmargin="0.5" ...
  7. Landscape: dot -Tps -Gsize="10,7.5" -Gmargin="0.5" -Grotate="90" ...

Example:


Root
V
V
ValExpNode
P+V
P+V
PDS(time_p1_super.t_tid)
PDS(time_p1_super.t_month)
P
P
Copy
P
PDS(time_p1_super.t_year)
P
VDS:IDX(time_p1_super.t_day)

GraphViz Information

http://www.graphviz.org/Documentation.php (http://www.graphviz.org/Documentation.php)

GRANT (Schema)

Grants privileges on a schema to a database user.

In the database with trust authentication, the GRANT and REVOKE statements work as expected. You cannot, however, depend on them for security because anyone can connect as the Database Superuser without supplying a password.

Syntax

```
GRANT { { CREATE | USAGE } [, ...]
      | ALL [ PRIVILEGES ]
      }
    ON SCHEMA schema-name [, ...]
    TO { username | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

Semantics

CREATE	allows username to create new tables.
USAGE	allows username to access all existing tables.
ALL	is synonymous with CREATE.
PRIVILEGES	is for SQL standard compatibility and is ignored.
schema-name	there is only one schema in 2.0.5-0 which is named PUBLIC.
username	grants the privilege to a specific user.
PUBLIC	grants the privilege to all users.
WITH GRANT OPTION	allows the recipient of the privilege to grant it to other users.

Notes

- Newly-created users do not have access to schema PUBLIC by default. Make sure to **GRANT USAGE ON SCHEMA PUBLIC** to all users you create.

GRANT (Table)

Grants privileges on a table to a user.

In the database with trust authentication, the GRANT and REVOKE statements work as expected. You cannot, however, depend on them for security because anyone can connect as the Database Superuser without supplying a password.

Syntax

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES } [,...]
        | ALL [ PRIVILEGES ]
      }
    ON [ TABLE ] tablename [, ...]
    TO { username | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

Semantics

SELECT	Allows the user to SELECT from any column of the specified table.
INSERT	Allows the user to INSERT tuples into the specified table and to use the ***COPY*** (page 143) command to load the table.
UPDATE	Allows the user to UPDATE tuples in the specified table.
DELETE	Allows DELETE of a row from the specified table.
REFERENCES	To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.
ALL	is synonomous with SELECT, INSERT, UPDATE, DELETE, REFERENCES.
PRIVILEGES	is for SQL standard compatibility and is ignored.
tablename	specifies the table on which to grant the privileges.
username	specifies the user to whom to grant the privileges.
PUBLIC	grants the privilege to all users.
WITH GRANT OPTION	allows the user to grant the same privileges to other users.

Notes

- In order to use the ***DELETE*** (page 158) or ***UPDATE*** (page 190) commands with a ***WHERE clause*** (page 177), a user must have both SELECT and UPDATE and DELETE privileges on the table.

INSERT

The INSERT command inserts values into the Write Optimized Store (WOS) for all projections of a table.

Syntax

```
INSERT [ /*+ direct */ ] INTO table [ ( column [, ...] ) ]
{ DEFAULT VALUES |
  VALUES ( { expression | DEFAULT } [, ...] ) | SELECT... (page 174)
}
```

Semantics

`/*+ direct */`	writes the data directly to disk (ROS) instead of memory (WOS). This syntax is only valid when used with INSERT...SELECT.
table	specifies the name of a table in the schema. You cannot INSERT tuples into a projection.
column	specifies a column of the table.
DEFAULT VALUES	fills all columns with their default values as specified in ***CREATE TABLE*** (page 152).
VALUES	specifies a list of values to store in the correspond columns. If no value is supplied for a column, Vertica implicitly adds a DEFAULT value, if present. Otherwise Vertica inserts a NULL value or, if the column is defined as NOT NULL, returns an error.
expression	specifies a value to store in the corresponding column.
DEFAULT	stores the default value in the corresponding column.
SELECT...	specifies a query (***SELECT*** (page 174) statement) that supplies the rows to be inserted.

Notes

- An INSERT ... SELECT ... statement refers to tables in both its INSERT and SELECT clauses. Isolation level applies only to the SELECT clauses and work just like an normal query except that you cannot use AT EPOCH LATEST or AT TIME in an INSERT ... SELECT statement. Instead, use the ***SET TRANSACTION CHARACTERISTICS*** (page 185) statement to set the isolation level to **READ COMMITTED**. This is necessary in order to use INSERT ... SELECT while the database is being loaded.
- You can list the target columns in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order; or the first N column names, if there are only N columns supplied by the VALUES clause or query. The values supplied by the VALUES clause or query are associated with the explicit or implicit column list left-to-right.
- You must insert one complete tuple at a time.
- Be aware of WOS Overload.

Examples

```
INSERT INTO FACT VALUES (101, 102, 103, 104);
INSERT INTO CUSTOMER VALUES (10, 'male', 'DPR', 'MA', 35);
INSERT INTO T1 (C0, C1) VALUES (1, 1001);
INSERT INTO films
  SELECT * FROM tmp_films
  WHERE date_prod < '2004-05-07';
```

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

LCOPY

The LCOPY command is identical to the ***COPY*** (page 143) command except that it loads data from a client system, rather than a cluster host.

Example

The following code loads the table TEST from the file `C:\load.dat` located on a system where the code is executed.

