HP Vertica Analytics Platform 6.1.x

SQL Reference Manual

Doc Revision 3

Date of Publication: Monday, October 28, 2013

Contents

SQL Data Types 71

SQL Functions 117

Syntax Conventions

The following are the syntax conventions used in the HP Vertica documentation.

Syntax Convention	Description
`Text without brackets/braces`	Indicates content you type, as shown.
`< Text inside angle brackets >`	Represents a placeholder for which you must supply a value. The variable is usually shown in italics. See *Placeholders* below.
`[ Text inside brackets ]`	Indicates optional items; for example, CREATE TABLE [*schema_name.*]*table_name* The brackets indicate that the *schema_name* is optional. Do not type the square brackets.
`{ Text inside braces }`	Indicates a set of options from which you choose one; for example: `QUOTES { ON | OFF }` indicates that exactly one of ON or OFF must be provided. You do not type the braces: `QUOTES ON`
Backslash \	Represents a continuation character used to indicate text that is too long to fit on a single line.
Ellipses . . .	Indicate a repetition of the previous parameter. For example, `option[,...]` means that you can enter multiple, comma-separated options. Showing ellipses in code examples might also mean that part of the text has been omitted for readability, such as in multi-row result sets.
`Indentation`	Is an attempt to maximize readability; SQL is a free-form language.
`Placeholders`	Represent items that must be replaced with appropriate identifiers or expressions and are usually shown in italics.
Vertical bar |	Is a separator for mutually exclusive items. For example: `[ ASC | DESC ]` Choose one or neither. You do not type the square brackets.

SQL Overview

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

Note: In HP Vertica, use a semicolon to end a statement or to combine multiple statements on one line.

HP Vertica Support for ANSI SQL Standards

HP Vertica SQL supports a subset of ANSI SQL-99.

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

Support for Historical Queries

Unlike most databases, the ***DELETE*** (page 807) command in HP Vertica does not delete data; it marks records as deleted. The ***UPDATE*** (page 929) command performs an INSERT and a DELETE. This behavior is necessary for historical queries. See Historical (Snapshot) Queries in the Programmer's Guide.

Joins

HP Vertica supports typical data warehousing query joins. For details, see Joins in the Programmer's Guide.

HP Vertica also provides the ***INTERPOLATE*** (page 61) predicate, which allows for a special type of join. The event series join is an HP Vertica SQL extension that lets you analyze two event series when their measurement intervals don't align precisely—such as when timestamps don't match. These joins provide a natural and efficient way to query misaligned event data directly, rather than having to normalize the series to the same measurement interval. See Event Series Joins in the Programmer's Guide for details.

Transactions

Session-scoped isolation levels determine transaction characteristics for transactions within a specific user session. You set them through the ***SET SESSION CHARACTERISTICS*** (page 914) command. Specifically, they determine what data a transaction can access when other transactions are running concurrently. See Transactions in the Concepts Guide.

System Limits

This section describes the system limits on the size and number of objects in an HP Vertica database. In most cases, computer memory and disk drive are the limiting factors.

Item	Limit
Number of nodes	Maximum 128 (without HP Vertica assistance).
Database size	Approximates the number of files times the file size on a platform, depending on the maximum disk configuration.
Table size	2^64 rows per node, or 2^63 bytes per column, whichever is smaller.
Row size	32 MB. The row size is approximately the sum of its maximum column sizes, where, for example, a VARCHAR(80) has a maximum size of 80 bytes.
Key size	Limited only by row size
Number of tables/projections per database	Limited by physical RAM, as the catalog must fit in memory.
Number of concurrent connections per node	Default of 50, limited by physical RAM (or threads per process), typically 1024.
Number of concurrent connections per cluster	Limited by physical RAM of a single node (or threads per process), typically 1024.
Number of columns per table	1600.
Number of rows per load	2^63.
Number of partitions	1024. While HP Vertica supports a maximum of 1024 partitions, few, if any, organizations will need to approach that maximum. Fewer partitions are likely to meet your business needs, while also ensuring maximum performance. Many customers, for example, partition their data by month, bringing their partition count to 12. HP Vertica recommends you keep the number of partitions between 10 and 20 to achieve excellent performance.
Length for a fixed-length column	65000 bytes.
Length for a variable-length column	65000 bytes.
Length of basic names	128 bytes. Basic names include table names, column names, etc.
Query length	No limit.
Depth of nesting subqueries	Unlimited in FROM, WHERE, or HAVING clause.

SQL Language Elements

This chapter presents detailed descriptions of the language elements and conventions of HP Vertica SQL.

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. Some unreserved keywords can be used in statements by preceding them with AS. For example, SOURCE is a keyword, but is not reserved, and you can use it as follows:

```
VMART=> select my_node AS SOURCE from nodes;
```

Keywords

Keyword are words that are specially handled by the grammar. Every SQL statement contains one or more keywords.

Begins with	Keyword
A	ABORT, ABSOLUTE, ACCESS, ACCESRANK, ACCOUNT, ACTION, ADD, ADMIN, AFTER, AGGREGATE, ALL, ALSO, ALTER, ANALYSE, ANALYZE, AND, ANY, ARRAY, AS, ASC, ASSERTION, ASSIGNMENT, AT, AUTHORIZATION, AUTO, AUTO_INCREMENT, AVAILABLE
B	BACKWARD, BEFORE, BEGIN, BETWEEN, BIGINT, BINARY, BIT, BLOCK_DICT, BLOCKDICT_COMP, BOOLEAN, BOTH, BY, BYTEA, BZIP
C	CACHE, CALLED, CASCADE, CASE, CAST, CATALOGPATH, CHAIN, CHAR, CHAR_LENGTH, CHARACTER, CHARACTER_LENGTH, CHARACTERISTICS, CHARACTERS, CHECK, CHECKPOINT, CLASS, CLOSE, CLUSTER, COLLATE, COLUMN, COLUMNS_COUNT, COMMENT, COMMIT, COMMITTED, COMMONDELTA_COMP, CONNECT, CONSTRAINT, CONSTRAINTS, COPY, CORRELATION, CREATE, CREATEDB, CREATEUSER, CROSS, CSV, CURRENT, CURRENT_DATABASE, CURRENT_DATE, CURRENT_SCHEMA, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER, CURSOR, CYCLE
D	DATA, DATABASE, DATAPATH, DATE, DATEDIFF, DATETIME, DAY, DEALLOCATE, DEC, DECIMAL, DECLARE, DECODE, DEFAULT, DEFAULTS, DEFERRABLE, DEFERRED, DEFINE, DEFINER, DELETE, DELIMITER, DELIMITERS, DELTARANGE_COMP, DELTARANGE_COMP_SP, DELTAVAL, DESC, DETERMINES, DIRECT, DIRECTCOLS, DIRECTGROUPED, DIRECTPROJ, DISABLE, DISCONNECT, DISTINCT, DISTVALINDEX, DO, DOMAIN, DOUBLE, DROP, DURABLE

E	EACH, ELSE, ENABLE, ENABLED, ENCLOSED, ENCODED, ENCODING, ENCRYPTED, END, ENFORCELENGTH, EPHEMERAL, EPOCH, ERROR, ESCAPE, EVENT, EVENTS, EXCEPT, EXCEPTIONS, EXCLUDE, EXCLUDING, EXCLUSIVE, EXECUTE, EXISTS, EXPIRE, EXPLAIN, EXPORT, EXTERNAL, EXTRACT
F	FAILED_LOGIN_ATTEMPTS, FALSE, FETCH, FILLER, FIRST, FLOAT, FOLLOWING, FOR, FORCE, FOREIGN, FORMAT, FORWARD, FREEZE, FROM, FULL, FUNCTION
G	GCDDELTA, GLOBAL, GRANT, GROUP, GROUPED, GZIP
H	HANDLER, HASH, HAVING, HOLD, HOSTNAME, HOUR, HOURS
I	IDENTIFIED, IDENTITY, IF, IGNORE, ILIKE, ILIKEB, IMMEDIATE, IMMUTABLE, IMPLICIT, IN, INCLUDING, INCREMENT, INDEX, INHERITS, INITIALLY, INNER, INOUT, INPUT, INSENSITIVE, INSERT, INSTEAD, INT, INTEGER, INTERPOLATE, INTERSECT, INTERVAL, INTERVALYM, INTO, INVOKER, IS, ISNULL, ISOLATION
J	JOIN
K	KEY, KSAFE
L	LANCOMPILER, LANGUAGE, LARGE, LAST, LATEST, LEADING, LEFT, LESS, LEVEL, LIBRARY, LIKE, LIKEB, LIMIT, LISTEN, LOAD, LOCAL, LOCALTIME, LOCALTIMESTAMP, LOCATION, LOCK
M	MANAGED, MATCH, MAXCONCURRENCY, MAXMEMORYSIZE, MAXVALUE, MEMORYCAP, MEMORYSIZE, MERGE, MERGEOUT, MICROSECONDS, MILLISECONDS, MINUTE, MINUTES, MINVALUE, MODE, MONEY, MONTH, MOVE, MOVEOUT
N	NAME, NATIONAL, NATIVE, NATURAL, NCHAR, NEW, NEXT, NO, NOCREATEDB, NOCREATEUSER, NODE, NODES, NONE, NOT, NOTHING, NOTIFY, NOTNULL, NOWAIT, NULL, NULLCOLS, NULLS, NULLSEQUAL, NULLIF, NUMBER, NUMERIC
O	OBJECT, OCTETS, OF, OFF, OFFSET, OIDS, OLD, ON, ONLY, OPERATOR, OPTION, OR, ORDER, OTHERS, OUT, OUTER, OVER, OVERLAPS, OVERLAY, OWNER
P	PARTIAL, PARTITION, PASSWORD, PASSWORD_GRACE_TIME, PASSWORD_LIFE_TIME, PASSWORD_LOCK_TIME, PASSWORD_MAX_LENGTH, PASSWORD_MIN_DIGITS, PASSWORD_MIN_LENGTH, PASSWORD_MIN_LETTERS, PASSWORD_MIN_LOWERCASE_LETTERS, PASSWORD_MIN_SYMBOLS, PASSWORD_MIN_UPPERCASE_LETTERS,PASSWORD_REUSE_MAX, PASSWORD_REUSE_TIME, PATTERN, PERCENT, PERMANENT, PINNED, PLACING, PLANNEDCONCURRENCY, POOL, POSITION, PRECEDING, PRECISION, PREPARE, PRESERVE, PREVIOUS, PRIMARY, PRIOR, PRIORITY, PRIVILEGES, PROCEDURAL, PROCEDURE, PROFILE, PROJECTION
Q	QUEUETIMEOUT, QUOTE

R	RANGE, RAW, READ, REAL, RECHECK, RECORD, RECOVER, REFERENCES, REFRESH, REINDEX, REJECTED, REJECTMAX, RELATIVE, RELEASE, RENAME, REPEATABLE, REPLACE, RESET, RESOURCE, RESTART, RESTRICT, RESULTS, RETURN, RETURNREJECTED, REVOKE, RIGHT, RLE, ROLE, ROLES, ROLLBACK, ROW, ROWS, RULE, RUNTIMECAP
S	SAMPLE, SAVEPOINT, SCHEMA, SCROLL, SECOND, SECONDS, SECURITY, SEGMENTED, SELECT, SEQUENCE, SERIALIZABLE, SESSION, SESSION_USER, SET, SETOF, SHARE, SHOW, SIMILAR, SIMPLE, SINGLEINITIATOR, SITE, SITES, SKIP, SMALLDATETIME, SMALLINT, SOME, SOURCE, SPLIT, STABLE, START, STATEMENT, STATISTICS, STDERR, STDIN, STDOUT, STORAGE, STREAM, STRICT, SUBSTRING, SYSDATE, SYSID, SYSTEM
T	TABLE, TABLESPACE, TEMP, TEMPLATE, TEMPORARY, TEMPSPACECAP, TERMINATOR, THAN, THEN, TIES, TIME, TIMESERIES, TIMESTAMP, TIMESTAMPADD, TIMESTAMPDIFF, TIMESTAMPTZ, TIMETZ, TIMEZONE, TINYINT, TO, TOAST, TRAILING, TRANSACTION, TRANSFORM, TREAT, TRICKLE, TRIGGER, TRIM, TRUE, TRUNCATE, TRUSTED, TUNING, TYPE
U	UNBOUNDED, UNCOMMITTED, UNCOMPRESSED, UNENCRYPTED, UNION, UNIQUE, UNKNOWN, UNLIMITED, UNLISTEN, UNLOCK, UNSEGMENTED, UNTIL, UPDATE, USAGE, USER, USING
V	VACUUM, VALIDATOR, VALINDEX, VALUE, VALUES, VARBINARY, VARCHAR, VARCHAR2, VARYING, VERBOSE, VERTICA, VIEW, VOLATILE
W	WAIT, WHEN, WHERE, WINDOW, WITH, WITHIN, WITHOUT, WORK, WRITE
Y	YEAR
Z	ZONE

Reserved Words

Many SQL keywords are also reserved words, but a reserved word is not necessarily a keyword. For example, a reserved word might be reserved for other/future use. In HP Vertica, reserved words can be used anywhere identifiers can be used, as long as you double-quote them.

Begins with	Reserved Word
A	ALL, ANALYSE, ANALYZE, AND, ANY, ARRAY, AS, ASC
B	BINARY, BOTH
C	CASE, CAST, CHECK, COLUMN, CONSTRAINT, CORRELATION, CREATE, CURRENT_DATABASE, CURRENT_DATE, CURRENT_SCHEMA, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER
D	DEFAULT, DEFERRABLE, DESC, DISTINCT, DO
E	ELSE, ENCODED, END, EXCEPT
F	FALSE, FOR, FOREIGN, FROM

G	GRANT, GROUP, GROUPED
H	HAVING
I	IN, INITIALLY, INTERSECT, INTERVAL, INTERVALYM, INTO
J	JOIN
K	KSAFE
L	LEADING, LIMIT, LOCALTIME, LOCALTIMESTAMP
M	MATCH
N	NEW, NOT, NULL, NULLSEQUAL
O	OFF, OFFSET, OLD, ON, ONLY, OR, ORDER
P	PINNED, PLACING, PRIMARY, PROJECTION
R	REFERENCES
S	SCHEMA, SEGMENTED, SELECT, SESSION_USER, SOME, SYSDATE
T	TABLE, THEN, TIMESERIES, TO, TRAILING, TRUE
U	UNBOUNDED, UNION, UNIQUE, UNSEGMENTED, USER, USING
W	WHEN, WHERE, WINDOW, WITH, WITHIN

Identifiers

Identifiers (names) of objects such as schema, table, projection, column names, and so on, can be up to 128 bytes in length.

Unquoted Identifiers

Unquoted SQL identifiers must begin with one of the following:

- Letters such as A-Z or a-z, including letters with diacritical marks and non-Latin letters)
- Underscore (_)

Subsequent characters in an identifier can be:

- Letters
- Digits(0-9)
- Dollar sign ($). Dollar sign is not allowed in identifiers according to the SQL standard and could cause application portability problems.
- Underscore (_)

Quoted Identifiers

Identifiers enclosed in double quote (") characters can contain any character. If you want to include a double quote, you need a pair of them; for example """". You can use names that would otherwise be invalid, such as names that include only numeric characters ("123") or contain space characters, punctuation marks, keywords, and so on; for example, `CREATE SEQUENCE "my sequence!";`

Double quotes are required for non-alphanumerics and SQL keywords such as "1time", "Next week" and "Select".

Note: Identifiers are not case-sensitive. Thus, identifiers "ABC", "ABc", and "aBc" are synonymous, as are ABC, ABc, and aBc.

Non-ASCII characters

HP Vertica accepts non-ASCII UTF-8 Unicode characters for table names, column names, and other ***identifiers*** (page 22), extending the cases in which upper/lower case distinctions are ignored (case-folded) to all alphabets, including Latin, Cyrillic, and Greek.

Identifiers are stored as created

SQL identifiers, such as table and column names, are no longer converted to lowercase. They are stored as created, and references to them are resolved using case-insensitive compares. It is not necessary to double quote mixed-case identifiers. For example, The following statement creates table ALLCAPS.

```
=> CREATE TABLE ALLCAPS(c1 varchar(30));
=> INSERT INTO ALLCAPS values('upper case');
```

The following statements are variations of the same query and all return identical results:

```
=> SELECT * FROM ALLCAPS;
=> SELECT * FROM allcaps;
=> SELECT * FROM "allcaps";
```

All three commands return the same result:

```
     c1
------------
 upper case
(1 row)
```

Note that the system returns an error if you try to create table AllCaps:

```
=> CREATE TABLE allcaps(c1 varchar(30));
   ROLLBACK:  table "AllCaps" already exists
```

See ***QUOTE_IDENT*** (page 391) for additional information.

Case-sensitive System Tables

The V_CATALOG.TABLES (page 978).TABLE_SCHEMA and TABLE_NAME columns are case sensitive when used with an equality (=) predicate in queries. For example, given the following sample schema, if you execute a query using the = predicate, HP Vertica returns 0 rows:

```
=> CREATE SCHEMA SS;
```

```
=> CREATE TABLE SS.TT (c1 int);
=> INSERT INTO ss.tt VALUES (1);
=> SELECT table_schema, table_name FROM v_catalog.tables WHERE table_schema ='ss';

 table_schema | table_name
--------------+------------
(0 rows)
```

TIP: Use the case-insensitive **ILIKE** predicate to return the expected results.

```
=> SELECT table_schema, table_name FROM v_catalog.tables WHERE table_schema ILIKE
'ss';
 table_schema | table_name
--------------+------------
 SS           | TT
(1 row)
```

Literals

Literals are numbers or strings used in SQL as constants. Literals are included in the select-list, along with expressions and built-in functions and can also be constants.

HP Vertica provides support for number-type literals (integers and numerics), string literals, VARBINARY string literals, and date/time literals. The various string literal formats are discussed in this section.

Number-type Literals

There are three types of numbers in HP Vertica: Integers, numerics, and floats.

- ***Integers*** (page 107) are whole numbers less than 2^63 and must be digits.
- ***Numerics*** (page 107) are whole numbers larger than 2^63 or that include a decimal point with a precision and a scale. Numerics can contain exponents. Numbers that begin with 0x are hexadecimal numerics.

Numeric-type values can also be generated using casts from character strings. This is a more general syntax. See the Examples section below, as well as ***Data Type Coercion Operators (CAST)*** (page 45).

Syntax

digits
digits.[*digits*] | [*digits*].*digits*
digits e[+-]*digits* | [*digits*].*digits* e[+-]*digits* | *digits*.[*digits*] e[+-]*digits*

Parameters

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

e	Represents an exponent marker.

Notes

- At least one digit must follow the exponent marker (`e`), if `e` is present.
- There cannot be any spaces or other characters embedded in the constant.
- Leading plus (+) or minus (-) signs are not considered part of the constant; they are unary operators applied to the constant.
- In most cases a numeric-type 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 45).
- Floating point literals are not supported. If you specifically need to specify a float, you can cast as described in ***Data Type Coercion Operators (CAST)*** (page 45).
- HP Vertica follows the IEEE specification for floating point, including NaN (not a number) and Infinity (Inf).
- 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. See ***Numeric Expressions*** (page 58) for examples.
- Dividing INTEGERS (x / y) yields a NUMERIC result. You can use the // operator to truncate the result to a whole number.