```
ODBCConnection<ODBCDriverConnect> test("VerticaSQL");
test.connect();
char *sql = "LCOPY test FROM 'C:\load.dat' DELIMITER '|';";
ODBCStatement stm(test.conn);
stm.execute(sql);
```

REVOKE (Schema)

Revokes privileges on a schema from a user.

In the database with trust authentication, the GRANT and REVOKE statements work as expected. You cannot, however, depend on them for security because anyone can connect as the Database Superuser without supplying a password.

Syntax

```
REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [,...]
        | ALL [ PRIVILEGES ]
    }
    ON SCHEMA schema-name [, ...]
    FROM { username | PUBLIC } [, ...]
```

Semantics

GRANT OPTION FOR	revokes the grant option for the privilege, not the privilege itself. If omitted, revokes both the privilege and the grant option.
CREATE	See ***GRANT (Schema)*** (page 167).
USAGE	
ALL	
PRIVILEGES	
schema-name	
username	
PUBLIC	

REVOKE (Table)

Revokes privileges on a table from a user.

In the database with trust authentication, the GRANT and REVOKE statements work as expected. You cannot, however, depend on them for security because anyone can connect as the Database Superuser without supplying a password.

Syntax

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | REFERENCES } [,...]
      | ALL [ PRIVILEGES ]
    }
    ON [ TABLE ] tablename [, ...]
    FROM { username | PUBLIC } [, ...]
```

Semantics

GRANT OPTION FOR	revokes the grant option for the privilege, not the privilege itself. If omitted, revokes both the privilege and the grant option.
SELECT	See ***GRANT (Table)*** (page 168).
INSERT	
UPDATE	
DELETE	
REFERENCES	
ALL	
PRIVILEGES	
tablename	
username	
PUBLIC	

ROLLBACK

The ROLLBACK command ends the current transaction and discards all changes that occurred during the transaction.

Syntax

```
ROLLBACK [ WORK | TRANSACTION ]
```

Semantics

`WORK` `TRANSACTION`	have no effect; they are optional keywords for readability.

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

SELECT

Retrieves a result set from one or more tables.

```
[ AT EPOCH LATEST ] | [ AT TIME 'timestamp' ]
SELECT * | [ ALL | DISTINCT ] expression [ AS output_name ] [, ...]
    [ FROM clause (on page 176) ]
    [ WHERE clause (on page 177) ]
    [ GROUP BY clause (on page 178) ]
    [ HAVING clause (on page 179) ]
    [ ORDER BY clause (on page 180) ]
    [ LIMIT clause (on page 181) ]
    [ OFFSET clause (on page 182) ]
```

Semantics

`AT EPOCH LATEST`	queries all data in the database up to but not including the current epoch without holding a lock or blocking write operations. See Snapshot Isolation for more information. AT EPOCH LATEST is ignored when applied to temporary tables (all rows are returned). By default, queries execute under the SERIALIZABLE isolation level, which holds locks and blocks write operations. For optimal query performance, use AT EPOCH LATEST.
`AT TIME 'timestamp'`	queries all data in the database up to and including the epoch representing the specified date and time without holding a lock or blocking write operations. This is called an Historical Query. AT EPOCH LATEST is ignored when applied to temporary tables (all rows are returned).
*	is equivalent to listing all columns of the tables in the ***FROM Clause*** (page 176). Vertica recommends that you *avoid* using SELECT * for performance reasons. An extremely large and wide result set can cause swapping.
`DISTINCT`	removes duplicate rows from the result set (or group).The DISTINCT set quantifier must immediately follow the SELECT keyword. Only one DISTINCT keyword can appear in the select list.
expression	forms the output rows of the SELECT statement. The expression can contain: ***Column References*** (on page 46) to columns computed in the ***FROM clause*** (page 176) ***Constants*** (on page 31) ***Mathematical Operators*** (on page 41) ***String Concatenation Operators*** (on page 42) ***Aggregate Expressions*** (page 44) ***CASE Expressions*** (page 45) ***SQL Functions*** (page 75)
output_name	specifies a different name for an output column. This name is primarily used to label

	the column for display. It can also be used to refer to the column's value in ***ORDER BY*** (page 180) and ***GROUP BY*** (page 178) clauses, but not in the ***WHERE*** (page 177) or ***HAVING*** (page 179) clauses.

Notes

- The SELECT list (between the key words SELECT and FROM) specifies expressions that form the output rows of the SELECT command.
- Subqueries are not supported.

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

FROM Clause

Specifies one or more source tables.

Syntax

```
FROM table-name [ [ AS ] alias [ ( column-alias [, ...] ) ] ] [, ...]
```

Semantics

table-name	specifies a table in the logical schema. Vertica selects a suitable projection to use. Each table can appear only *once* in the FROM clause.
alias	specifies a temporary name to be used for references to the table.
column-alias	specifies a temporary name to be used for references to the column.

Notes

- Joined table syntax (example: T1 JOIN T2 ON *boolean_expression*) is not allowed. Joins must be specified in the ***WHERE clause*** (page 177).
- Full Cartesian products (joins with no ***WHERE clause*** (page 177)) are not allowed.

WHERE Clause

Eliminates rows from the result table that do not satisfy one or more predicates.

Syntax

```
WHERE boolean-expression
```

Semantics

boolean-expression	is an expression that returns **true** or **false**. Only rows for which the expression is **true** become part of the result set.

The *boolean-expression* can include ***Boolean operators*** (on page 38) and the following elements:

BETWEEN-predicate (on page 50)

Boolean-predicate (on page 51)

column-value-predicate (on page 52)

IN-predicate (on page 53)

join-predicate (on page 54)

LIKE-predicate (on page 55)

NULL-predicate (on page 57)

Notes

- You can use parentheses to group expressions, predicates, and boolean operators. For example:
  ```
  WHERE NOT (A=1 AND B=2) OR C=3;
  ```

GROUP BY Clause

GROUP BY divides a query result set into groups of rows that match an expression.

Syntax

```
GROUP BY expression [ ,... ]
```

Semantics

expression	is any expression including constants and ***references to columns*** (see "Column References" on page 46) in the tables specified in the FROM clause.

Notes

- The *expression* cannot include ***aggregate functions*** (page 76).
- All non-aggregated columns in the SELECT list must be included in the GROUP BY clause.

Examples

```
SELECT C1, C2 FROM T1 GROUP BY C1, C2;
SELECT C1, AVG(C2)
FROM T1
GROUP BY C1;
SELECT x+y, SUM(z)
FROM foo
GROUP BY x+y;

SELECT x+y, y+z, COUNT(*)
FROM foo
GROUP BY x+y, y+z;

SELECT RTRIM(a) || LTRIM(b), AVG(c), COUNT(c)
FROM foo
GROUP BY RTRIM(a) || LTRIM(b);
```

Invalid Examples

```
SELECT C1, C2
FROM T1
GROUP BY C1;
```

HAVING Clause

Eliminates group rows that do not satisfy a predicate.

Syntax

```
HAVING predicate [, ...]
```

Semantics

predicate	is the same as specified for the ***WHERE clause*** (page 177).

Notes

- Each column referenced in *predicate* must unambiguously reference a grouping column, unless the reference appears within an aggregate function.
- You can use expressions in the HAVING clause.

ORDER BY Clause

Sorts a query result set on one or more columns.

Syntax

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

Semantics

expression	can be: • the name or ordinal number of a SELECT list item • an arbitrary expression formed from columns that do not appear in the SELECT list • a ***CASE*** (page 45) expression

Notes

- The ordinal number refers to the position of the result column, counting from the left beginning at one. This makes it possible to order by a column that does not have a unique name. (You can assign a name to a result column using the AS clause.)
- Vertica uses the ASCII collating sequence to store data and to compare character strings. In general the order is:
  - space
  - numbers
  - upper-case letters
  - lower-case letters

  Special characters collate in between and after the groups mentioned. See `man ascii` for details.
- For integer,bigint, and date/time data types, NULL appears first (smallest) in ascending order.
- For float, boolean, char, and varchar, NULL appears last (largest) in ascending order.

LIMIT Clause

Specifies the maximum number of result set rows to return.

Syntax

```
LIMIT { rows | ALL }
```

Semantics

rows	specifies the maximum number of rows to return
ALL	returns all rows (same as omitting LIMIT)

Notes

- When both LIMIT and ***OFFSET*** (page 182) are specified, specified number of rows are skipped before starting to count the rows to be returned.
- When using LIMIT, use an ***ORDER BY clause*** (page 180) that includes all columns in the select list. Otherwise the query returns an undefined subset of the result set. For example:

C1	C2
1	2
2	1
1	1
2	2

```
SELECT *
FROM T
ORDER BY C1
LIMIT 3;
```

The last row of the result set in this example is undefined in any SQL database but may be consistent enough in other databases for poorly-coded application programs or reports to get the same result set every time. In Vertica however, the distributed nature of the database makes the last row unpredictable, which may cause poorly-coded application programs or reports to get different a result set each time.

OFFSET Clause

Omits a specified number of rows from the beginning of the result set.

Syntax

```
OFFSET rows
```

Semantics

rows	specifies the number of result set rows to omit.

Notes

- When both ***LIMIT*** (page 181) and OFFSET are specified, specified number of rows are skipped before starting to count the rows to be returned.
- When using OFFSET, use an ***ORDER BY clause*** (page 180). Otherwise the query returns an undefined subset of the result set.

SET

The SET command sets one of several run-time parameters.