Examples

The following are examples of number-type literals:

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

Scientific notation:

```
=> SELECT NUMERIC '1e10';
  ?column?
-------------
 10000000000
(1 row)
```

`BINARY` scaling:

```
=> SELECT NUMERIC '1p10';
 ?column?
----------
     1024
(1 row)

=> SELECT FLOAT 'Infinity';
?column?
```

```
----------
 Infinity
(1 row)
```

The following examples illustrated using the / and // operators to divide integers:

```
VMart=> SELECT 40/25;
       ?column?
----------------------
 1.600000000000000000
(1 row)


VMart=> SELECT 40//25;
 ?column?
----------
        1
(1 row)
```

See Also

Data Type Coercion (page 112)

String Literals

String literals are string values surrounded by single or double quotes. Double-quoted strings are subject to the backslash, but single-quoted strings do not require a backslash, except for \' and \\.

You can embed single quotes and backslashes into single-quoted strings.

To include other backslash (escape) sequences, such as \t (tab), you must use the double-quoted form.

Single quoted strings require a preceding space between them and the preceding word because single quotes are allowed in identifiers.

See Also

STANDARD_CONFORMING_STRINGS (page 920)

ESCAPE_STRING_WARNING (page 905)

Internationalization Parameters and Implement Locales for International Data Sets in the Administrator's Guide

Character String Literals

Character string literals are a sequence of characters from a predefined character set and are enclosed by single quotes. If the single quote is part of the sequence, it must be doubled as `''''`.

Syntax

'*characters*'

Parameters

characters	Arbitrary sequence of characters bounded by single quotes (')

Single Quotes in a String

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

```
=> SELECT 'Chester''s gorilla';
     ?column?
-------------------
 Chester's gorilla
(1 row)
```

Standard Conforming Strings and Escape Characters

HP Vertica uses standard conforming strings as specified in the SQL standard, which means that backslashes are treated as string literals, not escape characters.

Note: Earlier versions of HP Vertica did not use standard conforming strings, and backslashes were always considered escape sequences. To revert to this older behavior, set the `StandardConformingStrings` parameter to '0', as described in Configuration Parameters in the Administrator's Guide.

Examples

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

=> SELECT 'This \is a string';
   WARNING:  nonstandard use of escape in a string literal at character 8
   HINT:  Use the escape string syntax for escapes, e.g., E'\r\n'.
     ?column?
------------------
 This is a string
(1 row)

vmartdb=> SELECT E'This \is a string';
     ?column?
------------------
 This is a string

=> SELECT E'This is a \n new line';
       ?column?
```

```
----------------------
 This is a
 new line
(1 row)

=> SELECT 'String''s characters';
      ?column?
--------------------
 String's characters
(1 row)
```

See Also

STANDARD_CONFORMING_STRINGS (page 920) and ***ESCAPE_STRING_WARNING*** (page 905) in the SQL Reference Manual

Internationalization Parameters and Implement Locales for International Data Sets in the Administrator's Guide

Dollar-quoted String Literals

Dollar-quoted string literals are rarely used but are provided here for your convenience.

The standard syntax for specifying string literals can be difficult to understand. To allow more readable queries in such situations, HP 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.

Syntax

```
$$characters$$
```

Parameters

characters	Arbitrary sequence of 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 the preceding word by whitespace; otherwise the dollar quoting delimiter would be taken as part of the preceding identifier.

Examples

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

```
=> SELECT 'SELECT 'fact';';
ERROR:  syntax error at or near "';'" at character 21
LINE 1: SELECT 'SELECT 'fact';';

=> SELECT 'SELECT $$fact';$$;
   ?column?
---------------
 SELECT $$fact
(1 row)

=> SELECT 'SELECT ''fact'';';
    ?column?
----------------
 SELECT 'fact';
(1 row)
```

Unicode String Literals

Syntax

U&'*characters*' [UESCAPE '<*Unicode escape character*>']

Parameters

characters	Arbitrary sequence of UTF-8 characters bounded by single quotes (')
Unicode escape character	A single character from the source language character set other than a hexit, plus sign (+), quote ('), double quote ("), or white space

Using standard conforming strings

With StandardConformingStrings enabled, HP Vertica supports SQL standard Unicode character string literals (the character set is UTF-8 only).

Before you enter a Unicode character string literal, enable standard conforming strings in one of the following ways.

- To enable for all sessions, update the StandardConformingStrings configuration parameter. See Configuration Parameters in the Administrator's Guide.
- To treats backslashes as escape characters for the current session, use the ***SET STANDARD_CONFORMING_STRINGS*** (page 920) statement.

See also ***Extended String Literals*** (page 31).

Examples

To enter a Unicode character in hexadecimal, such as the Russian phrase for "thank you, use the following syntax:

```
=> SET STANDARD_CONFORMING_STRINGS TO ON;


=> SELECT U&'\0441\043F\0430\0441\0438\0431\043E' as 'thank you';
 thank you
-----------
 спасибо
(1 row)
```

To enter in hexadecimal the German word 'mude' (where u is really u-umlaut):

```
=> SELECT U&'m\00fcde';
?column?
----------
müde
(1 row)

=> SELECT 'ü';
?column?
----------
ü
(1 row)
```

To enter into hexadecimal the LINEAR B IDEOGRAM B240 WHEELED CHARIOT:

```
=> SELECT E'\xF0\x90\x83\x8C';
 ?column?
----------
```

(wheeled chariot character)

```
(1 row)
```

Note: Not all fonts support this character.

See Also

STANDARD_CONFORMING_STRINGS (page 920) and ***ESCAPE_STRING_WARNING*** (page 905) in the SQL Reference Manual

Internationalization Parameters and Implement Locales for International Data Sets in the Administrator's Guide

VARBINARY String Literals

VARBINARY string literals allow you to specify hexadecimal or binary digits in a string literal.

Syntax

```
X'<hexadecimal digits>'
B'<binary digits>'
```

Parameters

X	Specifies hexadecimal digits. The <*hexadecimal digits*> string must be enclosed in single quotes (').

B	Specifies binary digits. The <*binary digits*> string must be enclosed in single quotes (').

Examples

```
=> SELECT X'abcd';
 ?column?
----------
 \253\315
(1 row)


=> SELECT B'101100';
 ?column?
----------
 ,
(1 row)
```

Extended String Literals

Syntax

E'*characters*'

Parameters

characters	Arbitrary sequence of characters bounded by single quotes (')

You can use C-style backslash sequence in extended string literals, which are an extension to the SQL standard. You specify an extended string literal by writing the letter E as a prefix (before the opening single quote); for example:

```
E'extended character string\n'
```

Within an extended string, the backslash character (\) starts a C-style backslash sequence, in which the combination of backslash and following character or numbers represent a special byte value, as shown in the following list. Any other character following a backslash is taken literally; for example, to include a backslash character, write two backslashes (\\).

- \\ 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
- \x##, where ## is a 1 or 2-digit hexadecimal number; for example \x07 is a tab

- `\###`, where `###` is a 1, 2, or 3-digit octal number representing a byte with the corresponding code.

When an extended string literal is concatenated across lines, write only `E` before the first opening quote:

```
=> SELECT E'first part o'
->    'f a long line';
         ?column?
--------------------------
 first part of a long line
(1 row)
```

Two adjacent single quotes are used as one single quote:

```
=> SELECT 'Aren''t string literals fun?';
          ?column?
----------------------------
 Aren't string literals fun?
(1 row)
```

Standard Conforming Strings and Escape Characters

When interpreting commands, such as those entered in vsql or in queries passed via JDBC or ODBC, HP Vertica uses standard conforming strings as specified in the SQL standard. In standard conforming strings, backslashes are treated as string literals (ordinary characters), not escape characters.

Note: Text read in from files or streams (such as the data inserted using the ***COPY*** (page 699) statement) are not treated as literal strings. The COPY command defines its own escape characters for the data it reads. See the ***COPY*** (page 699) statement documentation for details.

In HP Vertica databases prior to 4.0, the standard conforming strings was not on by default, and backslashes were considered escape sequences. After 4.0, escape sequences, including Windows path names, did not work as they had previously. For example, the TAB character `'\t'` is two characters: `'\'` and `'t'`.

`E'...'` is the ***Extended character string literal*** (page 31) format, so to treat backslashes as escape characters, use `E'\t'`.

The following options are available, but HP recommends that you migrate your application to use standard conforming strings at your earliest convenience, after warnings have been addressed.

- To revert to pre 4.0 behavior, set the StandardConformingStrings parameter to '0', as described in Configuration Parameters in the Administrator's Guide.
- To enable standard conforming strings permanently, set the StandardConformingStrings parameter to '1', as described in the procedure in the section, "Identifying Strings that are not Standard Conforming," below.
- To enable standard conforming strings per session, use ***SET STANDARD_CONFORMING_STRING TO ON*** (page 920), which treats backslashes as escape characters for the current session only.

The two sections that follow help you identify issues between HP Vertica 3.5 and 4.0.

Identifying Strings that are not Standard Conforming

The following procedure can be used to identify non-standard conforming strings in your application so that you can convert them into standard conforming strings:

1. Be sure the StandardConformingStrings parameter is off, as described in Internationalization Parameters in the Administrator's Guide.

   ```
   => SELECT SET_CONFIG_PARAMETER ('StandardConformingStrings' ,'0');
   ```

   Note: HP recommends that you migrate your application to use Standard Conforming Strings at your earliest convenience.

2. Turn on the EscapeStringWarning parameter. (ON is the default in HP Vertica Version 4.0 and later.)

   ```
   => SELECT SET_CONFIG_PARAMETER ('EscapeStringWarning','1');
   ```

   HP Vertica now returns a warning each time it encounters an escape string within a string literal. For example, HP Vertica interprets the `\n` in the following example as a new line:

   ```
   => SELECT 'a\nb';
      WARNING:  nonstandard use of escape in a string literal at character
   8
      HINT:  Use the escape string syntax for escapes, e.g., E'\r\n'.
    ?column?
   ----------
    a
   b
   (1 row)
   ```

   When StandardConformingStrings is ON, the string is interpreted as four characters: `a \ n b`.

   Modify each string that HP Vertica flags by extending it as in the following example:

   ```
   E'a\nb'
   ```

   Or if the string has quoted single quotes, double them; for example, `'one'' double'`.

3. Turn on the StandardConformingStrings parameter for all sessions:

   ```
   SELECT SET_CONFIG_PARAMETER ('StandardConformingStrings' ,'1');
   ```

Doubled Single Quotes

This section discusses vsql inputs that are not passed on to the server.

HP Vertica recognizes two consecutive single quotes within a string literal as one single quote character. For example, the following inputs, `'You''re here!'` ignored the second consecutive quote and returns the following:

```
=> SELECT 'You''re here!';
   ?column?
--------------
 You're here!
(1 row)
```

This is the SQL standard representation and is preferred over the form, `'You\'re here!'`, because backslashes are not parsed as before. You need to escape the backslash:

```
=> SELECT (E'You\'re here!');
   ?column?
```

```
--------------
 You're here!
(1 row)
```

This behavior change introduces a potential incompatibility in the use of the vsql \set command, which automatically concatenates its arguments. For example, the following works in both HP Vertica 3.5 and 4.0:

```
\set file  '\''  `pwd`  '/file.txt'  '\''
\echo :file
```

vsql takes the four arguments and outputs the following:

```
'/home/vertica/file.txt'
```

In HP Vertica 3.5 the above `\set file` command could be written all with the arguments run together, but in 4.0 the adjacent single quotes are now parsed differently:

```
\set file  '\''`pwd`'/file.txt''\''
\echo :file
'/home/vertica/file.txt''
```

Note the extra single quote at the end. This is due to the pair of adjacent single quotes together with the backslash-quoted single quote.

The extra quote can be resolved either as in the first example above, or by combining the literals as follows:

```
\set file '\''`pwd`'/file.txt'''
\echo :file
'/home/vertica/file.txt'
```

In either case the backslash-quoted single quotes should be changed to doubled single quotes as follows:

```
\set file '''' `pwd` '/file.txt'''
```

Additional Examples

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

=> SELECT E'This \is a string';
     ?column?
------------------
 This is a string

=> SELECT E'This is a \n new line';
       ?column?
----------------------
 This is a
 new line
(1 row)
```

```
=> SELECT 'String''s characters';
      ?column?
--------------------
 String's characters
(1 row)
```

Date/Time Literals

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.

HP 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 55).

Time Zone Values

HP 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. Obvious problems are:

- Although the `DATE` (page 80) type does not have an associated time zone, the `TIME` (page 95) type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
- HP 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, HP recommends using Date/Time types that contain both date and time when you use time zones. HP recommends that you do *not* use the type `TIME WITH TIME ZONE`, even though it is supported it 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.

HP 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 in which they occur.

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	MAY
JUNE	JUN
JULY	JUL
AUGUST	AUG
SEPTEMBER	SEP, SEPT
OCTOBER	OCT

NOVEMBER	NOV
DECEMBER	DEC

Interval Values

An interval value represents the duration between two points in time.

Syntax

```
[ @ ] quantity unit [ quantity unit... ] [ AGO ]
```

Parameters

@	(at sign) is optional and ignored
quantity	Is an integer ***numeric constant*** (page 24)
unit	Is one of the following units or abbreviations or plurals of the following units: MILLISECOND DAY DECADE SECOND WEEK CENTURY MINUTE MONTH MILLENNIUM HOUR YEAR
AGO	[Optional] specifies a negative interval value (an interval going back in time). 'AGO' is a synonym for '-'.

The amounts of different units are implicitly added up with appropriate sign accounting.

Notes

- 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'`
- The boundaries of an interval constant are:
  - '9223372036854775807 usec' to '9223372036854775807 usec ago'
  - 296533 years 3 mons 21 days 04:00:54.775807 to -296533 years -3 mons -21 days -04:00:54.775807
- The range of an interval constant is +/- 2^{63} - 1 (plus or minus two to the sixty-third minus one) microseconds.
- In HP Vertica, the interval fields are additive and accept large floating-point numbers.

Examples

```
SELECT INTERVAL '1 12:59:10';
  ?column?
------------
 1 12:59:10
(1 row)
SELECT INTERVAL '9223372036854775807 usec';
```

```
         ?column?
--------------------------
 106751991 04:00:54.775807
(1 row)
SELECT INTERVAL '-9223372036854775807 usec';
          ?column?
---------------------------
 -106751991 04:00:54.775807
(1 row)
SELECT INTERVAL '-1 day 48.5 hours';
 ?column?
----------
 -3 00:30
(1 row)
SELECT TIMESTAMP 'Apr 1, 07' - TIMESTAMP 'Mar 1, 07';
 ?column?
----------
 31
(1 row)
SELECT TIMESTAMP 'Mar 1, 07' - TIMESTAMP 'Feb 1, 07';
 ?column?
----------
 28
(1 row)
SELECT TIMESTAMP 'Feb 1, 07' + INTERVAL '29 days';
      ?column?
--------------------
  03/02/2007 00:00:00
(1 row)
SELECT TIMESTAMP WITHOUT TIME ZONE '1999-10-01 00:00:01' + INTERVAL '1 month
- 1 second'
AS "Oct 31";
      Oct 31
--------------------
 1999-10-31 00:00:00
(1 row)
```

interval-literal

The following table lists the units allowed for the required `interval-literal` parameter.

Unit	Description
a	Julian year, 365.25 days exactly
ago	Indicates negative time offset
c, cent, century	Century
centuries	Centuries
d, day	Day
days	Days

dec, decade	Decade
decades, decs	Decades
h, hour, hr	Hour
hours, hrs	Hours
ka	Julian kilo-year, 365250 days exactly
m	Minute or month for year/month, depending on context. See Notes below this table.
microsecond	Microsecond
microseconds	Microseconds
mil, millennium	Millennium
millennia, mils	Millennia
millisecond	Millisecond
milliseconds	Milliseconds
min, minute, mm	Minute
mins, minutes	Minutes
mon, month	Month
mons, months	Months
ms, msec, millisecond	Millisecond
mseconds, msecs	Milliseconds
q, qtr, quarter	Quarter
qtrs, quarters	Quarters
s, sec, second	Second
seconds, secs	Seconds
us, usec	Microsecond
microseconds, useconds, usecs	Microseconds
w, week	Week
weeks	Weeks
y, year, yr	Year
years, yrs	Years

Processing the input unit 'm'

The input unit `'m'` can represent either 'months' or 'minutes,' depending on the context. For instance, the following command creates a one-column table with an interval value:

```
=> CREATE TABLE int_test(i INTERVAL YEAR TO MONTH);
```

In the first `INSERT` statement, the values are inserted as 1 year, six months:

```
=> INSERT INTO int_test VALUES('1 year 6 months');
```

The second INSERT statement results in an error from specifying minutes for a YEAR TO MONTH interval. At runtime, the result will be a NULL:

```
=> INSERT INTO int_test VALUES('1 year 6 minutes');
ERROR: invalid input syntax for type interval year to month: "1 year 6 minutes"
```

In the third INSERT statement, the 'm' is processed as months (not minutes), because DAY TO SECOND is truncated:

```
=> INSERT INTO int_test VALUES('1 year 6 m'); -- the m counts as months
```

The table now contains two identical values, with no minutes:

```
=> SELECT * FROM int_test;
  i
-----
 1 year 6 months
 1 year 6 months
(2 rows)
```

In the following command, the 'm' counts as minutes, because the DAY TO SECOND interval-qualifier extracts day/time values from the input:

```
=> SELECT INTERVAL '1y6m' DAY TO SECOND;
 ?column?
-----------
 365 days 6 mins
(1 row)
```

interval-qualifier

The following table lists the optional interval qualifiers. Values in INTERVAL fields, other than SECOND, are integers with a default precision of 2 when they are not the first field.

You cannot combine day/time and year/month qualifiers. For example, the following intervals are not allowed:

- DAY TO YEAR
- HOUR TO MONTH

Interval Type	Units	Valid interval-literal entries
Day/time intervals	DAY	Unconstrained.
	DAY TO HOUR	An interval that represents a span of days and hours.
	DAY TO MINUTE	An interval that represents a span of days and minutes.
	DAY TO SECOND	(Default) interval that represents a span of days, hours, minutes, seconds, and fractions of a second if

		subtype unspecified.
	HOUR	Hours within days.
	HOUR TO MINUTE	An interval that represents a span of hours and minutes.
	HOUR TO SECOND	An interval that represents a span of hours and seconds.
	MINUTE	Minutes within hours.
	MINUTE TO SECOND	An interval that represents a span of minutes and seconds.
	SECOND	Seconds within minutes. **Note:** The SECOND field can have an interval fractional seconds precision, which indicates the number of decimal digits maintained following the decimal point in the SECONDS value. When SECOND is not the first field, it has a precision of 2 places before the decimal point.
Year/month intervals	MONTH	Months within year.
	YEAR	Unconstrained.
	YEAR TO MONTH	An interval that represents a span of years and months.

Operators

Operators are logical, mathematical, and equality symbols used in SQL to evaluate, compare, or calculate values.

Binary Operators

Each of the functions in the following table works with binary and varbinary data types.

Operator	Function	Description
'='	binary_eq	Equal to
'<>'	binary_ne	Not equal to
'<'	binary_lt	Less than
'<='	binary_le	Less than or equal to
'>'	binary_gt	Greater than

'>='	binary_ge	Greater than or equal to
'&'	binary_and	And
'~'	binary_not	Not
'|'	binary_or	Or
'#'	binary_xor	Either or
'||'	binary_cat	Concatenate

Notes

If the arguments vary in length binary operators treat the values as though they are all equal in length by right-extending the smaller values with the zero byte to the full width of the column (except when using the `binary_cat` function). For example, given the values `'ff'` and `'f'`, the value `'f'` is treated as `'f0'`.

Operators are strict with respect to nulls. The result is null if any argument is null. For example, `null <> 'a'::binary` returns null.