Syntax

```
SET run-time-parameter
```

Semantics

run-time-parameter	is one of the following: ***DATESTYLE*** (page 183) ***SESSION CHARACTERISTICS*** (page 185) ***TIMEZONE*** (page 186)

DATESTYLE

The SET DATESTYLE command changes the DATESTYLE run-time parameter for the current session.

Syntax

```
SET DATESTYLE TO { value | 'value' } [ ,... ]
```

Semantics

The DATESTYLE parameter can have multiple, non-conflicting values:

Value	Interpretation	Example
MDY	month-day-year	12/17/1997
DMY	day-month-year	17/12/1997
YMD	year-month-day	1997-12-17
ISO	ISO 8601/SQL standard (default)	1997-12-17 07:37:16-08
SQL	traditional style	12/17/1997 07:37:16.00 PST
POSTGRES	original style	Wed Dec 17 07:37:16 1997 PST
GERMAN	regional style	17.12.1997 07:37:16.00 PST

In the SQL and POSTGRES styles, day appears before month if DMY field ordering has been specified, otherwise month appears before day. (See ***Date/Time Constants*** (page 34) for how this setting also affects interpretation of input values.) The table below shows an example.

DATESTYLE	Input Ordering	Example Output
SQL, DMY	day/*month*/*year*	17/12/1997 15:37:16.00 CET
SQL, MDY	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST
Postgres, DMY	*day*/*month*/*year*	Wed 17 Dec 07:37:16 1997 PST

Notes

- The SQL standard requires the use of the ISO 8601 format. The name of the "SQL" output format is a historical accident.

- `INTERVAL` output looks like the input format, except that units like `CENTURY` or `WEEK` are converted to years and days and `AGO` is converted to an appropriate sign. In ISO mode the output looks like
  `[ quantity unit [ ... ] ] [ days ] [ hours:minutes:seconds ]`
- The ***SHOW*** (page 189) command displays the run-time parameters.

Example

SET DATESTYLE TO POSTGRES, MDY;

SESSION CHARACTERISTICS

SET SESSION CHARACTERISTICS sets the transaction characteristics for subsequent transactions of a user session. These are the isolation level and the access mode (read/write or read-only).

Syntax

```
SET SESSION CHARACTERISTICS AS TRANSACTION
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ |
                  READ COMMITTED | READ UNCOMMITTED }
{ READ WRITE | READ ONLY }
```

Semantics

ISOLATION LEVEL	determines what data the transaction can access when other transactions are running concurrently. It does not apply to temporary tables. The isolation level cannot be changed after the first query (SELECT) or DML statement (INSERT, DELETE, UPDATE) of a transaction has been executed.
SERIALIZABLE REPEATABLE READ	are identical in meaning. Both specify SERIALIZABLE isolation, which is the strictest level of SQL transaction isolation and the default in Vertica. This level emulates transactions executed one after another, serially, rather than concurrently. It holds locks, and blocks write operations and is thus not recommended for normal query operations.
READ COMMITTED READ UNCOMMITTED	are identical in meaning. Both specify READ COMMITTED isolation, which allows concurrent transactions and is the default in PostgreSQL and other databases. Use READ COMMITED isolation or Snapshot Isolation for normal query operations but be aware that there is a subtle difference between them (see below).
READ WRITE READ ONLY	determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: INSERT, UPDATE, DELETE, and COPY if the table they would write to is not a temporary table; all CREATE, ALTER, and DROP commands; GRANT, REVOKE, and EXPLAIN if the command it would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

READ COMMITTED vs. Snapshot Isolation

By itself, AT EPOCH LATEST produces purely historical query behavior. However, with READ COMMITTED, SELECT queries return the same result set as AT EPOCH LATEST plus any changes made by the current transaction.

This is standard ANSI SQL semantics for ACID transactions. Any select query within a transaction should see the transactions's own changes regardless of isolation level.

Notes

- SERIALIZABLE isolation does not apply to temporary tables, which are isolated by their transaction scope.
- Applications using SERIALIZABLE must be prepared to retry transactions due to serialization failures.

TIMEZONE

The SET TIMEZONE command changes the TIMEZONE run-time parameter for the current session.

Syntax

```
SET TIMEZONE TO { value | 'value' }
```

Semantics

Value	Interpretation
'PST8PDT'	The time zone for Berkeley, California.
'Europe/Rome'	The time zone for Italy.
'-*n*'	The time zone *n* hours west from UTC . Positive values are east from UTC.
INTERVAL '-*hh*:*mm*' HOUR TO MINUTE	The time zone *hh* hours and *mm* minutes west from UTC.
LOCAL DEFAULT	Set the time zone to your local time zone (the one that the server's operating system defaults to).

Notes

- The default timezone setting is the one specified in the TZ environment variable (for example, `export TZ=EST5EDT`) or if not set, from the Linux kernel's default time zone. See Set the Database Time Zone in the Installation Guide)
- The ***SHOW*** (page 189) command displays the run-time parameters.

Examples

```
SET TIMEZONE TO DEFAULT;
SET TIMEZONE TO '-7'; -- equivalent to PDT
SET TIMEZONE TO '-08:00' HOUR TO MINUTE; -- equivalent to PDT
```

See Also

- ***Time Zone Names for Setting timezone*** (page 187)

Time Zone Names for Setting TIMEZONE

The following time zone names are recognized by Vertica as valid settings for the TIMEZONE run-time parameter. Note that these names are conceptually as well as practically different from the names shown in Time Zone Abbreviations For Input: most of these names imply a local daylight-savings time rule, whereas the former names each represent just a fixed offset from UTC.

In many cases there are several equivalent names for the same zone. These are listed on the same line. The table is primarily sorted by the name of the principal city of the zone.

Time Zone
Africa
America
Antartica
Asia
Atlantic
Australia
CET
EET
Etc/GMT
Europe
Factory
GMT GMT+0 GMT-0 GMT0 Greenwich Etc/GMT Etc/GMT+0 Etc/GMT-0 Etc/GMT0 Etc/Greenwich
Indian
MET
Pacific
UCT Etc/UCT

UTC Universal Zulu Etc/UTC Etc/Universal Etc/Zulu
WET

In addition to the names listed in the table, Vertica will accept time zone names of the form *STDoffset* or *STDoffsetDST*, where *STD* is a zone abbreviation, *offset* is a numeric offset in hours west from UTC, and *DST* is an optional daylight-savings zone abbreviation, assumed to stand for one hour ahead of the given offset. For example, if `EST5EDT` were not already a recognized zone name, it would be accepted and would be functionally equivalent to USA East Coast time. When a daylight-savings zone name is present, it is assumed to be used according to USA time zone rules, so this feature is of limited use outside North America. One should also be wary that this provision can lead to silently accepting bogus input, since there is no check on the reasonableness of the zone abbreviations. For example, `SET TIMEZONE TO FOOBAR0` will work, leaving the system effectively using a rather peculiar abbreviation for GMT.

SHOW

The SHOW command displays run-time parameters for the current session.

Syntax

```
SHOW { name | ALL }
```

Semantics

name	is one of: • DATESTYLE • TIMEZONE
ALL	shows all run-time parameters.

Notes

- The SET Run-Time Parameter command sets the run-time parameters.

Examples

```
=> SHOW ALL;
      name        |  setting
------------------+------------
 vertica_options | 0
 datestyle       | ISO, MDY
 timezone        | US/Eastern
 search_path     | "PUBLIC"
(4 rows)
```

UPDATE

UPDATE replaces the values of the specified columns in all rows for which a specific condition is true. All other columns and rows in the table are unchanged.

Syntax

```
UPDATE table SET column = { expression | DEFAULT } [, ...]
    [ FROM from-list ]
    [ WHERE clause (on page 177) ]
```

Semantics

table	specifies the name of a table in the schema. You cannot UPDATE a projection.
column	specifies the name of a non-key column in the table.
expression	specifies a value to assign to the column. The expression can use the current values of this and other columns in the table. For example: `UPDATE T1 SET C1 = C1+1;`
from-list	A list of table expressions, allowing columns from other tables to appear in the WHERE condition and the update expressions. This is similar to the list of tables that can be specified in the ***FROM Clause*** (on page 176) of a SELECT command. Note that the target table must not appear in the *fromlist*.

Notes

- UPDATE inserts new tuples into the WOS and marks the old tuples for deletion. Thus, be aware of WOS Overload.
- You cannot UPDATE columns that have primary key or foreign key referential integrity constraints.
- In order to use the ***DELETE*** (page 158) or ***UPDATE*** (page 190) commands with a ***WHERE clause*** (page 177), a user must have both SELECT and DELETE privileges on the table.

Examples

```
UPDATE FACT SET PRICE = PRICE - COST * 80 WHERE COST > 100;
UPDATE CUSTOMER SET STATE = 'NH' WHERE CID > 100;
```

SQL Language References

PostgreSQL 8.0.12 Documentation (http://www.postgresql.org/docs/8.0/interactive/sql-commands.html)

BNF Grammar for SQL-99 (http://savage.net.au/SQL/sql-99.bnf.html)

SQL Virtual Tables (Monitoring API)

Vertica provides an API for monitoring various features and functions within a database in the form of virtual tables that can be queried using a limited form of the SELECT statement. You can use external tools to query the virtual tables and act upon the information as desired. For example, when a host failure causes the K-Safety level to fall below a desired level, you can use any means necessary to notify the appropriate personnel.

See the SQL Programmer's Guide section on Using the SQL Monitoring API for more information.