To apply the OR (`'|'`) operator to a varbinary type, explicitly cast the arguments; for example:

```
=> SELECT '1'::VARBINARY | '2'::VARBINARY;
 ?column?
----------
 3
(1 row)
```

Similarly, to apply the ***LENGTH*** (page 381), ***REPEAT*** (page 392), ***TO_HEX*** (page 260), and ***SUBSTRING*** (page 403) functions to a binary type, explicitly cast the argument; for example:

```
=> SELECT LENGTH('\\001\\002\\003\\004'::varbinary(4));
 LENGTH
--------
      4
(1 row)
```

When applying an operator or function to a column, the operator's or function's argument type is derived from the column type.

Examples

In the following example, the zero byte is not removed from column cat1 when values are concatenated:

```
=> SELECT 'ab'::BINARY(3) || 'cd'::BINARY(2) AS cat1, 'ab'::VARBINARY(3) ||
   'cd'::VARBINARY(2) AS cat2;
    cat1    | cat2
----------+------
 ab\000cd | abcd
(1 row)
```

When the binary value `'ab'::binary(3)` is translated to varbinary, the result is equivalent to `'ab\\000'::varbinary(3)`; for example:

```
=> SELECT 'ab'::binary(3);
```

```
 binary
--------
 ab\000
(1 row)
```

The following example performs a bitwise AND operation on the two input values (see also ***BIT_AND*** (page 119)):

```
=> SELECT '10001' & '011' as AND;
 AND
-----
   1
(1 row)
```

The following example performs a bitwise OR operation on the two input values (see also ***BIT_OR*** (page 120)):

```
=> SELECT '10001' | '011' as OR;
  OR
-------
 10011
(1 row)
```

The following example concatenates the two input values:

```
=> SELECT '10001' || '011' as CAT;
   CAT
----------
 10001011
(1 row)
```

Boolean Operators

Syntax

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

Parameters

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 52) construct.
- Do not confuse Boolean operators with the ***Boolean-predicate*** (page 60) or the ***Boolean*** (page 76) 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 Parameters

<	less than
>	greater than

<=	less than or equal to
>=	greater than or equal to
= or <=>	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.
- The <=> operator performs an equality comparison like the = operator, but it returns true, instead of NULL, if both operands are NULL, and false, instead of NULL, if one operand is NULL.

Data Type Coercion Operators (CAST)

Data 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'
```

Parameters

expression	Is an expression of any type
data-type	Converts the value of *expression* to one of the following data types: ***BINARY*** (page 72) ***BOOLEAN*** (page 76) ***CHARACTER*** (page 76) ***DATE/TIME*** (page 78) ***NUMERIC*** (page 103) ***DOUBLE PRECISION (FLOAT)*** (page 105)

Notes

- In HP Vertica, data type coercion (casting) can be invoked only by an explicit cast request. It must use, for example, one of the following constructs:

  ```
  CAST(x AS data-type-name)
  ```

  or

  ```
  x::data-type-name
  ```

- Type coercion format of *data-type* '*string*' can be used only 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.
- If a binary value is cast (implicitly or explicitly) to a binary type with a smaller length, the value is silently truncated. For example:

```
=> SELECT 'abcd'::BINARY(2);
 binary
--------
 ab
(1 row)
```

- No casts other than BINARY to and from VARBINARY and resize operations are currently supported.
- On binary data that contains a value with fewer bytes than the target column, values are right-extended with the zero byte `'\0'` to the full width of the column. Trailing zeros on variable length binary values are not right-extended:

```
=> SELECT 'ab'::BINARY(4), 'ab'::VARBINARY(4);
   binary   | varbinary
------------+-----------
 ab\000\000 | ab
(1 row)
```

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)
```

See Also

Data Type Coercion (page 112)

Date/Time Operators

Syntax

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

Parameters

```
+  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
`!`	Factorial	`5 !`	`120`
`+`	Addition	`2 + 3`	`5`
`-`	Subtraction	`2 - 3`	`-1`
`*`	Multiplication	`2 * 3`	`6`

/	Division (integer division produces NUMERIC results).	4 / 2	2.00...
//	With integer division, returns an INTEGER rather than a NUMERIC.	117.32 // 2.5	46
%	Modulo (remainder)	5 % 4	1
^	Exponentiation	2.0 ^ 3.0	8
|/	Square root	|/ 25.0	5
||/	Cube root	||/ 27.0	3
!!	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.
- HP Vertica supports the use of the factorial operators on positive and negative floating point (DOUBLE PRECISION (page 105)) numbers as well as integers. For example:

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

- Factorial is defined in term of the gamma function, where (-1) = Infinity and the other negative integers are undefined. For example

  (-4)! = NaN

  -4! = -(4!) = -24.
- 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.
- See MOD() (page 310) for details about the behavior of %.

NULL Operators

To check whether a value is or is not NULL, use the constructs:

expression IS NULL

expression IS NOT NULL

Alternatively, use equivalent, but nonstandard, constructs:

expression ISNULL
expression NOTNULL

Do not write *expression* = NULL because NULL represents an unknown value, and two unknown values are not necessarily equal. This behavior conforms to the SQL standard.

Note: Some applications might expect that *expression* = NULL returns true if *expression* evaluates to null. HP Vertica strongly recommends that these applications be modified to comply with the SQL standard.

String Concatenation Operators

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

Syntax

string || *string*

Parameters

string	Is an expression of type CHAR or VARCHAR

Notes

- || is used to concatenate expressions and constants. The expressions are cast to VARCHAR if possible, otherwise to VARBINARY, and must both be one or the other.
- Two consecutive strings within a single SQL statement on separate lines are automatically concatenated

Examples

The following example is a single string written on two lines:

```
=> SELECT E'xx'
-> '\\';
 ?column?
----------
 xx\
(1 row)
```

This example shows two strings concatenated:

```
=> SELECT E'xx' ||
-> '\\';
 ?column?
----------
 xx\\
(1 row)

=> SELECT 'auto' || 'mobile';
```

```
 ?column?
----------
 automobile
(1 row)
=> SELECT 'auto'
-> 'mobile';
 ?column?
----------
 automobile
(1 row)
=> SELECT 1 || 2;
 ?column?
----------
 12
(1 row)
=> SELECT '1' || '2';
 ?column?
----------
 12
(1 row)
=> SELECT '1'
-> '2';
 ?column?
----------
 12
(1 row)
```

Expressions

SQL expressions are the components of a query that compare a value or values against other values. They can also perform calculations. Expressions found inside any SQL command are usually in the form of a conditional statement.

Operator Precedence

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

Note: When an expression includes more than one operator, HP 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 52) 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 use it only when necessary. (In this particular example, it would be best to avoid the issue by writing `y > 1.5*x` instead.)

Aggregate Expressions

An aggregate expression represents the application of an ***aggregate function*** (page 118) across the rows or groups of rows selected by a query.

Using `AVG()` as an example, the syntax of an aggregate expression is one of the following:

- Invokes the aggregate across all input rows for which the given expression yields a non-null value:
  `AVG (`*`expression`*`)`

- Is the same as `AVG(`*expression*`)`, because ALL is the default:
  `AVG (ALL` *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.
  `AVG (DISTINCT` *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
```

Parameters

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
```

Parameters

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

```
[ [ [db-name.]schema. ] tablename. ] columnname
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
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.

If you do not specify a *schema*, HP Vertica searches the existing schemas according to the order defined in the `SET SEARCH_PATH` (page 912) command.

Example

This example uses the schema from the VMart Example Database.

In the following command, `transaction_type` and `transaction_time` are the unique column references, `store` is the name of the schema, and `store_sales_fact` is the table name:

```
=> SELECT transaction_type, transaction_time
   FROM store.store_sales_fact
   ORDER BY transaction_time;
 transaction_type | transaction_time
------------------+------------------
 purchase         | 00:00:23
 purchase         | 00:00:32
 purchase         | 00:00:54
 purchase         | 00:00:54
 purchase         | 00:01:15
 purchase         | 00:01:30
 purchase         | 00:01:50
 return           | 00:03:34
```

```
 return           | 00:03:35
 purchase         | 00:03:39
 purchase         | 00:05:13
 purchase         | 00:05:20
 purchase         | 00:05:23
 purchase         | 00:05:27
 purchase         | 00:05:30
 purchase         | 00:05:35
 purchase         | 00:05:35
 purchase         | 00:05:42
 return           | 00:06:36
 purchase         | 00:06:39
(20 rows)
```

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 where the comment begins with `/*` and extends to the matching occurrence of `*/`.

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

These block comments nest, as specified in the SQL standard. Unlike C, you can comment out larger blocks of code that might contain existing block comments.

Date/Time Expressions

HP 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 might 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 might have a text month.

- If the token is numeric only, then it is either a single field or an ISO 8601 concatenated date (for example, 19990113 for January 13, 1999) or time (for example, 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 (for example, today), day (for example, Thursday), month (for example, January), or noise word (for example, 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" (for example, 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 our implementation, 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 (for example, 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.

Special Date/Time Values

HP 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 **Note:** NOW is not the same as the ***NOW*** (see "***NOW [Date/Time]***" on page 235) 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 200)
- ***CURRENT_TIME*** (page 200)
- ***CURRENT_TIMESTAMP*** (page 201)
- ***LOCALTIME*** (page 226)
- ***LOCALTIMESTAMP*** (page 226)

The latter four accept an optional precision specification. (See Date/Time Functions.) 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 60) for determining whether an expression value is NULL.

Notes

- HP Vertica stores data in projections, which are sorted in a specific way. All columns are stored in ASC (ascending) order. For columns of data type NUMERIC, INTEGER, DATE, TIME, TIMESTAMP, and INTERVAL, NULL values are placed at the beginning of sorted projections (NULLS FIRST), while for columns of data type FLOAT, STRING, and BOOLEAN, NULL values are placed at the end (NULLS LAST). For details, see Analytics Null Placement and Minimizing Sort Operations in the Programmer's Guide.
- HP Vertica also accepts NUL characters ('\0') in constant strings and no longer removes null characters from VARCHAR fields on input or output. NUL is the ASCII abbreviation for the NULL character.

- You can write queries with expressions that contain the <=> operator for NULL=NULL joins. See Equi-joins and Non Equi-Joins in the Programmer's Guide.

See Also

NULL-handling Functions (page 321)

Numeric Expressions

HP Vertica follows the IEEE specification for floating point, including NaN.

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.

Examples

```
=> SELECT CBRT('Nan'); -- cube root
 cbrt
------
  NaN
(1 row)
=> SELECT 'Nan' > 1.0;
 ?column?
----------
 f
(1 row)
```

Predicates

Predicates are truth-tests. If the predicate test is true, it returns a value. Each predicate is evaluated per row, so that when the predicate is part of an entire table SELECT statement, the statement can return multiple results.

Predicates consist of a set of parameters and arguments. For example, in the following example WHERE clause:

```
WHERE name = 'Smith';
```

- `name = 'Smith'` is the predicate
- `'Smith'` is an expression

BETWEEN-predicate

The special BETWEEN predicate is available as a convenience.

Syntax

a BETWEEN *x* AND *y*

Notes

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

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
```

Notes

- A null input is treated as the value `UNKNOWN`.
- `IS UNKNOWN` and `IS NOT UNKNOWN` are effectively the same as the ***NULL-predicate*** (page 69), 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.
- Do not confuse the boolean-predicate with ***Boolean Operators*** (on page 44) or the ***Boolean*** (page 76) data type, which can have only two values: true and false.

column-value-predicate

Syntax

```
column-name comparison-op constant-expression
```

Parameters

column-name	Is a single column of one the tables specified in the `FROM` ***clause*** (page 876).
comparison-op	Is one of the ***comparison operators*** (on page 44).
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 69).

Examples

```
table.column1 = 2
table.column2 = 'Seafood'
table.column3 IS NULL
```

IN-predicate

Syntax

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

Parameters

column-expression	One or more columns from the tables specified in the ***FROM clause*** (page 876).
list-expression	A comma-separated list of constant values matching the data type of the *column-expression*

Examples

```
x, y IN ((1,2), (3, 4)), OR x, y IN (SELECT a, b FROM table)
x IN (5, 6, 7)
```

INTERPOLATE

Used to join two event series together using some ordered attribute, event series joins let you compare values from two series directly, rather than having to normalize the series to the same measurement interval.

Syntax

expression1 INTERPOLATE PREVIOUS VALUE *expression2*

Parameters

expression1 *expression2*	Is the ***column-reference*** *(see "**Column References**" on page 54)* from one the tables specified in the ***FROM clause*** (page 876). The column-reference can be any data type, but DATE/TIME types are the most useful, especially TIMESTAMP,since you are joining data that represents an event series.
PREVIOUS VALUE	Pads the non-preserved side with the previous values from relation when there is no match. Input rows are sorted in ascending logical order of the join column. **Note:** An ORDER BY clause, if used, does not determine the input order but only determines query output order.

Notes

- An event series join is an extension of a regular outer join. Instead of padding the non-preserved side with null values when there is no match, the event series join pads the non-preserved side with the previous values from the table.

- The difference between expressing a regular outer join and an event series join is the INTERPOLATE predicate, which is used in the ON clause. See the **Examples** section below Notes and Restrictions. See also Event Series Joins in the Programmer's Guide.
- Data is logically partitioned on the table in which it resides, based on other ON clause equality predicates.
- Interpolated values come from the table that contains the null, not from the other table.
- HP Vertica does not guarantee that there will be no null values in the output. If there is no previous value for a mismatched row, that row will be padded with nulls.
- Event series join requires that both tables be sorted on columns in equality predicates, in any order, followed by the INTERPOLATED column. If data is already sorted in this order, then an explicit sort is avoided, which can improve query performance. For example, given the following tables:

  ```
  ask: exchange, stock, ts, price
  bid: exchange, stock, ts, price
  ```

  In the query that follows

  - `ask` is sorted on `exchange, stock` (or the reverse), `ts`
  - `bid` is sorted on `exchange, stock` (or the reverse), `ts`

  ```
  SELECT ask.price - bid.price, ask.ts, ask.stock, ask.exchange
     FROM ask FULL OUTER JOIN bid
     ON ask.stock = bid.stock AND ask.exchange = bid.exchange
     AND ask.ts INTERPOLATE PREVIOUS VALUE bid.ts;
  ```

Restrictions

- Only one INTERPOLATE expression is allowed per join.
- INTERPOLATE expressions are used only with ANSI SQL-99 syntax (the ON clause), which is already true for full outer joins.
- INTERPOLATE can be used with equality predicates only.
- The AND operator is supported but not the OR and NOT operators.
- Expressions and implicit or explicit casts are not supported, but subqueries are allowed.

Example

The examples that follow use this simple schema.

```
CREATE TABLE t(x TIME);
CREATE TABLE t1(y TIME);
INSERT INTO t VALUES('12:40:23');
INSERT INTO t VALUES('14:40:25');
INSERT INTO t VALUES('14:45:00');
INSERT INTO t VALUES('14:49:55');
INSERT INTO t1 VALUES('12:40:23');
INSERT INTO t1 VALUES('14:00:00');
COMMIT;
```

Normal full outer join

```
=> SELECT * FROM t FULL OUTER JOIN t1 ON t.x = t1.y;
```

Notice the null rows from the non-preserved table:

```
    x     |    y
----------+----------
 12:40:23 | 12:40:23
 14:40:25 |
 14:45:00 |
 14:49:55 |
          | 14:00:00
(5 rows)
```

Full outer join with interpolation

```
=> SELECT * FROM t FULL OUTER JOIN t1 ON t.x INTERPOLATE PREVIOUS VALUE t1.y;
```

In this case, the rows with no entry point are padded with values from the previous row.

```
    x     |    y
----------+----------
 12:40:23 | 12:40:23
 12:40:23 | 14:00:00
 14:40:25 | 14:00:00
 14:45:00 | 14:00:00
 14:49:55 | 14:00:00
(5 rows)
```

Normal Left Outer Join

```
=> SELECT * FROM t LEFT OUTER JOIN t1 ON t.x = t1.y;
```

Again, there are nulls in the non-preserved table

```
    x     |    y
----------+----------
 12:40:23 | 12:40:23
 14:40:25 |
 14:45:00 |
 14:49:55 |
(4 rows)
```

Left Outer Join with Interpolation

```
=> SELECT * FROM t LEFT OUTER JOIN t1 ON t.x INTERPOLATE PREVIOUS VALUE t1.y;
```

Nulls padded with interpolated values.

```
    x     |    y
----------+----------
 12:40:23 | 12:40:23
 14:40:25 | 14:00:00
 14:45:00 | 14:00:00
 14:49:55 | 14:00:00
(4 rows)
```

Inner joins

For inner joins, there is no difference between a regular inner join and an event series inner join. Since null values are eliminated from the result set, there is nothing to interpolate.

A regular inner join returns only the single matching row at 12:40:23:

```
=> SELECT * FROM t INNER JOIN t1 ON t.x = t1.y;
```

```
    x      |    y
----------+----------
 12:40:23 | 12:40:23
(1 row)
```

An event series inner join finds the same single-matching row at 12:40:23:

```
=> SELECT * FROM t INNER JOIN t1 ON t.x INTERPOLATE PREVIOUS VALUE t1.y;
    x      |    y
----------+----------
 12:40:23 | 12:40:23
(1 row)
```

Semantics

When you write an event series join in place of normal join, values are evaluated as follows (using the schema in the above examples):

- `t` is the outer, preserved table
- `t1` is the inner, non-preserved table
- For each row in outer table `t`, the ON clause predicates are evaluated for each combination of each row in the inner table `t1`.
- If the ON clause predicates evaluate to true for any combination of rows, those combination rows are produced at the output.
- If the ON clause is false for all combinations, a single output row is produced with the values of the row from `t` along with the columns of `t1` chosen from the row in `t1` with the greatest `t1.y` value such that `t1.y < t.x`; If no such row is found, pad with nulls.

Note: `t LEFT OUTER JOIN t1` is equivalent to `t1 RIGHT OUTER JOIN t`.

In the case of a full outer join, all values from both tables are preserved.

See Also

Event Series Joins in the Programmer's Guide

join-predicate

Combines records from two or more tables in a database.

Syntax

column-reference (see "*Column References*" on page 54) = *column-reference*

Parameters

column-reference	Refers to a column of one the tables specified in the ***FROM clause*** (page 876).

LIKE-predicate

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

Syntax

```
string [ NOT ]{ LIKE | ILIKE | LIKEB | ILIKEB }
... pattern [ESCAPE 'escape-character' ]
```

Parameters

string	(CHAR, VARCHAR, BINARY, VARBINARY) is the column value to be compared to the *pattern*.
NOT	Returns true if LIKE returns false, and the reverse; 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*. An ESCAPE character can be used to escape itself, underscore (_), and % only. This is enforced only for non-default collations. To match the ESCAPE character itself, use two consecutive escape characters. The default ESCAPE character is the backslash (\) character, although standard SQL specifies no default ESCAPE character. ESCAPE works for char and varchar strings only.
escape-character	Causes character to be treated as a literal, rather than a wildcard, when preceding an underscore or percent sign character in the *pattern*.

Notes

- The LIKE predicate is compliant with the SQL standard.
- In the default locale, LIKE and ILIKE handle UTF-8 character-at-a-time, locale-insensitive comparisons. ILIKE handles language-independent case-folding.

  Note: In non-default locales, LIKE and ILIKE do locale-sensitive string comparisons, including some automatic normalization, using the same algorithm as the "=" operator on VARCHAR types.
- The LIKEB and ILIKEB predicates do byte-at-a-time ASCII comparisons, providing access to HP Vertica 4.0 functionality.
- LIKE and ILIKE are stable for character strings, but immutable for binary strings, while LIKEB and ILIKEB are both immutable

- For `collation=binary` settings, the behavior is similar to HP Vertica 4.0. For other collations, `LIKE` operates on UTF-8 character strings, with the exact behavior dependent on collation parameters, such as strength. In particular, `ILIKE` works by setting S=2 (ignore case) in the current session locale. See Locale Specification in the Administrator's Guide.
- Although the SQL standard specifies no default `ESCAPE` character, in HP Vertica the default is the backslash (`\`) and works for `CHAR` and `VARCHAR` strings only.

  Tip: HP recommends that you specify an explicit escape character in all cases, to avoid problems should this behavior change. To use a backslash character as a literal, either specify a different escape character or use two backslashes.
- `ESCAPE` expressions evaluate to exactly one octet — or one UTF-8 character for non-default locales.
- An `ESCAPE` character can be used only to escape itself, `_`, and `%`. This is enforced only for non-default collations.
- `LIKE` requires that the entire string expression 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 binary data types, you must use a valid binary character as the escape character, since backslash is not a valid `BINARY` character.
- The following symbols are substitutes for the actual keywords:

  ```
  ~~     LIKE
  ~#     LIKEB
  ~~*    ILIKE
  ~#*    ILIKEB
  !~~    NOT LIKE
  !~#    NOT LIKEB
  !~~*   NOT ILIKE
  !~#*   NOT IILIKEB
  ```

  The `ESCAPE` keyword is not valid for the above symbols.
- HP Vertica extends support for single-row subqueries as the pattern argument for LIKEB and ILIKEB; for example:

  ```
  SELECT * FROM t1 WHERE t1.x LIKEB (SELECT MAX (t2.a) FROM t2);
  ```

Querying Case-sensitive data in System Tables

The `V_CATALOG.TABLES` (page 978)`.TABLE_SCHEMA` and `TABLE_NAME` columns are case sensitive when used with an equality (=) predicate in queries. For example, given the following sample schema, if you execute a query using the = predicate, HP Vertica returns 0 rows:

```
=> CREATE SCHEMA SS;
=> CREATE TABLE SS.TT (c1 int);
=> INSERT INTO ss.tt VALUES (1);
=> SELECT table_schema, table_name FROM v_catalog.tables WHERE table_schema ='ss';

 table_schema | table_name
--------------+------------
```

```
(0 rows)
```

TIP: Use the case-insensitive **ILIKE** predicate to return the expected results.