VT_COLUMN_STORAGE

VT_COLUMN_STORAGE monitors the amount of disk storage used by each column of each projection on each node.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
NODE	a VARCHAR value containing the name of the node that is reporting the requested information.
COLUMN	a VARCHAR value containing a projection column name
NUM_ROWS	an INTEGER value containing the number of rows in the column (cardinality)
NUM_BYTES	an INTEGER value containing the disk storage allocation of the column in bytes
PROJECTION_NAME	a VARCHAR value containing the projection name
TABLE	a VARCHAR value containing the associated table name

Example

```
=> \pset expanded
=> SELECT * FROM VT_COLUMN_STORAGE;
-[ RECORD 1 ]---+-----------------------------------------------
timestamp       | 2007-12-05 23:40:11
node            | site01
column_name     | c0_cdr_summary_partition_id
num_rows        | 124437
num_bytes       | 182910
projection_name | p_12_cdr_summary
table           | cdr_summary
-[ RECORD 2 ]---+-----------------------------------------------
timestamp       | 2007-12-05 23:40:11
node            | site01
column_name     | c100_cdr_summary_redirect_line
num_rows        | 124437
```

```
num_bytes       | 8070
projection_name | p_12_cdr_summary
table           | cdr_summary
-[ RECORD 3 ]---+-----------------------------------------------
timestamp       | 2007-12-05 23:40:11
node            | site01
column_name     | c101_cdr_summary_redirect_int_num
num_rows        | 124437
num_bytes       | 17611
projection_name | p_12_cdr_summary
table           | cdr_summary
-[ RECORD 4 ]---+-----------------------------------------------
timestamp       | 2007-12-05 23:40:11
node            | site01
column_name     | c102_cdr_summary_redirect_natr_addr_ind
num_rows        | 124437
num_bytes       | 124514
projection_name | p_12_cdr_summary
table           | cdr_summary
-[ RECORD 5 ]---+-----------------------------------------------
timestamp       | 2007-12-05 23:40:11
node            | site01
column_name     | c103_cdr_summary_orig_redirect_reason_cd
num_rows        | 124437
num_bytes       | 16753
projection_name | p_12_cdr_summary
table           | cdr_summary
                ⋮
```

VT_DISK_STORAGE

VT_DISK_STORAGE monitors the amount of disk storage used by the database on each node.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
NODE	a VARCHAR value containing the name of the node that is reporting the requested information.
DISK_BLK_SIZE	an INTEGER value containing the block size of the disk
USED_BLKS	an INTEGER value containing the number of disk blocks in use
MB_USED	an INTEGER value containing the number of megabytes of disk storage in use
FREE_BLKS	an INTEGER value containing the number of free disk blocks available
MB_FREE	an INTEGER value containing the number of megabytes of free storage available
PERCENTAGE_FREE	an INTEGER value containing the percentage of free disk space remaining

Example

```
=> \pset expanded
=> SELECT * FROM VT_DISK_STORAGE;
-[ RECORD 1 ]---+--------------------
timestamp       | 2007-12-05 23:15:45
node            | site01
disk_blk_size   | 4096
used_blks       | 245869291
mb_used         | 960426
free_blks       | 77953370
mb_free         | 304505
percentage_free | 31%
-[ RECORD 2 ]---+--------------------
timestamp       | 2007-12-05 23:15:45
node            | site02
disk_blk_size   | 4096
used_blks       | 245869291
mb_used         | 960426
free_blks       | 159819606
mb_free         | 624295
percentage_free | 65%
-[ RECORD 3 ]---+--------------------
timestamp       | 2007-12-05 23:15:45
node            | site03
disk_blk_size   | 4096
used_blks       | 245869291
mb_used         | 960426
free_blks       | 163915783
mb_free         | 640296
percentage_free | 66%
-[ RECORD 4 ]---+--------------------
timestamp       | 2007-12-05 23:15:45
node            | site04
disk_blk_size   | 4096
used_blks       | 245869291
mb_used         | 960426
free_blks       | 162976758
mb_free         | 636627
percentage_free | 66%
                ⋮
```

VT_LOAD_STREAMS

VT_LOAD_STREAMS monitors load metrics for each load stream on each node.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
NODE	a VARCHAR value containing the name of the node that is reporting the requested information.
STREAM	a VARCHAR value containing the name of the file being loaded or (if using

	the standard input) the name STDIN.
LOAD_START_TIMESTAMP	a VARCHAR value containing the Linux system time when the load started
ROWS_LOADED	an INTEGER value containing the number of rows loaded
ROWS_REJECTED	an INTEGER value containing the number of rows rejected
BYTES_READ	an INTEGER value containing the number of bytes read from the input file
INPUT_FILE_SIZE	an INTEGER value containing the size of the input file in bytes When using STDIN as input size of input file size is zero (0).
PERCENT_COMPLETE	an INTEGER value containing the percent of the rows in the input file that have been loaded. If using STDIN, this column remains at zero (0) until the COPY statement is complete.