```
=> SELECT table_schema, table_name FROM v_catalog.tables WHERE table_schema ILIKE
'ss';
 table_schema | table_name
--------------+------------
 SS           | TT
(1 row)
```

Examples

```
'abc' LIKE 'abc'  true
'abc' LIKE 'a%'   true
'abc' LIKE '_b_'  true
'abc' LIKE 'c'    false
'abc' LIKE 'ABC' false
'abc' ILIKE 'ABC' true
'abc' not like 'abc' false
not 'abc' like 'abc' false
```

The following example illustrates pattern matching in locales.

```
\locale default
=> CREATE TABLE src(c1 VARCHAR(100));
=> INSERT INTO src VALUES (U&'\00DF'); --The sharp s (ß)
=> INSERT INTO src VALUES ('ss');
=> COMMIT;
```

Querying the `src` table in the default locale returns both ss and sharp s.

```
=> SELECT * FROM src;
 c1
----
 ß
 ss
(2 rows)
```

The following query combines pattern-matching predicates to return the results from column `c1`:

```
=> SELECT c1, c1 = 'ss' AS equality, c1 LIKE 'ss' AS LIKE, c1
   ILIKE 'ss' AS ILIKE FROM src;

 c1 | equality | LIKE | ILIKE
----+----------+------+-------
 ß  | f        | f    | f
 ss | t        | t    | t
(2 rows)
```

The next query specifies unicode format for `c1`:

```
=> SELECT c1, c1 = U&'\00DF' AS equality, c1 LIKE U&'\00DF' AS LIKE,
   c1 ILIKE U&'\00DF' AS ILIKE from src;

 c1 | equality | LIKE | ILIKE
----+----------+------+-------
 ß  | t        | t    | t
```

```
 ss | f        | f    | f
(2 rows)
```

Now change the locale to German with a strength of 1 (ignore case and accents):

```
\locale LDE_S1
=> SELECT c1, c1 = 'ss' AS equality, c1 LIKE 'ss' as LIKE,
   c1 ILIKE 'ss' AS ILIKE from src;

 c1 | equality | LIKE | ILIKE
----+----------+------+-------
 ß  | t        | t    | t
 ss | t        | t    | t
(2 rows)
```

This example illustrates binary data types with pattern-matching predicates:

```
=> CREATE TABLE t (c BINARY(1));
=> INSERT INTO t values(HEX_TO_BINARY('0x00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFF'));
=> SELECT TO_HEX(c) from t;
 TO_HEX
--------
 00
 ff
(2 rows)
select * from t;
  c
------
 \000
 \377
(2 rows)

=> SELECT c, c = '\000', c LIKE '\000', c ILIKE '\000' from t;

  c   | ?column? | ?column? | ?column?
------+----------+----------+----------
 \000 | t        | t        | t
 \377 | f        | f        | f
(2 rows)

=> SELECT c, c = '\377', c LIKE '\377', c ILIKE '\377' from t;
  c   | ?column? | ?column? | ?column?
------+----------+----------+----------
 \000 | f        | f        | f
 \377 | t        | t        | t
(2 rows)
```

NULL-predicate

Tests for null values.

Syntax

value_expression IS [NOT] NULL

Parameters

value_expression	A column name, literal, or function.

Examples

Column name:

```
=> SELECT date_key FROM date_dimension WHERE date_key IS NOT NULL;
 date_key
----------
        1
      366
     1462
     1097
        2
        3
        6
        7
        8
...
```

Function:

```
=> SELECT MAX(household_id) IS NULL FROM customer_dimension;
 ?column?
----------
 f
(1 row)
```

Literal:

```
=> SELECT 'a' IS NOT NULL;
 ?column?
----------
 t
(1 row)
```

See Also

NULL Value (page 57)

SQL Data Types

The following tables summarize the data types that HP Vertica supports. It also shows the default placement of null values in projections. The Size column is listed as uncompressed bytes.

Type	Size	Description	NULL Sorting
Binary types			
BINARY	1 to 65000	Fixed-length binary string	NULLS LAST
VARBINARY	1 to 65000	Variable-length binary string	NULLS LAST
BYTEA	1 to 65000	Variable-length binary string (synonym for VARBINARY)	NULLS LAST
RAW	1 to 65000	Variable-length binary string (synonym for VARBINARY)	NULLS LAST
Boolean types			
BOOLEAN	1	True or False or NULL	NULLS LAST
Character types			
CHAR	1 to 65000	Fixed-length character string	NULLS LAST
VARCHAR	1 to 65000	Variable-length character string	NULLS LAST
Date/time types			
DATE	8	Represents a month, day, and year	NULLS FIRST
DATETIME	8	Represents a date and time with or without timezone (synonym for TIMESTAMP)	NULLS FIRST
SMALLDATETIME	8	Represents a date and time with or without timezone (synonym for TIMESTAMP)	NULLS FIRST
TIME	8	Represents a time of day without timezone	NULLS FIRST
TIME WITH TIMEZONE	8	Represents a time of day with timezone	NULLS FIRST
TIMESTAMP	8	Represents a date and time without timezone	NULLS FIRST
TIMESTAMP WITH TIMEZONE	8	Represents a date and time with timezone	NULLS FIRST
INTERVAL	8	Measures the difference between two points in time	NULLS FIRST
Approximate numeric types			

DOUBLE PRECISION	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
FLOAT	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
FLOAT(n)	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
FLOAT8	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
REAL	8	Signed 64-bit IEEE floating point number, requiring 8 bytes of storage	NULLS LAST
Exact numeric types			
INTEGER	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
INT	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
BIGINT	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
INT8	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
SMALLINT	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
TINYINT	8	Signed 64-bit integer, requiring 8 bytes of storage	NULLS FIRST
DECIMAL	8+	8 bytes for the first 18 digits of precision, plus 8 bytes for each additional 19 digits	NULLS FIRST
NUMERIC	8+	8 bytes for the first 18 digits of precision, plus 8 bytes for each additional 19 digits	NULLS FIRST
NUMBER	8+	8 bytes for the first 18 digits of precision, plus 8 bytes for each additional 19 digits	NULLS FIRST
MONEY	8+	8 bytes for the first 18 digits of precision, plus 8 bytes for each additional 19 digits	NULLS FIRST

Binary Data Types

Store raw-byte data, such as IP addresses, up to 65000 bytes.

Syntax

```
BINARY ( length )
{ VARBINARY | BINARY VARYING | BYTEA | RAW } ( max-length )
```

Parameters

length | *max-length*	Specifies the length of the string (column width, declared in bytes (octets), in ***CREATE TABLE*** (page 770) statements).

Notes

- BYTEA and RAW are synonyms for VARBINARY.
- The data types BINARY and BINARY VARYING (VARBINARY) are collectively referred to as *binary string types* and the values of binary string types are referred to as *binary strings*.
- A binary string is a sequence of octets, or bytes. Binary strings store raw-byte data, while character strings store text.
- A binary value value of NULL appears last (largest) in ascending order.
- The binary data types, BINARY and VARBINARY, are similar to the ***character data types*** (page 76), CHAR and VARCHAR, respectively, except that binary data types contain byte strings, rather than character strings.
- **BINARY** — A fixed-width string of *length* bytes, where the number of bytes is declared as an optional specifier to the type. If length is omitted, the default is 1. Where necessary, values are right-extended to the full width of the column with the zero byte. For example:

```
=> SELECT TO_HEX('ab'::BINARY(4));
  to_hex
----------
 6162000
```

- **VARBINARY** — A variable-width string up to a length of *max-length* bytes, where the maximum number of bytes is declared as an optional specifier to the type. The default is the default attribute size, which is 80, and the maximum length is 65000 bytes. Varbinary values are not extended to the full width of the column. For example:

```
=> SELECT TO_HEX('ab'::VARBINARY(4));
 to_hex
--------
 6162
```

- You can use several formats when working with binary values, but the hexadecimal format is generally the most straightforward and is emphasized in HP Vertica documentation.
- Binary operands &, ~, | and # have special behavior for binary data types, as described in ***Binary Operators*** (page 41).
- On input, strings are translated from:
  - hexadecimal representation to a binary value using the HEX_TO_BINARY (page 367) function
  - bitstring representation to a binary value using the BITSTRING_TO_BINARY (page 360) function.

  Both functions take a VARCHAR argument and return a VARBINARY value. See the Examples section below.
- Binary values can also be represented in octal format by prefixing the value with a backslash '\'.

Note: If you use vsql, you must use the escape character (\) when you insert another backslash on input; for example, input `'\141'` as `'\\141'`.

You can also input values represented by printable characters. For example, the hexadecimal value `'0x61'` can also be represented by the symbol `'a'`.

See Loading Different Formats in the Administrator's Guide.

- Like the input format the output format is a hybrid of octal codes and printable ASCII characters. A byte in the range of printable ASCII characters (the range `[0x20, 0x7e]`) is represented by the corresponding ASCII character, with the exception of the backslash (`'\'`), which is escaped as `'\\'`. All other byte values are represented by their corresponding octal values. For example, the bytes {97,92,98,99}, which in ASCII are `{a,\,b,c}`, are translated to text as `'a\\bc'`.
- The following aggregate functions are supported for binary data types:
  - ***BIT_AND*** (page 119)
  - ***BIT_OR*** (page 120)
  - ***BIT_XOR*** (page 122)
  - ***MAX*** (page 128)
  - ***MIN*** (page 129)

  `BIT_AND`, `BIT_OR`, and `BIT_XOR` are bitwise operations that are applied to each non-null value in a group, while `MAX` and `MIN` are bytewise comparisons of binary values.
- Like their ***binary operator*** (page 41) counterparts, if the values in a group vary in length, the aggregate functions treat the values as though they are all equal in length by extending shorter values with zero bytes to the full width of the column. For example, given a group containing the values `'ff', null, and 'f'`, a binary aggregate ignores the null value and treats the value `'f'` as `'f0'`. Also, like their binary operator counterparts, these aggregate functions operate on `VARBINARY` types explicitly and operate on `BINARY` types implicitly through casts. See ***Data Type Coercion Operators (CAST)*** (page 45).

Examples

The following example shows `VARBINARY HEX_TO_BINARY (page 367)(VARCHAR)` and `VARCHAR TO_HEX (page 260)(VARBINARY)` usage.

Table `t` and and its projection are created with binary columns:

```
=> CREATE TABLE t (c BINARY(1));
=> CREATE PROJECTION t_p (c) AS SELECT c FROM t;
```

Insert minimum byte and maximum byte values:

```
=> INSERT INTO t values(HEX_TO_BINARY('0x00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFF'));
```

Binary values can then be formatted in hex on output using the TO_HEX function:

```
=> SELECT TO_HEX(c) FROM t;
 to_hex
 --------
  00
  ff
```

```
 (2 rows)
```

The BIT_AND, BIT_OR, and BIT_XOR functions are interesting when operating on a group of values. For example, create a sample table and projections with binary columns:

This examples uses the following schema, which creates table t with a single column of VARBINARY data type:

```
=> CREATE TABLE t (
      c VARBINARY(2) );
=> INSERT INTO t values(HEX_TO_BINARY('0xFF00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFFFF'));
=> INSERT INTO t values(HEX_TO_BINARY('0xF00F'));
```

Query table t to see column c output:

```
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 ff00
 ffff
 f00f
(3 rows)
```

Now issue the bitwise AND operation. Because these are aggregate functions, an implicit GROUP BY operation is performed on results using (ff00&(ffff)&f00f):

```
=> SELECT TO_HEX(BIT_AND(c)) FROM t;
to_hex
--------
f000
(1 row)
```

Issue the bitwise OR operation on (ff00|(ffff)|f00f):

```
=> SELECT TO_HEX(BIT_OR(c)) FROM t;
to_hex
--------
ffff
(1 row)
```

Issue the bitwise XOR operation on (ff00#(ffff)#f00f):

```
=> SELECT TO_HEX(BIT_XOR(c)) FROM t;
to_hex
--------
f0f0
(1 row)
```

See Also

Aggregate functions ***BIT_AND*** (page 119), ***BIT_OR*** (page 120), ***BIT_XOR*** (page 122), ***MAX*** (page 128), and ***MIN*** (page 129)

Binary Operators (page 41)

COPY (page 699)

Data Type Coercion Operators (CAST) (page 45)

IP conversion function ***INET_ATON*** (page 292), ***INET_NTOA*** (page 293), ***V6_ATON*** (page 294), ***V6_NTOA*** (page 295), ***V6_SUBNETA*** (page 296), ***V6_SUBNETN*** (page 297), ***V6_TYPE*** (page 298)

String functions ***BITCOUNT*** (page 359), ***BITSTRING_TO_BINARY*** (page 360), ***HEX_TO_BINARY*** (page 367), ***LENGTH*** (page 381), ***REPEAT*** (page 392), ***SUBSTRING*** (page 403), ***TO_HEX*** (page 260), and ***TO_BITSTRING*** (page 256)

Loading Binary Data in the Administrator's Guide

Boolean Data Type

HP 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
```

Parameters

Valid literal data values for input are:

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

Notes

- Do not confuse the `BOOLEAN` data type with ***Boolean Operators*** (on page 44) or the ***Boolean-predicate*** (on page 60).
- The keywords `TRUE` and FALSE are preferred and are SQL-compliant.
- A Boolean value of NULL appears last (largest) in ascending order.
- All other values must be enclosed in single quotes.
- Boolean values are output using the letters t and f.

See Also

NULL Value (page 57)

Data Type Coercion Chart (page 115)

Character Data Types

Stores strings of letters, numbers, and symbols.

Character data can be stored as fixed-length or variable-length strings. Fixed-length strings are right-extended with spaces on output; variable-length strings are not extended.

Syntax

```
[ CHARACTER | CHAR ] ( octet_length )
[ VARCHAR | CHARACTER VARYING ] ( octet_length )
```

Parameters

octet_length	Specifies the length of the string (column width, declared in bytes (octets), in ***CREATE TABLE*** (page 770) statements).

Notes

- The data types CHARACTER (CHAR) and CHARACTER VARYING (VARCHAR) are collectively referred to as *character string types,* and the values of character string types are known as *character strings*.
- CHAR is conceptually a fixed-length, blank-padded string. Any trailing blanks (spaces) are removed on input, and only restored on output. The default length is 1, and the maximum length is 65000 octets (bytes).
- VARCHAR is a variable-length character data type. The default length is 80, and the maximum length is 65000 octets. Values can include trailing spaces.
- When you define character columns, specify the maximum size of any string to be stored in a column. For example, to store strings up to 24 octets in length, use either of the following definitions:

  ```
  CHAR(24)     /* fixed-length */
  VARCHAR(24) /* variable-length */
  ```

- The maximum length parameter for VARCHAR and CHAR data type refers to the number of octets that can be stored in that field, not the number of characters (Unicode code points). When using multibyte UTF-8 characters, the fields must be sized to accommodate from 1 to 4 octets per character, depending on the data. If the data loaded into a VARCHAR/CHAR column exceeds the specified maximum size for that column, data is truncated on UTF-8 character boundaries to fit within the specified size. See COPY (page 699).

  Note: Remember to include the extra octets required for multibyte characters in the column-width declaration, keeping in mind the 65000 octet column-width limit.
- String literals in SQL statements must be enclosed in single quotes.
- Due to compression in HP Vertica, the cost of overestimating the length of these fields is incurred primarily at load time and during sorts.
- NULL appears last (largest) in ascending order. See also ***GROUP BY Clause*** (page 878) for additional information about NULL ordering.

The difference between NULL and NUL

NUL represents a character whose ASCII/Unicode code is 0, sometimes qualified "ASCII NUL".

NULL means no value, and is true of a field (column) or constant, not of a character.

CHAR and VARCHAR string data types accept ASCII NULs.

The following example casts the input string containing NUL values to VARCHAR:

```
=> SELECT 'vert\0ica'::CHARACTER VARYING AS VARCHAR;
 VARCHAR
---------
 vert\0ica
(1 row)
```

The result contains 9 characters:

```
=> SELECT LENGTH('vert\0ica'::CHARACTER VARYING);
 length
--------
      9
(1 row)
```

If you use an ***extended string literal*** (page 31), the length is 8 characters:

```
=> SELECT E'vert\0ica'::CHARACTER VARYING AS VARCHAR;
 VARCHAR
---------
 vertica
(1 row)

=> SELECT LENGTH(E'vert\0ica'::CHARACTER VARYING);
 LENGTH
--------
      8
(1 row)
```

See Also

Data Type Coercion (page 112)

Date/Time Data Types

HP Vertica supports the full set of SQL date and time data types. In most cases, a combination of `DATE`, `DATETIME`, `SMALLDATETIME`, `TIME`, `TIMESTAMP WITHOUT TIME ZONE`, and `TIMESTAMP WITH TIME ZONE`, and `INTERVAL` provides a complete range of date/time functionality required by any application.

In compliance with the SQL standard, HP Vertica also supports the `TIME WITH TIME ZONE` data type.

The following table lists the date/time data types, their sizes, values, and resolution.