Example

```
=> \pset expanded
```

VT_PROJECTION_STORAGE

VT_PROJECTION_STORAGE monitors the amount of disk storage used by each projection on each node.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
NODE	a VARCHAR value containing the name of the node that is reporting the requested information.
PROJECTION_NAME	a VARCHAR value containing the name of the projection
NUM_COLUMNS	an INTEGER value containing the number of columns in the projection
NUM_ROWS	an INTEGER value containing the number of rows in the projection
NUM_BYTES	an INTEGER value containing the number of bytes of disk storage used by the projection
TABLE_NAME	a VARCHAR value containing associated table name

Example

```
=> \pset expanded
=> SELECT * FROM VT_PROJECTION_STORAGE;
-[ RECORD 1 ]---+---------------------------
timestamp       | 2007-12-05 23:44:00
node            | site01
projection_name | p_12_cdr_summary
num_columns     | 195
num_rows        | 124437
```

```
num_bytes       | 22715067
table_name      | cdr_summary
-[ RECORD 2 ]---+----------------------------
timestamp       | 2007-12-05 23:44:00
node            | site01
projection_name | p_13_cdr_summary
num_columns     | 195
num_rows        | 115357
num_bytes       | 20107846
table_name      | cdr_summary
-[ RECORD 3 ]---+----------------------------
timestamp       | 2007-12-05 23:44:00
node            | site01
projection_name | p_1_lerg6_called
num_columns     | 12
num_rows        | 27000
num_bytes       | 329214
table_name      | lerg6_called
-[ RECORD 4 ]---+----------------------------
timestamp       | 2007-12-05 23:44:00
node            | site01
projection_name | p_0_lerg6_calling
num_columns     | 12
num_rows        | 27000
num_bytes       | 329146
table_name      | lerg6_calling
-[ RECORD 5 ]---+----------------------------
timestamp       | 2007-12-05 23:44:00
node            | site01
projection_name | p_2_lerg6_charge
num_columns     | 12
num_rows        | 27000
num_bytes       | 329188
table_name      | lerg6_charge
                ⋮
```

VT_QUERY_METRICS

VT_QUERY_METRICS monitors the sessions and queries executing on each node.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
NODE	a VARCHAR value containing the name of the node that is reporting the requested information.
ACTIVE_USER_SESSIONS	an INTEGER value containing the number of active user sessions (connections).
ACTIVE_SYS_SESSIONS	an INTEGER value containing the number of active system sessions (connections).

TOTAL_USER_SESSIONS	an INTEGER value containing the total number of user sessions
TOTAL_SYS_SESSIONS	an INTEGER value containing the total number of system sessions.
TOTAL_ACTIVE_SESSIONS	an INTEGER value containing the total number of active user and system sessions.
TOTAL_SESSIONS	an INTEGER value containing the total number of user and system sessions.
QUERIES_CURRENTLY_RUNNING	an INTEGER value containing the number of queries currently running
TOTAL_QUERIES_EXECUTED	an INTEGER value containing the total number of queries executed

Example

```
=> \pset expanded
=> SELECT * FROM VT_QUERY_METRICS;
-[ RECORD 1 ]-------------+--------------------
timestamp                 | 2007-12-05 23:45:48
node                      | site01
active_user_sessions      | 1
active_sys_sessions       | 2
total_user_sessions       | 20
total_sys_sessions        | 81
total_active_sessions     | 3
total_sessions            | 3
queries_currently_running | 0
total_queries_executed    | 132
-[ RECORD 2 ]-------------+--------------------
timestamp                 | 2007-12-05 23:45:48
node                      | site02
active_user_sessions      | 0
active_sys_sessions       | 3
total_user_sessions       | 0
total_sys_sessions        | 327
total_active_sessions     | 3
total_sessions            | 3
queries_currently_running | 0
total_queries_executed    | 0
-[ RECORD 3 ]-------------+--------------------
timestamp                 | 2007-12-05 23:45:48
node                      | site03
active_user_sessions      | 0
active_sys_sessions       | 3
total_user_sessions       | 0
total_sys_sessions        | 327
total_active_sessions     | 3
total_sessions            | 3
queries_currently_running | 0
total_queries_executed    | 0
-[ RECORD 4 ]-------------+--------------------
```

```
timestamp                 | 2007-12-05 23:45:48
node                      | site04
active_user_sessions      | 0
active_sys_sessions       | 3
total_user_sessions       | 0
total_sys_sessions        | 327
total_active_sessions     | 3
total_sessions            | 3
queries_currently_running | 0
total_queries_executed    | 0
                          ⋮
```

VT_RESOURCE_USAGE

VT_RESOURCE_USAGE monitors system resource management on each node.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
NODE	a VARCHAR value containing the name of the node that is reporting the requested information.
NUM_REQUESTS	an INTEGER value containing the cumulative number of requests for threads, file handles, and memory (in kilobytes).
NUM_LOCAL_REQUESTS	an INTEGER value containing the cumulative number of local requests
CURRENT_REQ_QUE_DEPTH	an INTEGER value containing the current request queue depth
CURRENT_NUM_ACTIVE_THREADS	an INTEGER value containing the current number of active threads
CURRENT_NUM_OPEN_FILE_HANDLES	an INTEGER value containing the current number of open file handles