Name	Size	Description	Low Value	High Value	Resolution
`DATE`	8 bytes	Dates only (no time of day)	~ 25e+15 BC	~ 25e+15 AD	1 day

TIME [(p)]	8 bytes	Time of day only (no date)	00:00:00.00	23:59:60.999999	1 microsecond
TIMETZ [(p)]	8 bytes	Time of day only, with time zone	00:00:00.00+14	23:59:59.999999-14	1 microsecond
TIMESTAMP [(p)]	8 bytes	Both date and time, without time zone	290279-12-22 19:59:05.224194 BC	294277-01-09 04:00:54:775806 AD	1 microsecond
TIMESTAMPTZ [(p)]	8 bytes	Both date and time, with time zone	290279-12-22 19:59:05.224194 BC UTC	294277-01-09 04:00:54:775806 AD UTC	1 microsecond
INTERVAL [(p)] DAY TO SECOND	8 bytes	Time intervals	-106751991 days 04:00:54.775807	+-106751991 days 04:00:54.775807	1 microsecond
INTERVAL [(p)] YEAR TO MONTH	8 bytes	Time intervals	~ -768e15 yrs	~ 768e15 yrs	1 month

Time zone abbreviations for input

HP Vertica recognizes the files in `/opt/vertica/share/timezonesets` as date/time input values and defines the default list of strings accepted in the AT TIME ZONE *zone* parameter. The names are not necessarily used for date/time output — output is driven by the official time zone abbreviations associated with the currently selected time zone parameter setting.

Notes

- In HP Vertica, `TIME ZONE` is a synonym for `TIMEZONE`.
- HP Vertica uses Julian dates for all date/time calculations, which can correctly predict and calculate any date more recent than 4713 BC to far into the future, based on the assumption that the average length of the year is 365.2425 days.
- All date/time types are stored in eight bytes.
- 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';
           ?column?
----------------------------
 2012-03-13 11:42:22.766989
(1 row)
```

- In HP Vertica, the `INTERVAL` (page 81) data type is SQL:2008 compliant and allows modifiers, called ***interval qualifiers*** (page 40), that divide the INTERVAL type into two primary subtypes, `DAY TO SECOND` (the default) and `YEAR TO MONTH`. You use the `SET INTERVALSTYLE` (page 906) command to change the `intervalstyle` run-time parameter for the current session.

  Intervals are represented internally as some number of microseconds and printed as up to 60 seconds, 60 minutes, 24 hours, 30 days, 12 months, and as many years as necessary. Fields can be positive or negative.

See Also

Set the Default Time Zone and Using Time Zones with HP Vertica in the Installation Guide

Sources for Time Zone and Daylight Saving Time Data
http://www.twinsun.com/tz/tz-link.htm

DATE

Consists of a month, day, and year.

Syntax

```
DATE
```

Parameters/limits

Low Value	High Value	Resolution
~ 25e+15 BC	~ 25e+15 AD	1 DAY

See ***SET DATESTYLE*** (page 903) for information about ordering.

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

DATETIME

DATETIME is an alias for ***TIMESTAMP*** (page 97).

INTERVAL

Measures the difference between two points in time. The `INTERVAL` data type is divided into two major subtypes:

- `DAY TO SECOND` (day/time, in microseconds)
- `YEAR TO MONTH` (year/month, in months)

A day/time interval represents a span of days, hours, minutes, seconds, and fractional seconds. A year/month interval represents a span of years and months. Intervals can be positive or negative.

Syntax

`INTERVAL [ (p) ] [ - ] '`***`interval-literal`*** `(on page 38)' [ `***`interval-qualifier`*** `(on page 40) ]`

Parameters

(p)	[Optional] Specifies the precision for the number of digits retained in the seconds field. Enter the precision value in parentheses (). The interval precision can range from 0 to 6. The default is 6.
-	[Optional] Indicates a negative interval.
`'interval-literal'`	Indicates a literal character string expressing a specific interval.
`interval-qualifier`	[Optional] Specifies a range of interval subtypes with optional precision specifications. If omitted, the default is `DAY TO SECOND(6)`. Sometimes referred to as *subtype* in this topic. Within the single quotes of an `interval-literal`, units can be plural, but outside the quotes, the `interval-qualifier` must be singular.

Limits

Name	Low Value	High Value	Resolution
`INTERVAL [(p)] DAY TO SECOND`	–106751991 days 04:00:54.775807	+–106751991 days 04:00:54.775807	1 microsecond
`INTERVAL [(p)] YEAR TO MONTH`	~ –768e15 yrs	~ 768e15 yrs	1 month

Displaying or omitting interval units in output

To display or omit interval units from the output of a SELECT INTERVAL query, use the INTERVALSTYLE (page 906) and DATESTYLE (page 903) settings. These settings affect only the interval output format, not the interval input format.

To omit interval units from the output, set INTERVALSTYLE to PLAIN. This is the default value, and it follows the SQL:2008 standard (ISO):

```
=> SET INTERVALSTYLE TO PLAIN;
SET

=> SELECT INTERVAL '3 2';
?column?
----------
 3 02:00
```

When INTERVALSTYLE is set to PLAIN, units are omitted from the output, even if you specify the units in the query:

```
=> SELECT INTERVAL '3 days 2 hours';
 ?column?
----------
 3 02:00
```

To display interval units in the output, set INTERVALSTYLE to UNITS:

```
=> SET INTERVALSTYLE TO UNITS;
SET

=> SELECT INTERVAL '3 2';
 ?column?
----------------
 3 days 2 hours
```

When INTERVALSTYLE is set to UNITS to display units in the result, the DATESTYLE (page 903) setting controls the format of the units in the output.

If you set DATESTYLE to SQL, interval units are omitted from the output, even if you set INTERVALSTYLE to UNITS:

```
=> SET INTERVALSTYLE TO UNITS;
SET

=> SET DATESTYLE TO SQL;
SET

=> SELECT INTERVAL '3 2';
 ?column?
----------
 3 02:00
```

To display interval units on output, set DATESTYLE to ISO:

```
=> SET INTERVALSTYLE TO UNITS;
SET
```

```
=> SET DATESTYLE TO ISO;
SET

=> SELECT INTERVAL '3 2';
 ?column?
----------------
 3 days 2 hours
```

To check the `INTERVALSTYLE` or `DATESTYLE` setting, use the ***SHOW*** (page 923) command:

```
=> SHOW INTERVALSTYLE;
     name      | setting
---------------+---------
 intervalstyle | units

=> SHOW DATESTYLE;
   name    | setting
-----------+----------
 datestyle | ISO, MDY
```

Specifying units on input

You can specify interval units in the interval-literal:

```
=> SELECT INTERVAL '3 days 2 hours';
 ?column?
--------------
 3 days 2 hours
```

The following command uses the same interval-literal as the previous example, but specifies a `MINUTE` interval-qualifier to so that the results are displayed only in minutes:

```
=> SELECT INTERVAL '3 days 2 hours' MINUTE;
 ?column?
-----------
 4440 mins
```

HP Vertica allows combinations of units in the interval-qualifier, as in the next three examples:

```
=> SELECT INTERVAL '1 second 1 millisecond' DAY TO SECOND;
 ?column?
--------------
 1.001 secs

=> SELECT INTERVAL '28 days 3 hours 65 min' HOUR TO MINUTE;
 ?column?
-----------
 676 hours 5 mins
```

Units less than a month are not valid for `YEAR TO MONTH` interval-qualifiers:

```
=> SELECT INTERVAL '1 Y 30 DAYS' YEAR TO MONTH;
ERROR:  invalid input syntax for type interval year to month: "1 Y 30 DAYS"
```

If you replace `DAYS` in the interval-literal with `M` to represent months, HP Vertica returns the correct information of 1 year, 3 months:

```
=> SELECT INTERVAL '1 Y 3 M' YEAR TO MONTH;
 ?column?
----------
 1 year 3 months
```

in the previous example, `M` was used as the interval-literal, representing months. If you specify a `DAY TO SECOND` interval-qualifier, HP Vertica knows that `M` represents minutes, as in the following example:

```
=> SELECT INTERVAL '1 D 3 M' DAY TO SECOND;
 ?column?
----------
 1 day 3 mins
```

The next two examples use units in the input to return microseconds:

```
=> SELECT INTERVAL '4:5 1 2 34us';
 ?column?
-------------------
 1 day 04:05:02.000034

=> SELECT INTERVAL '4:5 1d 2 34us' HOUR TO SECOND;
 ?column?
-----------------
 28 hours 5 mins 2.000034 secs
```

How the interval-qualifier affects output units

The interval-qualifier specifies a range of interval subtypes to apply to the interval-literal. You can also specify the precision in the interval-qualifier.

If an interval-qualifier is not specified, the default subtype is `DAY TO SECOND(6)`, regardless of what is inside the quotes. For example, as an extension to SQL:2008, both of the following commands return `910 days`:

```
=> SELECT INTERVAL '2-6';
 ?column?
-----------------
 910 days

=> SELECT INTERVAL '2 years 6 months';
 ?column?
-----------------
 910 days
```

However, if you change the interval-qualifier to `YEAR TO MONTH`, you get the following results:

```
=> SELECT INTERVAL '2 years 6 months' YEAR TO MONTH;
 ?column?
-----------------
 2 years 6 months
```

An interval-qualifier can extract other values from the input parameters. For example, the following command extracts the `HOUR` value from the input parameters:

```
=> SELECT INTERVAL '3 days 2 hours' HOUR;
 ?column?
```

```
----------
 74 hours
```

When specifying intervals that use subtype `YEAR TO MONTH`, the returned value is kept as months:

```
=> SELECT INTERVAL '2 years 6 months' YEAR TO MONTH;
 ?column?
-----------
 2 years 6 months
```

The primary day/time (`DAY TO SECOND`) and year/month (`YEAR TO MONTH`) subtype ranges can be restricted to more specific range of types by an interval-qualifier. For example, `HOUR TO MINUTE` is a limited form of day/time interval, which can be used to express time zone offsets.

```
=> SELECT INTERVAL '1 3' HOUR to MINUTE;
 ?column?
---------------
 01:03
```

The formats hh:mm:ss and hh:mm are used only when at least two of the fields specified in the interval-qualifier are non-zero and there are no more than 23 hours or 59 minutes:

```
=> SELECT INTERVAL '2 days 12 hours 15 mins' DAY TO MINUTE;
 ?column?
--------------
 2 days 12:15

=> SELECT INTERVAL '15 mins 20 sec' MINUTE TO SECOND;
 ?column?
----------
 00:15:20

=> SELECT INTERVAL '1 hour 15 mins 20 sec' MINUTE TO SECOND;
 ?column?
-----------------
 75 mins 20 secs
```

Specifying precision

SQL:2008 allows you to specify precision for the interval output by entering the precision value in parentheses after the `INTERVAL` keyword or the interval-qualifier. HP Vertica rounds the input to the number of decimal places specified. `SECOND(2)` and `SECOND (2)` produce the same result:

If you specify two different precisions, HP Vertica picks the lesser of the two:

```
=> SELECT INTERVAL(1) '1.2467' SECOND(2);
 ?column?
----------
 1.2 secs
```

When you specify a precision *inside* an interval-literal, HP Vertica processes the precision by removing the parentheses. In this example, `(3)` is processed as 3 minutes, the first omitted field:

```
=> SELECT INTERVAL '28 days 3 hours 1.234567 second(3)';
```

```
 ?column?
--------------------
 28 days 03:03:01.234567
```

The following command specifies that the day field can hold 4 digits, the hour field 2 digits, the minutes field 2 digits, the seconds field 2 digits, and the fractional seconds field 6 digits:

```
=> SELECT INTERVAL '1000 12:00:01.123456' DAY(4) TO SECOND(6);
 ?column?
--------------------------
 1000 days 12:00:01.123456
```

AN HP Vertica extension lets you specify the seconds precision on the `INTERVAL` keyword. The result is the same:

```
=> SELECT INTERVAL(6) '1000 12:00:01.123456' DAY(4) TO SECOND;
 1000 days 12:00:01.123456
```

Casting with intervals

You can cast a string to an interval:

```
=> SELECT CAST('3700 sec' AS INTERVAL);
 ?column?
----------
 01:01:40
```

You can cast an interval to a string:

```
=> SELECT CAST((SELECT INTERVAL '3700 seconds') AS VARCHAR(20));
 ?column?
----------
 01:01:40
```

You can cast intervals *within* the day/time or the year/month subtypes but not between them. Use CAST to convert interval types:

```
=> SELECT CAST(INTERVAL '4440' MINUTE as INTERVAL);
 ?column?
----------
 3 days 2 hours
```

```
=> SELECT CAST(INTERVAL -'01:15' as INTERVAL MINUTE);
 ?column?
----------
 -75 mins
```

Processing signed intervals

In the SQL:2008 standard, a minus sign before an interval-literal or as the first character of the interval-literal negates the entire literal, not just the first component. In HP Vertica, a leading minus sign negates the entire interval, not just the first component. The following commands both return the same value:

```
=> SELECT INTERVAL '-1 month - 1 second';
 ?column?
----------
 -29 days 23:59:59

=> SELECT INTERVAL -'1 month - 1 second';
 ?column?
----------
 -29 days 23:59:59
```

Use one of the following commands instead to return the intended result:

```
=> SELECT INTERVAL -'1 month 1 second';
 ?column?
----------
 -30 days 1 sec

=> SELECT INTERVAL -'30 00:00:01';
 ?column?
----------
 -30 days 1 sec
```

Two negatives together return a positive:

```
=> SELECT INTERVAL -'-1 month - 1 second';
 ?column?
----------
 29 days 23:59:59

=> SELECT INTERVAL -'-1 month 1 second';
 ?column?
----------
 30 days 1 sec
```

You can use the year-month syntax with no spaces. HP Vertica allows the input of negative months but requires two negatives when paired with years.

```
=> SELECT INTERVAL '3-3' YEAR TO MONTH;
 ?column?
----------
 3 years 3 months
=> SELECT INTERVAL '3--3' YEAR TO MONTH;
 ?column?
----------
 2 years 9 months
```

When the interval-literal looks like a year/month type, but the type is day/second, or vice versa, HP Vertica reads the interval-literal from left to right, where number-number is years-months, and number <space> <signed number> is whatever the units specify. HP Vertica processes the following command as (–) 1 year 1 month = (–) 365 + 30 = –395 days:

```
=> SELECT INTERVAL '-1-1' DAY TO HOUR;
 ?column?
----------
 -395 days
```

If you insert a space in the interval-literal, HP Vertica processes it based on the subtype `DAY TO HOUR`: (–) 1 day – 1 hour = (–) 24 – 1 = –23 hours:

```
=> SELECT INTERVAL '-1 -1' DAY TO HOUR;
 ?column?
----------
 -23 hours
```

Two negatives together returns a positive, so HP Vertica processes the following command as (–) 1 year – 1 month = (–) 365 – 30 = –335 days:

```
=> SELECT INTERVAL '-1--1' DAY TO HOUR;
 ?column?
----------
 -335 days
```

If you omit the value after the hyphen, HP Vertica assumes 0 months and processes the following command as 1 year 0 month –1 day = 365 + 0 – 1 = –364 days:

```
=> SELECT INTERVAL '1- -1' DAY TO HOUR;
 ?column?
----------
 364 days
```

Processing interval-literals without units

You can specify quantities of days, hours, minutes, and seconds without explicit units. HP Vertica recognizes colons in interval-literals as part of the timestamp:

```
=> SELECT INTERVAL '1 4 5 6';
 ?column?
------------
 1 day 04:05:06

=> SELECT INTERVAL '1 4:5:6';
 ?column?
------------
 1 day 04:05:06

=> SELECT INTERVAL '1 day 4 hour 5 min 6 sec';
 ?column?
------------
 1 day 04:05:06
```

If HP Vertica cannot determine the units, it applies the quantity to any missing units based on the interval-qualifier. In the next two examples, HP Vertica uses the default interval-qualifier (`DAY TO SECOND(6)`) and assigns the trailing `1` to days, since it has already processed hours, minutes, and seconds in the output:

```
=> SELECT INTERVAL '4:5:6 1';
 ?column?
------------
 1 day 04:05:06

=> SELECT INTERVAL '1 4:5:6';
 ?column?
------------
 1 day 04:05:06
```

In the next two examples, HP Vertica recognizes `4:5` as hours:minutes. The remaining values in the interval-literal are assigned to the missing units; `1` is assigned to days and `2` is assigned to seconds:

```
SELECT INTERVAL '4:5 1 2';
 ?column?
------------
 1 day 04:05:02

=> SELECT INTERVAL '1 4:5 2';
 ?column?
------------
 1 day 04:05:02
```

Specifying the interval-qualifier can change how HP Vertica interprets `4:5`:

```
=> SELECT INTERVAL '4:5' MINUTE TO SECOND;
 ?column?
------------
 00:04:05
```

Using INTERVALYM for INTERVAL YEAR TO MONTH

`INTERVALYM` is an alias for the `INTERVAL YEAR TO MONTH` subtypes and is used only on input:

```
=> SELECT INTERVALYM '1 2';
 ?column?
------------
 1 year 2 months
```

Operations with intervals

If you divide an interval by an interval, you get a FLOAT:

```
=> SELECT INTERVAL '28 days 3 hours' HOUR(4) / INTERVAL '27 days 3 hours' HOUR(4);
 ?column?
------------
```

```
 1.036866359447
```

An INTERVAL divided by FLOAT returns an INTERVAL:

```
=> SELECT INTERVAL '3' MINUTE / 1.5;
 ?column?
------------
 2 mins
```

INTERVAL MODULO (remainder) INTERVAL returns an INTERVAL:

```
=> SELECT INTERVAL '28 days 3 hours' HOUR % INTERVAL '27 days 3 hours' HOUR;
 ?column?
------------
 24 hours
```

If you add INTERVAL and TIME, the result is TIME, modulo 24 hours:

```
=> SELECT INTERVAL '1' HOUR + TIME '1:30';
 ?column?
------------
 02:30:00
```

Fractional seconds in interval units

HP Vertica supports intervals in milliseconds (hh:mm:ss:ms), where 01:02:03:25 represents 1 hour, 2 minutes, 3 seconds, and 025 milliseconds. Milliseconds are converted to fractional seconds as in the following example, which returns 1 day, 2 hours, 3 minutes, 4 seconds, and 25.5 milliseconds:

```
=> SELECT INTERVAL '1 02:03:04:25.5';
 ?column?
------------
 1 day 02:03:04.0255
```

HP Vertica allows fractional minutes. The fractional minutes are rounded into seconds:

```
=> SELECT INTERVAL '10.5 minutes';
 ?column?
------------
 00:10:30

=> select interval '10.659 minutes';
  ?column?
-------------
 00:10:39.54

=> select interval '10.3333333333333 minutes';
 ?column?
----------
 00:10:20
```

Notes

- The HP Vertica INTERVAL data type is SQL:2008 compliant, with extensions. On HP Vertica databases created prior to version 4.0, all `INTERVAL` columns are interpreted as `INTERVAL DAY TO SECOND`, as in the previous releases.
- An `INTERVAL` can include only the subset of units that you need; however, year/month intervals represent calendar years and months with no fixed number of days, so year/month interval values cannot include days, hours, minutes. When year/month values are specified for day/time intervals, the intervals extension assumes 30 days per month and 365 days per year. Since the length of a given month or year varies, day/time intervals are never output as months or years, only as days, hours, minutes, and so on.
- Day/time and year/month intervals are logically independent and cannot be combined with or compared to each other. In the following example, an interval-literal that contains `DAYS` cannot be combined with the `YEAR TO MONTH` type:

  ```
  => SELECT INTERVAL '1 2 3' YEAR TO MONTH;
  ERROR 3679:  Invalid input syntax for interval year to month: "1 2 3"
  ```
- HP Vertica accepts intervals up to 2^63 – 1 microseconds or months (about 18 digits).
- `INTERVAL YEAR TO MONTH` can be used in an analytic RANGE window when the `ORDER BY` column type is `TIMESTAMP/TIMESTAMP WITH TIMEZONE`, or `DATE`. Using `TIME/TIME WITH TIMEZONE` are not supported.
- You can use `INTERVAL DAY TO SECOND` when the `ORDER BY` column type is `TIMESTAMP/TIMESTAMP WITH TIMEZONE`, `DATE`, and `TIME/TIME WITH TIMEZONE`.

Examples

The table in this section contains additional interval examples. The `INTERVALSTYLE` is set to `PLAIN` (omitting units on output) for brevity.

Note: If you omit the ***interval-qualifier*** (page 40), the interval type defaults to `DAY TO SECOND(6)`.

Command	Result
`SELECT INTERVAL '00:2500:00';`	`1 17:40`
`SELECT INTERVAL '2500' MINUTE TO SECOND;`	`2500`
`SELECT INTERVAL '2500' MINUTE;`	`2500`
`SELECT INTERVAL '28 days 3 hours' HOUR TO SECOND;`	`675:00`
`SELECT INTERVAL(3) '28 days 3 hours';`	`28 03:00`
`SELECT INTERVAL(3) '28 days 3 hours 1.234567';`	`28 03:01:14.074`
`SELECT INTERVAL(3) '28 days 3 hours 1.234567 sec';`	`28 03:00:01.235`
`SELECT INTERVAL(3) '28 days 3.3 hours' HOUR TO SECOND;`	`675:18`
`SELECT INTERVAL(3) '28 days 3.35 hours' HOUR TO SECOND;`	`675:21`
`SELECT INTERVAL(3) '28 days 3.37 hours' HOUR TO SECOND;`	`675:22:12`
`SELECT INTERVAL '1.234567 days' HOUR TO SECOND;`	`29:37:46.5888`
`SELECT INTERVAL '1.23456789 days' HOUR TO SECOND;`	`29:37:46.665696`
`SELECT INTERVAL(3) '1.23456789 days' HOUR TO SECOND;`	`29:37:46.666`
`SELECT INTERVAL(3) '1.23456789 days' HOUR TO SECOND(2);`	`29:37:46.67`
`SELECT INTERVAL(3) '01:00:01.234567' as "one hour+";`	`01:00:01.235`
`SELECT INTERVAL(3) '01:00:01.234567' = INTERVAL(3) '01:00:01.234567';`	`t`
`SELECT INTERVAL(3) '01:00:01.234567' = INTERVAL '01:00:01.234567';`	`f`
`SELECT INTERVAL(3) '01:00:01.234567' = INTERVAL '01:00:01.234567' HOUR TO SECOND(3);`	`t`

SELECT INTERVAL(3) '01:00:01.234567' = INTERVAL '01:00:01.234567' MINUTE TO SECOND(3);	t
SELECT INTERVAL '255 1.1111' MINUTE TO SECOND(3);	255:01.111
SELECT INTERVAL '@ - 5 ago';	5
SELECT INTERVAL '@ - 5 minutes ago';	00:05
SELECT INTERVAL '@ 5 minutes ago';	-00:05
SELECT INTERVAL '@ ago -5 minutes';	00:05
SELECT DATE_PART('month', INTERVAL '2-3' YEAR TO MONTH);	3
SELECT FLOOR((TIMESTAMP '2005-01-17 10:00' - TIMESTAMP '2005-01-01') / INTERVAL '7');	2

See Also

Interval Values (page 37) for a description of the values that can be represented in an INTERVAL type

INTERVALSTYLE (page 906) and ***DATESTYLE*** (page 903)

AGE_IN_MONTHS (page 197) and ***AGE_IN_YEARS*** (page 198)

interval-literal

The following table lists the units allowed for the required `interval-literal` parameter.

Unit	Description
a	Julian year, 365.25 days exactly
ago	Indicates negative time offset
c, cent, century	Century
centuries	Centuries
d, day	Day
days	Days
dec, decade	Decade
decades, decs	Decades
h, hour, hr	Hour
hours, hrs	Hours
ka	Julian kilo-year, 365250 days exactly
m	Minute or month for year/month, depending on context. See Notes below this table.
microsecond	Microsecond
microseconds	Microseconds
mil, millennium	Millennium
millennia, mils	Millennia

millisecond	Millisecond
milliseconds	Milliseconds
min, minute, mm	Minute
mins, minutes	Minutes
mon, month	Month
mons, months	Months
ms, msec, millisecond	Millisecond
mseconds, msecs	Milliseconds
q, qtr, quarter	Quarter
qtrs, quarters	Quarters
s, sec, second	Second
seconds, secs	Seconds
us, usec	Microsecond
microseconds, useconds, usecs	Microseconds
w, week	Week
weeks	Weeks
y, year, yr	Year
years, yrs	Years

Processing the input unit 'm'

The input unit `'m'` can represent either 'months' or 'minutes,' depending on the context. For instance, the following command creates a one-column table with an interval value:

```
=> CREATE TABLE int_test(i INTERVAL YEAR TO MONTH);
```

In the first `INSERT` statement, the values are inserted as 1 year, six months:

```
=> INSERT INTO int_test VALUES('1 year 6 months');
```

The second `INSERT` statement results in an error from specifying minutes for a YEAR TO MONTH interval. At runtime, the result will be a NULL:

```
=> INSERT INTO int_test VALUES('1 year 6 minutes');
ERROR: invalid input syntax for type interval year to month: "1 year 6 minutes"
```

In the third `INSERT` statement, the 'm' is processed as months (not minutes), because `DAY TO SECOND` is truncated:

```
=> INSERT INTO int_test VALUES('1 year 6 m'); -- the m counts as months
```

The table now contains two identical values, with no minutes:

```
=> SELECT * FROM int_test;
  i
-----
```

```
 1 year 6 months
 1 year 6 months
(2 rows)
```

In the following command, the 'm' counts as minutes, because the DAY TO SECOND interval-qualifier extracts day/time values from the input:

```
=> SELECT INTERVAL '1y6m' DAY TO SECOND;
 ?column?
-----------
 365 days 6 mins
(1 row)
```

interval-qualifier

The following table lists the optional interval qualifiers. Values in INTERVAL fields, other than SECOND, are integers with a default precision of 2 when they are not the first field.

You cannot combine day/time and year/month qualifiers. For example, the following intervals are not allowed:

- DAY TO YEAR
- HOUR TO MONTH

Interval Type	Units	Valid interval-literal entries
Day/time intervals	DAY	Unconstrained.
	DAY TO HOUR	An interval that represents a span of days and hours.
	DAY TO MINUTE	An interval that represents a span of days and minutes.
	DAY TO SECOND	(Default) interval that represents a span of days, hours, minutes, seconds, and fractions of a second if subtype unspecified.
	HOUR	Hours within days.
	HOUR TO MINUTE	An interval that represents a span of hours and minutes.
	HOUR TO SECOND	An interval that represents a span of hours and seconds.
	MINUTE	Minutes within hours.
	MINUTE TO SECOND	An interval that represents a span of minutes and seconds.

	SECOND	Seconds within minutes. **Note:** The SECOND field can have an interval fractional seconds precision, which indicates the number of decimal digits maintained following the decimal point in the SECONDS value. When SECOND is not the first field, it has a precision of 2 places before the decimal point.
Year/month intervals	MONTH	Months within year.
	YEAR	Unconstrained.
	YEAR TO MONTH	An interval that represents a span of years and months.

SMALLDATETIME

SMALLDATETIME is an alias for ***TIMESTAMP*** (page 97).

TIME

Consists of a time of day with or without a time zone.

Syntax

```
TIME [ (p) ] [ { WITH | WITHOUT } TIME ZONE ] | TIMETZ
[ AT TIME ZONE (see "TIME AT TIME ZONE" on page 96) ]
```

Parameters

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

Name	Low Value	High Value	Resolution
TIME [*p*]	00:00:00.00	23:59:60.999999	1 µs
TIME [*p*] WITH TIME ZONE	00:00:00.00+14	23:59:59.999999-14	1 µs

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

Notes

- HP Vertica permits coercion from TIME and TIME WITH TIME ZONE types to TIMESTAMP or TIMESTAMP WITH TIME ZONE or INTERVAL (Day to Second).
- HP Vertica supports adding milliseconds to a TIME or TIMETZ value.

```
=> CREATE TABLE temp (datecol TIME);
=> INSERT INTO temp VALUES (TIME '12:47:32.62');
=> INSERT INTO temp VALUES (TIME '12:55:49.123456');
=> INSERT INTO temp VALUES (TIME '01:08:15.12374578');
=> SELECT * FROM temp;
     datecol
-----------------
 12:47:32.62
 12:55:49.123456
 01:08:15.123746
(3 rows)
```

See Also

Data Type Coercion Chart (page 115)

TIME AT TIME ZONE

The `TIME AT TIME ZONE` construct converts `TIMESTAMP` and `TIMESTAMP WITH ZONE` types to different time zones.

TIME ZONE is a synonym for TIMEZONE. Both are allowed in HP Vertica syntax.

Syntax

timestamp AT TIME ZONE *zone*

Parameters

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. The files in /opt/vertica/share/timezonesets define the default list of strings accepted in the *zone* parameter	

Examples

The local time zone is PST8PDT. The first example takes a zone-less timestamp and interprets it as MST time (UTC- 7) to produce a UTC timestamp, which is then rotated to PST (UTC-8) for display:

```
=> SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
        timezone
------------------------
 2001-02-16 22:38:40-05
(1 row)
```

The second example takes a timestamp specified in EST (UTC-5) and converts it to local time in MST (UTC-7):

```
=> SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
      timezone
---------------------
 2001-02-16 18:38:40
(1 row)
```

TIMESTAMP

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

Syntax

TIMESTAMP [(*p*)] [{ WITH | WITHOUT } TIME ZONE] | TIMESTAMPTZ
[***AT TIME ZONE*** (see **"*TIME AT TIME ZONE*"** on page 96)]

Parameters

p	Optional precision value that specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of *p* is 0 to 6.
WITH TIME ZONE	Specifies that valid values must include a time zone. All TIMESTAMP WITH TIME ZONE values are stored internally in UTC. They are converted to local time in the zone specified by the time zone 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.

Limits

In the following table, values are rounded. See ***Date/Time Data Types*** (page 78) for additional detail.

Name	Low Value	High Value	Resolution
TIMESTAMP [(p)] [WITHOUT TIME ZONE]	290279 BC	294277 AD	1 µs
TIMESTAMP [(p)] WITH TIME ZONE	290279 BC	294277 AD	1 µs

Notes

- TIMESTAMP is an alias for DATETIME and SMALLDATETIME.
- Valid input for TIMESTAMP types consists of a concatenation of a date and a time, followed by an optional time zone, followed by an optional AD or BC.
- 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.

  TIMESTAMP '2004-10-19 10:23:54**+02**' is a TIMESTAMP WITH TIME ZONE.

  Note: HP 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, HP 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. 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 TIME ZONE parameter, and is converted to UTC using the offset for the TIME ZONE zone.
- When a TIMESTAMP WITH TIME ZONE value is output, it is always converted from UTC to the current TIME ZONE zone and displayed as local time in that zone. To see the time in another time zone, either change TIME ZONE or use the AT TIME ZONE (page 102) construct.
- Conversions between TIMESTAMP WITHOUT TIME ZONE and TIMESTAMP WITH TIME ZONE normally assume that the TIMESTAMP WITHOUT TIME ZONE value are taken or given as TIME ZONE local time. A different zone reference can be specified for the conversion using AT TIME ZONE.
- TIMESTAMPTZ and TIMETZ are not parallel SQL constructs. TIMESTAMPTZ records a time and date in GMT, converting from the specified TIME ZONE. TIMETZ records the specified time and the specified time zone, in minutes, from GMT.timezone
- The following list represents typical date/time input variations:
  - 1999-01-08 04:05:06
  - 1999-01-08 04:05:06 -8:00
  - January 8 04:05:06 1999 PST
- HP Vertica supports adding a floating-point (in days) to a TIMESTAMP or TIMESTAMPTZ value.
- HP Vertica supports adding milliseconds to a TIMESTAMP or TIMESTAMPTZ value.
- In HP Vertica, ***intervals*** (page 81) are represented internally as some number of microseconds and printed as up to 60 seconds, 60 minutes, 24 hours, 30 days, 12 months, and as many years as necessary. Fields are either positive or negative.

Examples

You can return infinity by specifying 'infinity':

```
=> SELECT TIMESTAMP 'infinity';
 timestamp
-----------
 infinity
(1 row)
```

To use the minimum TIMESTAMP value lower than the minimum rounded value:

```
=> SELECT '-infinity'::timestamp;
 timestamp
-----------
 -infinity
(1 row)
```

TIMESTAMP/TIMESTAMPTZ has +/-infinity values.

AD/BC can be placed almost anywhere within the input string; for example:

```
SELECT TIMESTAMPTZ 'June BC 1, 2000 03:20 PDT';
        timestamptz
```

```
--------------------------
 2000-06-01 05:20:00-05 BC
(1 row)
```

Notice the results are the same if you move the BC after the 1:

```
SELECT TIMESTAMPTZ 'June 1 BC, 2000 03:20 PDT';
        timestamptz
--------------------------
 2000-06-01 05:20:00-05 BC
(1 row)
```

And the same if you place the BC in front of the year:

```
SELECT TIMESTAMPTZ 'June 1, BC 2000 03:20 PDT';
        timestamptz
--------------------------
 2000-06-01 05:20:00-05 BC
(1 row);
```

The following example returns the year 45 before the Common Era:

```
=> SELECT TIMESTAMP 'April 1, 45 BC';
       timestamp
------------------------
 0045-04-01 00:00:00 BC
(1 row)
```

If you omit the BC from the date input string, the system assumes you want the year 45 in the current century:

```
=> SELECT TIMESTAMP 'April 1, 45';
      timestamp
---------------------
 2045-04-01 00:00:00
(1 row)
```

In the following example, HP Vertica returns results in years, months, and days, whereas other RDBMS might return results in days only:

```
=> SELECT TIMESTAMP WITH TIME ZONE '02/02/294276'- TIMESTAMP WITHOUT TIME ZONE
'02/20/2009' AS result;
           result
-----------------------------
 292266 years 11 mons 12 days
(1 row)
```

To specify a specific time zone, add it to the statement, such as the use of 'ACST' in the following example:

```
=> SELECT T1 AT TIME ZONE 'ACST', t2 FROM test;
      timezone       | t2
---------------------+-------------
 2009-01-01 04:00:00 | 02:00:00-07
 2009-01-01 01:00:00 | 02:00:00-04
 2009-01-01 04:00:00 | 02:00:00-06
```

You can specify a floating point in days:

```
=> SELECT 'NOW'::TIMESTAMPTZ + INTERVAL '1.5 day' AS '1.5 days from now';
```

```
       1.5 days from now
-------------------------------
 2009-03-18 21:35:23.633-04
(1 row)
```

The following example illustrates the difference between TIMESTAMPTZ with and without a precision specified:

```
=> SELECT TIMESTAMPTZ(3) 'now', TIMESTAMPTZ 'now';
        timestamptz         |          timestamptz
----------------------------+-------------------------------
 2009-02-24 11:40:26.177-05 | 2009-02-24 11:40:26.177368-05
(1 row)
```

The following statement returns an error because the TIMESTAMP is out of range:

```
=> SELECT TIMESTAMP '294277-01-09 04:00:54.775808';
ERROR:  date/time field value out of range: "294277-01-09 04:00:54.775808"
```

There is no 0 AD, so be careful when you subtract BC years from AD years:

```
=> SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
      2001
(1 row)
```

The following commands create a table with a TIMESTAMP column that contains milliseconds:

```
CREATE TABLE temp (datecol TIMESTAMP);
INSERT INTO temp VALUES (TIMESTAMP '2010-03-25 12:47:32.62');
INSERT INTO temp VALUES (TIMESTAMP '2010-03-25 12:55:49.123456');
INSERT INTO temp VALUES (TIMESTAMP '2010-03-25 01:08:15.12374578');
 SELECT * FROM temp;
          datecol
----------------------------
 2010-03-25 12:47:32.62
 2010-03-25 12:55:49.123456
 2010-03-25 01:08:15.123746
(3 rows)
```

Additional Examples

Command	Result
select (timestamp '2005-01-17 10:00' - timestamp '2005-01-01');	16 10:10
select (timestamp '2005-01-17 10:00' - timestamp '2005-01-01') / 7;	2 08:17:08.571429
select (timestamp '2005-01-17 10:00' - timestamp '2005-01-01') day;	16
select cast((timestamp '2005-01-17 10:00' - timestamp '2005-01-01') day as integer) / 7;	2
select floor((timestamp '2005-01-17 10:00' - timestamp '2005-01-01') / interval '7');	2
select timestamptz '2009-05-29 15:21:00.456789';	2009-05-29 15:21:00.456789-04
select timestamptz '2009-05-28';	2009-05-28 00:00:00-04
select timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28';	1 15:21:00.456789
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz	1 15:21:00.456789

'2009-05-28');	
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28')(3);	1 15:21:00.457
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28')second;	141660.456789
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28') year;	0
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2007-01-01') month;	28
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2007-01-01') year;	2
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2007-01-01') year to month;	2-4
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28') second(3);	141660.457
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28') minute(3);	2361
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28') minute;	2361
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28') minute to second(3);	2361:00.457
select (timestamptz '2009-05-29 15:21:00.456789'-timestamptz '2009-05-28') minute to second;	2361:00.456789

TIMESTAMP AT TIME ZONE

The TIMESTAMP AT TIME ZONE (or TIMEZONE) construct converts TIMESTAMP and TIMESTAMP WITH TIMEZONE intervals to different time zones.

NOTE: TIME ZONE is a synonym for TIMEZONE. Both are allowed in HP Vertica syntax.

Syntax

timestamp AT TIME ZONE *zone*

Parameters

timestamp	TIMESTAMP	Converts UTC to local time in the given time zone
	TIMESTAMP WITH TIME ZONE	Converts local time in given time zone to UTC
	TIME	Converts local time.
	TIME WITH TIME ZONE	Converts local time across time zones
zone	Specifies the time zone either as a text string, (such as 'America/Chicago') or as an interval (INTERVAL '-08:00'). The preferred way to express a time zone is in the format 'America/Chicago'. For a list of time zone text strings, see Set the Default Time Zone in the Installation Guide. To view the default list of acceptable strings for the *zone* parameter, see the files in: /opt/vertica/share/timezonesets	

Examples

If you indicate a TIME interval timezone (such as `America/Chicago` in the following example), the interval function converts the interval to the timezone you specify and includes the UTC offset value (`-05` here):

```
=> select time '10:00' at time zone 'America/Chicago';
      ?column?
--------------------
  09:00:00-05
(1 row)
```

Casting a TIMESTAMPTZ interval to a TIMESTAMP without a zone depends on the local time zone.