Example

```
=> \pset expanded
=> SELECT * FROM VT_RESOURCE_USAGE;
-[ RECORD 1 ]-----------------+--------------------
timestamp                     | 2007-12-26 18:42:55
node                          | site01
num_requests                  | 1
num_local_requests            | 0
current_req_que_depth         | 0
current_num_active_threads    | 12
current_num_open_file_handles | 8
-[ RECORD 2 ]-----------------+--------------------
timestamp                     | 2007-12-26 18:42:55
node                          | site02
```

```
num_requests                 | 1
num_local_requests           | 1
current_req_que_depth        | 0
current_num_active_threads   | 28
current_num_open_file_handles | 325
-[ RECORD 3 ]----------------+--------------------
timestamp                    | 2007-12-26 18:42:55
node                         | site03
num_requests                 | 1
num_local_requests           | 0
current_req_que_depth        | 0
current_num_active_threads   | 12
current_num_open_file_handles | 8
-[ RECORD 4 ]----------------+--------------------
timestamp                    | 2007-12-26 18:42:55
node                         | site04
num_requests                 | 0
num_local_requests           | 0
current_req_que_depth        | 0
current_num_active_threads   | 0
current_num_open_file_handles | 0
```

VT_SYSTEM

VT_SYSTEM monitors the overall state of the database.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
CURRENT_EPOCH	an INTEGER value containing the current epoch number
K_SAFETY_LEVEL	an INTEGER value containing the K-Safety level of the database (see ***MARK_DESIGN_KSAFE*** (page 133))
CATALOG_REV_NUM	an INTEGER value containing the catalog version number
TUPLE_MOVER_MODE	a VARCHAR value containing the tuple mover mode (`bulk mode` or `normal`) (see ***SELECT SET_ATM_MODE*** (see "SET_ATM_MODE" on page 135))

Example

```
=> \pset expanded
=> SELECT * FROM VT_SYSTEM;
-[ RECORD 1 ]----+--------------------
timestamp        | 2007-12-05 23:14:41
current_epoch    | 28
k_safety_level   | 1
catalog_rev_num  | 356
tuple_mover_mode | normal
```

VT_TABLE_STORAGE

VT_TABLE_STORAGE monitors the amount of disk storage used by each table on each node.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
NODE	a VARCHAR value containing the name of the node that is reporting the requested information.
TABLE_NAME	a VARCHAR value containing the table name
NUM_PROJECTIONS	an INTEGER value containing the number of projections using columns of the table
NUM_COLUMNS	an INTEGER value containing the number of columns in the table
NUM_ROWS	an INTEGER value containing the number of rows in the table (cardinality)
NUM_BYTES	an INTEGER value containing the number of bytes used to store the projections

Example

```
=> \pset expanded
=> SELECT * FROM VT_TABLE_STORAGE;
-[ RECORD 1 ]---+---------------------
timestamp       | 2007-12-05 23:47:36
node            | site01
table_name      | cdr_summary
num_projections | 2
num_columns     | 195
num_rows        | 124437
num_bytes       | 42822913
-[ RECORD 2 ]---+---------------------
timestamp       | 2007-12-05 23:47:36
node            | site01
table_name      | lerg6_called
num_projections | 4
num_columns     | 12
num_rows        | 27000
num_bytes       | 329214
-[ RECORD 3 ]---+---------------------
timestamp       | 2007-12-05 23:47:36
node            | site01
table_name      | lerg6_calling
num_projections | 4
num_columns     | 12
num_rows        | 27000
num_bytes       | 329146
-[ RECORD 4 ]---+---------------------
timestamp       | 2007-12-05 23:47:36
node            | site01
```

```
table_name      | lerg6_charge
num_projections | 4
num_columns     | 12
num_rows        | 27000
num_bytes       | 329188
-[ RECORD 5 ]---+----------------------
timestamp       | 2007-12-05 23:47:36
node            | site01
table_name      | lerg6_enriched_called
num_projections | 4
num_columns     | 12
num_rows        | 27000
num_bytes       | 329173
                ⋮
```

VT_TUPLE_MOVER

VT_TUPLE_MOVER monitors the status of the Tuple Mover on each node.

No output from VT_TUPLE_MOVER means that the Tuple Mover is not performing an operation.

Column Name	Description
TIMESTAMP	a VARCHAR value containing the Linux system time of query execution in a format that can be used as a ***Date/Time Expression*** (page 47).
NODE	a VARCHAR value containing the name of the node that is reporting the requested information.

OPERATION	a VARCHAR value containing one of the following:
	`Moveout`
	`Mergeout`
	`Analyze Statistics`

STATUS	a VARCHAR value containing `Running` or an empty string to indicate 'not running.'
PROJ	a VARCHAR value containing the name of the projection being processed
START_EPOCH	an INTEGER value containing the first epoch of the mergeout operation (not applicable for other operations)
STOP_EPOCH	an INTEGER value containing the last epoch of the mergeout operation (not applicable for other operations)
NUM_MINI_ROS	an INTEGER value containing the number of ROS Containers
SIZE	an INTEGER value containing the size in bytes of all ROS containers in the mergeout operation (not applicable for other operations)

PLAN	a VARCHAR value containing one of the following string values:
	`Moveout`
	`Mergeout`
	`Analyze`
	`Replay Delete`

Example

```
=> \pset expanded
```

Index

S

T

U

V

W