```
=> select (varchar '2013-03-31 5:10 AMERICA/CHICAGO')::timestamp;
      ?column?
---------------------
 2013-03-31 06:10:00
(1 row)
```

NOTE: For a complete list of valid time zone definitions, see ***Wikipedia - tz database time zones http://en.wikipedia.org/wiki/List_of_tz_database_time_zones***.

Casting a TIME (or TIMETZ) interval to a TIMESTAMP returns the local date and time, without the UTC offset:

```
=> select (time '3:01am')::timestamp;
      ?column?
---------------------
 2012-08-30 03:01:00
(1 row)

=> select (timetz '3:01am')::timestamp;
      ?column?
---------------------
 2012-08-22 03:01:00
(1 row)
```

Casting the same interval (TIME or TIMETZ) to a TIMESTAMPTZ returns the local date and time appended with the UTC offset (`-04` here):

```
=> select (time '3:01am')::timestamptz;
      ?column?
---------------------
 2012-08-30 03:01:00-04
(1 row)
```

Numeric Data Types

Numeric data types are numbers stored in database columns. These data types are typically grouped by:

- **Exact** numeric types , values where the precision and scale need to be preserved. The exact numeric types are `BIGINT`, `DECIMAL`, `INTEGER`, `NUMERIC`, `NUMBER`, and `MONEY`.
- **Approximate** numeric types, values where the precision needs to be preserved and the scale can be floating. The approximate numeric types are `DOUBLE PRECISION`, `FLOAT`, and `REAL`.

Implicit casts from `INTEGER`, `FLOAT`, and `NUMERIC` to `VARCHAR` are not supported. If you need that functionality, write an explicit cast using one of the following forms:

`CAST(x AS data-type-name)` or `x::data-type-name`

The following example casts a float to an integer:

```
=> SELECT(FLOAT '123.5')::INT;
 ?column?
----------
      124
(1 row)
```

String-to-numeric data type conversions accept formats of quoted constants for scientific notation, binary scaling, hexadecimal, and combinations of numeric-type literals:

- Scientific notation:
  ```
  => SELECT FLOAT '1e10';
    ?column?
  -------------
   10000000000
  (1 row)
  ```
- `BINARY` scaling:
  ```
  => SELECT NUMERIC '1p10';
   ?column?
  ----------
       1024
  (1 row)
  ```
- Hexadecimal:
  ```
  => SELECT NUMERIC '0x0abc';
   ?column?
  ----------
       2748
  (1 row)
  ```

DOUBLE PRECISION (FLOAT)

HP 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 | FLOAT(n) | FLOAT8 | REAL ]
```

Parameters

Note: 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 might show slight discrepancies.

- All of the `DOUBLE PRECISION` data types are synonyms for 64-bit IEEE FLOAT.
- The *n* in `FLOAT(n)` must be between 1 and 53, inclusive, but a 53-bit fraction is always used. See the IEEE-754 standard for details.
- For exact numeric storage and calculations (money for example), use `NUMERIC`.
- Floating point calculations depend on the behavior of the underlying processor, operating system, and compiler.
- Comparing two floating-point values for equality might not work as expected.

Values

`COPY` (page 699) accepts floating-point data in the following format:

- Optional leading white space
- An optional plus ("+") or minus sign ("-")
- 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 "."), 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 HP 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: HP 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.

HP Vertica uses a different NaN value to represent floating point NULL (0x7ffffffffffffffeLL). This is a Positive, Quiet, Non-signaling NaN and is reserved by HP Vertica

The load file format of a null value is user defined, as described in the `COPY` (page 699) command. The HP 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

- `NULL` appears last (largest) in ascending order.
- All overflows in floats generate +/-infinity or NaN, per the IEEE floating point standard.

INTEGER

A signed 8-byte (64-bit) data type.

Syntax

```
[ INTEGER | INT | BIGINT | INT8 | SMALLINT | TINYINT ]
```

Parameters

INT, INTEGER, INT8, SMALLINT, TINYINT, and BIGINT are all synonyms for the same signed 64-bit integer data type. Automatic compression techniques are used to conserve disk space in cases where the full 64 bits are not required.

Notes

- 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.
- NULL appears first (smallest) in ascending order.
- HP Vertica does not have an explicit 4-byte (32-bit integer) or smaller types. HP Vertica's encoding and compression automatically eliminate the storage overhead of values that fit in less than 64 bits.

Restrictions

- The JDBC type INTEGER is 4 bytes and is not supported by HP Vertica. Use BIGINT instead.
- HP Vertica does not support the SQL/JDBC types NUMERIC, SMALLINT, or TINYINT.
- HP Vertica does not check for overflow (positive or negative) except in the aggregate function SUM (page 137)(). If you encounter overflow when using SUM, use SUM_FLOAT (page 138)(), which converts to floating point.

See Also

Data Type Coercion Chart (page 115)

NUMERIC

Numeric data types store numeric data. For example, a money value of $123.45 can be stored in a NUMERIC(5,2) field.

Syntax

```
NUMERIC | DECIMAL | NUMBER | MONEY [ ( precision [ , scale ] ) ]
```

Parameters

precision	The total number of significant digits that the data type stores.

	precision must be positive and <= 1024. If you assign a value that exceeds the *precision* value, an error occurs.
scale	The maximum number of digits to the right of the decimal point that the data type stores. *scale* must be non-negative and less than or equal to *precision*. If you omit the *scale* parameter, the *scale* value is set to 0. If you assign a value with more decimal digits than *scale*, the value is rounded to *scale* digits.

Notes

- NUMERIC, DECIMAL, NUMBER, and MONEY are all synonyms that return NUMERIC types. However, the default values NUMBER and MONEY are different:

```
Type        Precision  Scale
NUMERIC     37         15
DECIMAL     37         15
NUMBER      38         0
MONEY       18         4
```

- NUMERIC data types support exact representations of numbers that can be expressed with a number of digits before and after a decimal point. This contrasts slightly with existing HP Vertica data types:
  - DOUBLE PRECISION (page 105) (FLOAT) types support ~15 digits, variable exponent, and represent numeric values approximately.
  - INTEGER (page 107) (and similar) types support ~18 digits, whole numbers only.
- NUMERIC data types are generally called *exact* numeric data types because they store numbers of a specified precision and scale. The *approximate* numeric data types, such as DOUBLE PRECISION, use floating points and are less precise.
- Supported numeric operations include the following:
  - Basic math: +, –, *, /
  - Aggregation: SUM, MIN, MAX, COUNT
  - Comparison operators: <, <=, =, <=>, <>, >, >=
- NUMERIC divide operates directly on numeric values, without converting to floating point. The result has at least 18 decimal places and is rounded.
- NUMERIC mod (including %) operates directly on numeric values, without converting to floating point. The result has the same scale as the numerator and never needs rounding.
- NULL appears first (smallest) in ascending order.
- ***COPY*** (page 699) accepts a DECIMAL data type with a decimal point ('.'), prefixed by – or +(optional).
- LZO, RLE, and BLOCK_DICT are supported encoding types. Anything that can be used on an INTEGER can also be used on a NUMERIC, as long as the precision is <= 18.
- The NUMERIC data type is preferred for non-integer constants, because it is always exact. For example:

```
=> SELECT 1.1 + 2.2 = 3.3;
 ?column?
----------
 t
```

```
(1 row)
=>  SELECT 1.1::float + 2.2::float = 3.3::float;
 ?column?
----------
 f
(1 row)
```

- Performance of the NUMERIC data type has been fine tuned for the common case of 18 digits of precision.
- Some of the more complex operations used with NUMERIC data types result in an implicit cast to FLOAT. When using SQRT, STDDEV, transcendental functions such as LOG, and TO_CHAR/TO_NUMBER formatting, the result is always FLOAT.

Examples

The following series of commands creates a table that contains a NUMERIC data type and then performs some mathematical operations on the data:

```
=> CREATE TABLE num1 (id INTEGER, amount NUMERIC(8,2));
```

Insert some values into the table:

```
=> INSERT INTO num1 VALUES (1, 123456.78);
```

Query the table:

```
=> SELECT * FROM num1;
  id  |  amount
------+-----------
    1 | 123456.78
(1 row)
```

The following example returns the NUMERIC column, amount, from table num1:

```
=> SELECT amount FROM num1;
  amount
-----------
 123456.78
(1 row)
```

The following syntax adds one (1) to the amount:

```
=> SELECT amount+1 AS 'amount' FROM num1;
  amount
-----------
 123457.78
(1 row)
```

The following syntax multiplies the amount column by 2:

```
=> SELECT amount*2 AS 'amount' FROM num1;
  amount
-----------
 246913.56
(1 row)
```

The following syntax returns a negative number for the amount column:

```
=> SELECT -amount FROM num1;
```

```
  ?column?
------------
 -123456.78
(1 row)
```

The following syntax returns the absolute value of the `amount` argument:

```
=> SELECT ABS(amount) FROM num1;
    ABS
-----------
 123456.78
(1 row)
```

The following syntax casts the `NUMERIC amount` as a `FLOAT` data type:

```
=> SELECT amount::float FROM num1;
  amount
-----------
 123456.78
(1 row)
```

See Also

Mathematical Functions (page 300)

Numeric data type overflow

HP Vertica does not check for overflow (positive or negative) except in the aggregate function `SUM (page 137)()`. If you encounter overflow when using `SUM`, use `SUM_FLOAT (page 138)()` which converts to floating point.

Dividing zero by zero returns zero:

```
=> select 0/0;
       ?column?
----------------------
0.000000000000000000
(1 row)


=> select 0.0/0;
       ?column?
-----------------------
0.0000000000000000000


=> select 0 // 0;
 ?column?
----------
        0
```

Dividing zero as a FLOAT by zero returns NaN:

```
=> select 0.0::float/0;
```

```
?column?
----------
     NaN


=> select 0.0::float//0;
 ?column?
----------
      NaN
```

Dividing a non-zero FLOAT by zero returns Infinity:

```
=> select 2.0::float/0;
?column?
----------
Infinity


=> select 200.0::float//0;
?column?
----------
Infinity
```

All other division-by-zero operations return an error:

```
=> select 1/0;
ERROR 3117:  Division by zero
=> select 200/0;
ERROR 3117:  Division by zero
=> select 200.0/0;
ERROR 3117:  Division by zero
=> select 116.43 // 0;
ERROR 3117:  Division by zero
```

Add, subtract, and multiply operations ignore overflow. Sum and average operations use 128-bit arithmetic internally. `SUM (page 137)()` reports an error if the final result overflows, suggesting the use of `SUM_FLOAT (page 138)(INT)`, which converts the 128-bit sum to a `FLOAT8`. For example:

```
=> CREATE TEMP TABLE t (i INT);
=> INSERT INTO t VALUES (1<<62);
=> INSERT INTO t VALUES (1<<62);
=> INSERT INTO t VALUES (1<<62);
=> INSERT INTO t VALUES (1<<62);
=> INSERT INTO t VALUES (1<<62);
=> SELECT SUM(i) FROM t;
   ERROR: sum() overflowed
   HINT: try sum_float() instead
=> SELECT SUM_FLOAT(i) FROM t;
     sum_float
---------------------
 2.30584300921369e+19
```

Data Type Coercion

HP Vertica currently has two types of cast, implicit and explicit. HP Vertica implicitly casts (coerces) expressions from one type to another under certain circumstances.

NOTE: Non-standard implicit casts from numeric types to varchar are not supported. Use explicit casts if required. A non-standard implicit cast from char to float exists to match the one from varchar to float.

When there is no ambiguity as to the data type of an expression value, it is implicitly coerced to match the expected data type. In the following command,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';
 ?column?
----------
        4
(1 row)
```

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

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

Another example is to first get today's date:

```
=> SELECT DATE 'now';
  ?column?
------------
 2012-05-30
(1 row)
```

The following command converts DATE to a TIMESTAMP and adds a day and a half to the results by using INTERVAL:

```
=> SELECT DATE 'now' + INTERVAL '1 12:00:00';
      ?column?
---------------------
 2012-05-31 12:00:00
(1 row)
```

Most implicit casts stay within their relational family and go in one direction, from less detailed to more detailed. For example:

- DATE to TIMESTAMP/TZ
- INTEGER to NUMERIC to FLOAT
- CHAR to FLOAT
- CHAR to VARCHAR
- CHAR and/or VARCHAR to FLOAT

More specifically, data type coercion works in this manner in HP Vertica:

Type	Direction	Type	Notes
INT8	>	FLOAT8	Implicit, can lost significance
FLOAT8	>	INT8	Explicit, rounds
VARCHAR	<->	CHAR	Implicit, adjusts trailing spaces
VARBINARY	<->	BINARY	Implicit, adjusts trailing NULs

No other types cast to or from varbinary or binary. In the following list, <any> means one these types: INT8, FLOAT8, DATE, TIME, TIMETZ, TIMESTAMP, TIMESTAMPTZ, INTERVAL

- <any> -> VARCHAR—implicit
- VARCHAR -> <any>—explicit, except that VARCHAR->FLOAT is implicit
- <any> <-> CHAR—explicit
- DATE -> TIMESTAMP/TZ—implicit
- TIMESTAMP/TZ -> DATE—explicit, loses time-of-day
- TIME -> TIMETZ—implicit, adds local timezone
- TIMETZ -> TIME—explicit, loses timezone
- TIME -> INTERVAL—implicit, day to second with days=0
- INTERVAL -> TIME—explicit, truncates non-time parts
- TIMESTAMP <-> TIMESTAMPTZ—implicit, adjusts to local timezone
- TIMESTAMP/TZ -> TIME—explicit, truncates non-time parts
- TIMESTAMPTZ -> TIMETZ—explicit

IMPORTANT: Implicit casts from `INTEGER`, `FLOAT`, and `NUMERIC` to `VARCHAR` are not supported. If you need that functionality, write an explicit cast:

```
CAST(x AS data-type-name)
```

or

```
x::data-type-name
```

The following example casts a `FLOAT` to an `INTEGER`:

```
=> SELECT(FLOAT '123.5')::INT;
 ?column?
----------
      124
(1 row)
```

String-to-numeric data type conversions accept formats of quoted constants for scientific notation, binary scaling, hexadecimal, and combinations of numeric-type literals:

- Scientific notation:

```
=> SELECT FLOAT '1e10';
  ?column?
-------------
 10000000000
(1 row)
```

- `BINARY` scaling:

```
=> SELECT NUMERIC '1p10';
 ?column?
----------
     1024
(1 row)
```

- Hexadecimal:

```
=> SELECT NUMERIC '0x0abc';
 ?column?
----------
     2748
(1 row)
```

Examples

```
=> SELECT NUMERIC '12.3e3', '12.3p10'::NUMERIC, CAST('0x12.3p-10e3' AS NUMERIC);
 ?column? | ?column? |     ?column?
----------+----------+------------------
    12300 |  12595.2 | 17.76123046875000
(1 row)

=> SELECT (18. + 3./16)/1024*1000;
                ?column?
----------------------------------------
 17.761230468750000000000000000000000000
(1 row)
```

Note: In SQL expressions, pure numbers between -(2^63-1) and (2^63-1) are `INTEGERs`; numbers with decimal points are `NUMERIC`, and do not support the above notation. Numbers using `e` notation are `FLOAT`.

The following two examples show queries that once work but now fail; below the failed query is a rewrite with the cast to VARCHAR to make such queries work again:

```
=> SELECT TO_NUMBER(1);
ERROR:  function to_number(int) does not exist
HINT:  No function matches the given name and argument types. You may need to add
explicit type casts.

=> SELECT TO_NUMBER(1::VARCHAR);
 to_number
-----------
         1
(1 row)

=> SELECT TO_DATE(20100302, 'YYYYMMDD');
ERROR:  function to_date(int, "unknown") does not exist
```

```
HINT:  No function matches the given name and argument types. You may need to add
explicit type casts.

=> SELECT TO_DATE(20100302::VARCHAR, 'YYYYMMDD');
  to_date
------------
 2010-03-02
(1 row)
```

See Also

Data Type Coercion Chart (page 115)

Data Type Coercion Operators (CAST) (page 45)

Data Type Coercion Chart

Conversion Types

The following table defines all possible type conversions that HP Vertica supports. The values across the top row are the data types you want, and the values down the first column on the left are the data types that you have.

Want> Have	BOOL	INT	NUM	FLT	VCHR	CHAR	TS	TSTZ	DATE	TIME	TTZ	INTDS	INTYM	VBIN	BIN
BOOL	N/A	a			a	a									
INT	i	N/A	i	i	a**	a**						a	a		
NUM		a	Yes	i	a**	a**									
FLT		a	a	N/A	a**	a**									
VCHR	e	e	e	i	Yes	i	e	e	e	e	e	e	e		
CHAR	e	e	e	i	i	Yes	e	e	e	e	e	e	e		
TS					a	a	Yes	i	a	a					
TSTZ					a	a	i	Yes	a	a	a				
DATE					a	a	i	a	N/A						
TIME					a	a	e	e		Yes	i	e			
TTZ					a	a	e	e		a	Yes				
INTDS		a			a	a				e		Yes			
INTYM		a			a	a							Yes		
VBIN														Yes	i
BIN														i	Yes

Notes

- i = implicit. HP Vertica implicitly converts the source data to the target column's data type when what needs to be converted is clear. For example, with "INT + NUMERIC -> NUMERIC", the integer is implicitly cast to numeric(18,0); another precision/scale conversion may occur as part of the add.

- a = assignment. Coercion implicitly occurs when values are assigned to database columns in an INSERT or UPDATE..SET command. For example, in a statement that includes INSERT ... VALUES('2.5'), where the target column is NUMERIC(18,5), a cast from VARCHAR to NUMERIC(18,5) is inferred.
- e = explicit. The source data requires explicit casting to the target column's data type.
- N/A — no possible conversion can take place (such as INT->INT).
- Yes — HP Vertica supports a conversion of data types without explicit casting, such as NUMERIC(10,6) -> NUMERIC(18,4).
- Double asterisks (**) mean that the numeric meaning is lost, and the value is subject to VARCHAR/CHAR compares.

Abbreviations used in table

- BOOL = Boolean
- INT = Integer
- NUM = Numeric
- FLT = Float
- VCHR = Varchar
- TS = Timestamp
- TSTZ = Timestamp with Time Zone
- TTZ = Time with Time Zone
- INTDS = Interval Day/Second
- INTYM = Interval Year/Month
- VBIN = Varbinary
- BIN = Binary

See Also

Data Type Coercion Operators (CAST) (page 45)

SQL Functions

Functions return information from the database and are allowed anywhere an expression is allowed. The exception is ***HP Vertica-specific functions*** (page 425), which are not allowed everywhere.

Some functions could produce different results on different invocations with the same set of arguments. The following three categories of functions are defined based on their behavior:

- **Immutable (invariant)**: When run with a given set of arguments, immutable functions always produce the same result. The function is independent of any environment or session settings, such as locale. For example, 2+2 always equals 4. Another immutable function is AVG(). Some immutable functions can take an optional stable argument; in this case they are treated as stable functions.
- **Stable**: When run with a given set of arguments, stable functions produce the same result within a single query or scan operation. However, a stable function could produce different results when issued under a different environment, such as a change of locale and time zone. Expressions that could give different results in the future are also stable, for example `SYSDATE()` or `'today'`.
- **Volatile**: Regardless of the arguments or environment, volatile functions can return different results on multiple invocations. RANDOM() is one example.

This chapter describes the functions that HP Vertica supports.

- Each function is annotated with behavior type as immutable, stable or volatile.
- All HP Vertica-specific functions can be assumed to be volatile and are not annotated individually.

Aggregate Functions

Note: All functions in this section that have an ***analytic*** (page 141) function counterpart are appended with [Aggregate] to avoid confusion between the two.

Aggregate functions summarize data over groups of rows from a query result set. The groups are specified using the ***GROUP BY*** (page 878) clause. They are allowed only in the select list and in the ***HAVING*** (page 880) and ***ORDER BY*** (page 893) clauses of a ***SELECT*** (page 870) statement (as described in ***Aggregate Expressions*** (page 51)).

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).
- HP Vertica does not support nested aggregate functions.

You can also use some of the simple aggregate functions as analytic (window) functions. See ***Analytic Functions*** (page 141) for details. See also Using SQL Analytics in the Programmer's Guide.

AVG [Aggregate]

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.

Behavior Type

Immutable

Syntax

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

Parameters

`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`	The value whose average is calculated over a set of rows. Can be any expression resulting in `DOUBLE PRECISION`.

Notes

The AVG() aggregate function is different from the AVG() analytic function, which computes an average of an expression over a group of rows within a window.

Examples

The following example returns the average income from the customer table:

```
=> SELECT AVG(annual_income) FROM customer_dimension;
     avg
--------------
 2104270.6485
(1 row)
```

See Also

AVG (page 150) analytic function

COUNT (page 123) and ***SUM*** (page 137)

Numeric Data Types (page 103)

BIT_AND

Takes the bitwise AND of all non-null input values. If the input parameter is NULL, the return value is also NULL.

Behavior Type

Immutable

Syntax

```
BIT_AND ( expression )
```

Parameters

expression	The [BINARY |VARBINARY] input value to be evaluated. BIT_AND() operates on VARBINARY types explicitly and on BINARY types implicitly through ***casts*** (page 115).

Notes

- The function returns the same value as the argument data type.
- For each bit compared, if **all** bits are 1, the function returns 1; otherwise it returns 0.
- If the columns are different lengths, the return values are treated as though they are all equal in length and are right-extended with zero bytes. For example, given a group containing the hex values 'ff', null, and 'f', the function ignores the null value and extends the value 'f' to 'f0'..

Example

This examples uses the following schema, which creates table `t` with a single column of `VARBINARY` data type:

```
=> CREATE TABLE t (
      c VARBINARY(2) );
=> INSERT INTO t values(HEX_TO_BINARY('0xFF00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFFFF'));
=> INSERT INTO t values(HEX_TO_BINARY('0xF00F'));
```

Query table `t` to see column `c` output:

```
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 ff00
 ffff
 f00f
(3 rows)
```

Query table `t` to get the AND value for column `c`:

```
SELECT TO_HEX(BIT_AND(c)) FROM t;
 TO_HEX
--------
 f000
(1 row)
```

The function is applied pairwise to all values in the group, resulting in `f000`, which is determined as follows:

1. `ff00` (record 1) is compared with `ffff` (record 2), which results in `ff00`.
2. The result from the previous comparison is compared with `f00f` (record 3), which results in `f000`.

See Also

Binary Data Types (page 72)

BIT_OR

Takes the bitwise `OR` of all non-null input values. If the input parameter is `NULL`, the return value is also `NULL`.

Behavior Type

Immutable

Syntax

`BIT_OR ( expression )`

Parameters

expression	The [BINARY |VARBINARY] input value to be evaluated. BIT_OR() operates on VARBINARY types explicitly and on BINARY types implicitly through ***casts*** (page 115).

Notes

- The function returns the same value as the argument data type.
- For each bit compared, if **any** bit is 1, the function returns 1; otherwise it returns 0.
- If the columns are different lengths, the return values are treated as though they are all equal in length and are right-extended with zero bytes. For example, given a group containing the hex values 'ff', null, and 'f', the function ignores the null value and extends the value 'f' to 'f0'.

Example

This examples uses the following schema, which creates table t with a single column of VARBINARY data type:

```
=> CREATE TABLE t (
      c VARBINARY(2) );
=> INSERT INTO t values(HEX_TO_BINARY('0xFF00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFFFF'));
=> INSERT INTO t values(HEX_TO_BINARY('0xF00F'));
```

Query table t to see column c output:

```
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 ff00
 ffff
 f00f
(3 rows)
```

Query table t to get the OR value for column c:

```
SELECT TO_HEX(BIT_OR(c)) FROM t;
 TO_HEX
--------
 ffff
(1 row)
```

The function is applied pairwise to all values in the group, resulting in ffff, which is determined as follows:

1. ff00 (record 1) is compared with ffff, which results in ffff.
2. The ff00 result from the previous comparison is compared with f00f (record 3), which results in ffff.

See Also

Binary Data Types (page 72)

BIT_XOR

Takes the bitwise XOR of all non-null input values. If the input parameter is NULL, the return value is also NULL.

Behavior Type

Immutable

Syntax

```
BIT_XOR ( expression )
```

Parameters

expression	The [BINARY |VARBINARY] input value to be evaluated. BIT_XOR() operates on VARBINARY types explicitly and on BINARY types implicitly through ***casts*** (page 115).

Notes

- The function returns the same value as the argument data type.
- For each bit compared, if there are an odd number of arguments with set bits, the function returns 1; otherwise it returns 0.
- If the columns are different lengths, the return values are treated as though they are all equal in length and are right-extended with zero bytes. For example, given a group containing the hex values 'ff', null, and 'f', the function ignores the null value and extends the value 'f' to 'f0'.

Example

First create a sample table and projections with binary columns:

This examples uses the following schema, which creates table t with a single column of VARBINARY data type:

```
=> CREATE TABLE t (
      c VARBINARY(2) );
=> INSERT INTO t values(HEX_TO_BINARY('0xFF00'));
=> INSERT INTO t values(HEX_TO_BINARY('0xFFFF'));
=> INSERT INTO t values(HEX_TO_BINARY('0xF00F'));
```

Query table t to see column c output:

```
=> SELECT TO_HEX(c) FROM t;
 TO_HEX
--------
 ff00
 ffff
 f00f
(3 rows)
```

Query table t to get the XOR value for column c:

```
SELECT TO_HEX(BIT_XOR(c)) FROM t;
 TO_HEX
--------
 f0f0
(1 row)
```

See Also

Binary Data Types (page 72)

CORR

Returns the coefficient of correlation of a set of expression pairs (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.Syntax

```
SELECT CORR (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT CORR (Annual_salary, Employee_age) FROM employee_dimension;
         CORR
----------------------
 -0.00719153413192422
(1 row)
```

COUNT [Aggregate]

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`.

Behavior Type

Immutable

Syntax

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

Parameters

*	Indicates that the count does not apply to any specific column or expression in the select list. Requires a `FROM` ***clause*** (page 876).
`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	Returns the number of rows in each group for which the *expression* is not null. Can be any expression resulting in BIGINT.

Notes

The COUNT() aggregate function is different from the COUNT() analytic function, which returns the number over a group of rows within a window.

Examples

The following query returns the number of distinct values in the primary_key column of the date_dimension table:

```
=> SELECT COUNT (DISTINCT date_key) FROM date_dimension;
 count
-------
  1826
(1 row)
```

The next example returns all distinct values of evaluating the expression x+y for all records of fact.

```
=> SELECT COUNT (DISTINCT date_key + product_key) FROM inventory_fact;
 count
-------
 21560
(1 row)
```

An equivalent query is as follows (using the LIMIT key to restrict the number of rows returned):

```
=> SELECT COUNT(date_key + product_key) FROM inventory_fact
   GROUP BY date_key LIMIT 10;
 count
-------
   173
    31
   321
   113
   286
    84
   244
   238
   145
   202
(10 rows)
```

Each distinct product_key value in table inventory_fact and returns the number of distinct values of date_key in all records with the specific distinct product_key value.

```
=> SELECT product_key, COUNT (DISTINCT date_key) FROM  inventory_fact
   GROUP BY product_key LIMIT 10;
 product_key | count
-------------+-------
```

```
           1 |    12
           2 |    18
           3 |    13
           4 |    17
           5 |    11
           6 |    14
           7 |    13
           8 |    17
           9 |    15
          10 |    12
(10 rows)
```

This query counts each distinct `product_key` value in table `inventory_fact` with the constant "1".

```
=> SELECT product_key, COUNT (DISTINCT product_key) FROM  inventory_fact
  GROUP BY product_key LIMIT 10;
product_key | count
------------+-------
           1 |     1
           2 |     1
           3 |     1
           4 |     1
           5 |     1
           6 |     1
           7 |     1
           8 |     1
           9 |     1
          10 |     1
(10 rows)
```

This query selects each distinct `date_key` value and counts the number of distinct `product_key` values for all records with the specific `product_key` value. It then sums the `qty_in_stock` values in all records with the specific `product_key` value and groups the results by `date_key`.

```
=> SELECT date_key, COUNT (DISTINCT product_key), SUM(qty_in_stock) FROM
inventory_fact
   GROUP BY date_key LIMIT 10;
 date_key | count |  sum
----------+-------+--------
        1 |   173 |  88953
        2 |    31 |  16315
        3 |   318 | 156003
        4 |   113 |  53341
        5 |   285 | 148380
        6 |    84 |  42421
        7 |   241 | 119315
        8 |   238 | 122380
        9 |   142 |  70151
       10 |   202 |  95274
(10 rows)
```

This query selects each distinct `product_key` value and then counts the number of distinct `date_key` values for all records with the specific `product_key` value and counts the number of distinct `warehouse_key` values in all records with the specific `product_key` value.

```
=> SELECT product_key, COUNT (DISTINCT date_key), COUNT (DISTINCT warehouse_key)
   FROM inventory_fact GROUP BY product_key LIMIT 15;
 product_key | count | count
-------------+-------+-------
           1 |    12 |    12
           2 |    18 |    18
           3 |    13 |    12
           4 |    17 |    18
           5 |    11 |     9
           6 |    14 |    13
           7 |    13 |    13
           8 |    17 |    15
           9 |    15 |    14
          10 |    12 |    12
          11 |    11 |    11
          12 |    13 |    12
          13 |     9 |     7
          14 |    13 |    13
          15 |    18 |    17
(15 rows)
```

This query selects each distinct `product_key` value, counts the number of distinct `date_key` and `warehouse_key` values for all records with the specific `product_key` value, and then sums all `qty_in_stock` values in records with the specific `product_key` value. It then returns the number of `product_version` values in records with the specific `product_key` value.

```
=> SELECT product_key, COUNT (DISTINCT date_key), COUNT (DISTINCT warehouse_key),
   SUM (qty_in_stock), COUNT (product_version)
   FROM inventory_fact GROUP BY product_key LIMIT 15;
 product_key | count | count |  sum  | count
-------------+-------+-------+-------+-------
           1 |    12 |    12 |  5530 |    12
           2 |    18 |    18 |  9605 |    18
           3 |    13 |    12 |  8404 |    13
           4 |    17 |    18 | 10006 |    18
           5 |    11 |     9 |  4794 |    11
           6 |    14 |    13 |  7359 |    14
           7 |    13 |    13 |  7828 |    13
           8 |    17 |    15 |  9074 |    17
           9 |    15 |    14 |  7032 |    15
          10 |    12 |    12 |  5359 |    12
          11 |    11 |    11 |  6049 |    11
          12 |    13 |    12 |  6075 |    13
          13 |     9 |     7 |  3470 |     9
          14 |    13 |    13 |  5125 |    13
          15 |    18 |    17 |  9277 |    18
(15 rows)
```

The following example returns the number of warehouses from the `warehouse` dimension table:

```
=> SELECT COUNT(warehouse_name) FROM warehouse_dimension;
 count
```

```
-------
   100
(1 row)
```

The next example returns the total number of vendors:

```
=> SELECT COUNT(*) FROM vendor_dimension;
 count
-------
    50
(1 row)
```

See Also

Analytic Functions (page 141)

AVG (page 118)

SUM (page 137)

Using SQL Analytics in the Programmer's Guide

COVAR_POP

Returns the population covariance for a set of expression pairs (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
SELECT COVAR_POP (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT COVAR_POP (Annual_salary, Employee_age) FROM employee_dimension;
     COVAR_POP
-------------------
 -9032.34810730019
(1 row)
```

COVAR_SAMP

Returns the sample covariance for a set of expression pairs (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
COVAR_SAMP (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT COVAR_SAMP (Annual_salary, Employee_age) FROM employee_dimension;
    COVAR_SAMP
------------------
 -9033.25143244343
(1 row)
```

MAX [Aggregate]

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

Behavior Type

Immutable

Syntax

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

Parameters

ALL | DISTINCT	Are meaningless in this context.
expression	Can be any expression for which the maximum value is calculated, typically a ***column reference*** (see "***Column References***" on page 54).

Notes

The `MAX()` aggregate function is different from the `MAX()` analytic function, which returns the maximum value of an expression over a group of rows within a window.

Example

This example returns the largest value (dollar amount) of the `sales_dollar_amount` column.

```
=> SELECT MAX(sales_dollar_amount) AS highest_sale FROM store.store_sales_fact;
 highest_sale
--------------
         600
(1 row)
```

See Also

Analytic Functions (page 141)

MIN (page 129)

MIN [Aggregate]

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

Behavior Type

Immutable

Syntax

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

Parameters

`ALL | DISTINCT`	Are meaningless in this context.
`expression`	Can be any expression for which the minimum value is calculated, typically a ***column reference*** (see "***Column References***" on page 54).

Notes

The `MIN()` aggregate function is different from the `MIN()` analytic function, which returns the minimum value of an expression over a group of rows within a window.

Example

This example returns the lowest salary from the `employee` dimension table.

```
=> SELECT MIN(annual_salary) AS lowest_paid FROM employee_dimension;
 lowest_paid
-------------
        1200
(1 row)
```

See Also

Analytic Functions (page 141)

MAX (page 128)

Using SQL Analytics in the Programmer's Guide

REGR_AVGX

Returns the average of the independent expression in an expression pair (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
SELECT REGR_AVGX (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=>=> SELECT REGR_AVGX (Annual_salary, Employee_age) FROM employee_dimension;
 REGR_AVGX
-----------
    39.321
(1 row)
```

REGR_AVGY

Returns the average of the dependent expression in an expression pair (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
REGR_AVGY (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT REGR_AVGY (Annual_salary, Employee_age) FROM employee_dimension;
 REGR_AVGY
------------
 58354.4913
(1 row)

(1 row)
```

REGR_COUNT

Returns the number of expression pairs (*expression1* and *expression2*). The return value is of type INTEGER. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns 0.

Syntax

```
SELECT REGR_COUNT (expression1, expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT REGR_COUNT (Annual_salary, Employee_age) FROM employee_dimension;
 REGR_COUNT
------------
      10000
(1 row)
```

REGR_INTERCEPT

Returns the y-intercept of the regression line determined by a set of expression pairs (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
SELECT REGR_INTERCEPT (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
b=> SELECT REGR_INTERCEPT (Annual_salary, Employee_age) FROM employee_dimension;
  REGR_INTERCEPT
------------------
 59929.5490163437
(1 row)
```

REGR_R2

Returns the square of the correlation coefficient of a set of expression pairs (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
SELECT REGR_R2 (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT REGR_R2 (Annual_salary, Employee_age) FROM employee_dimension;
       REGR_R2
----------------------
 5.17181631706311e-05
(1 row)
```

REGR_SLOPE

Returns the slope of the regression line, determined by a set of expression pairs (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
SELECT REGR_SLOPE (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT REGR_SLOPE (Annual_salary, Employee_age) FROM employee_dimension;
    REGR_SLOPE
------------------
 -40.056400303749
(1 row)
```

REGR_SXX

Returns the sum of squares of the independent expression in an expression pair (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.,

Syntax

```
SELECT REGR_SXX (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT REGR_SXX (Annual_salary, Employee_age) FROM employee_dimension;
  REGR_SXX
------------
 2254907.59
(1 row)
```

REGR_SXY

Returns the sum of products of the independent expression multiplied by the dependent expression in an expression pair (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
SELECT REGR_SXY (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT REGR_SXY (Annual_salary, Employee_age) FROM employee_dimension;
     REGR_SXY
-------------------
 -90323481.0730019
(1 row)
```

REGR_SYY

Returns the sum of squares of the dependent expression in an expression pair (*expression1* and *expression2*). The return value is of type DOUBLE PRECISION. The function eliminates expression pairs where either expression in the pair is NULL. If no rows remain, the function returns NULL.

Syntax

```
SELECT REGR_SYY (expression1,expression2)
```

Parameters

expression1	The dependent expression. Is of type DOUBLE PRECISION.
expression2	The independent expression. Is of type DOUBLE PRECISION.

Example

```
=> SELECT REGR_SYY (Annual_salary, Employee_age) FROM employee_dimension;
     REGR_SYY
------------------
 69956728794707.2
(1 row)
```

STDDEV [Aggregate]

Note: The non-standard function `STDDEV()` is provided for compatibility with other databases. It is semantically identical to `STDDEV_SAMP()` (page 136).

Evaluates the statistical sample standard deviation for each member of the group. The `STDDEV()` return value is the same as the square root of the `VAR_SAMP()` function:

```
STDDEV(expression) = SQRT(VAR_SAMP(expression))
```

When `VAR_SAMP()` returns null, this function returns null.

Behavior Type

Immutable

Syntax

```
STDDEV ( expression )
```

Parameters

expression	Any `NUMERIC` ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Notes

The `STDDEV()` aggregate function is different from the `STDDEV()` analytic function, which computes the statistical sample standard deviation of the current row with respect to the group of rows within a window.

Examples

The following example returns the statistical sample standard deviation for each household ID from the `customer_dimension` table of the VMart example database:

```
=> SELECT STDDEV(household_id) FROM customer_dimension;
   STDDEV
-----------------
 8651.5084240071
```

See Also

Analytic Functions (page 141)

STDDEV_SAMP (page 136)

Using SQL Analytics in the Programmer's Guide

STDDEV_POP [Aggregate]

Evaluates the statistical population standard deviation for each member of the group. The `STDDEV_POP()` return value is the same as the square root of the `VAR_POP()` function

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

When `VAR_SAMP()` returns null, this function returns null.

Behavior Type

Immutable

Syntax

```
STDDEV_POP ( expression )
```

Parameters

expression	Any `NUMERIC` ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Notes

The STDDEV_POP() aggregate function is different from the STDDEV_POP() analytic function, which evaluates the statistical population standard deviation for each member of the group of rows within a window.

Examples

The following example returns the statistical population standard deviation for each household ID in the customer table.

```
=> SELECT STDDEV_POP(household_id) FROM customer_dimension;
   stddev_samp
------------------
 8651.41895973367
(1 row)
```

See Also

Analytic Functions (page 141)

Using SQL for Analytics in the Programmer's Guide

STDDEV_SAMP [Aggregate]

Evaluates the statistical sample standard deviation for each member of the group. The STDDEV_SAMP() return value is the same as the square root of the VAR_SAMP() function:

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

When VAR_SAMP() returns null, this function returns null.

Behavior Type:

Immutable

Syntax

```
STDDEV_SAMP ( expression )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Notes

- STDDEV_SAMP() is semantically identical to the non-standard function, STDDEV() (page 134), which is provided for compatibility with other databases.

- The `STDDEV_SAMP()` aggregate function is different from the `STDDEV_SAMP()` analytic function, which computes the statistical sample standard deviation of the current row with respect to the group of rows within a window.

Examples

The following example returns the statistical sample standard deviation for each household ID from the `customer` dimension table.

```
=> SELECT STDDEV_SAMP(household_id) FROM customer_dimension;
   stddev_samp
------------------
 8651.50842400771
(1 row)
```

See Also

Analytic Functions (page 141)

STDDEV (page 134)

Using SQL Analytics in the Programmer's Guide

SUM [Aggregate]

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.

Behavior Type

Immutable

Syntax

`SUM ( [ ALL | DISTINCT ] expression )`

Parameters

`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 `NUMERIC` ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Notes

- The SUM() aggregate function is different from the SUM() analytic function, which computes the sum of an expression over a group of rows within a window.
- If you encounter data overflow when using SUM(), use SUM_FLOAT() (page 138) which converts the data to a floating point.

Example

This example returns the total sum of the product_cost column.

```
=> SELECT SUM(product_cost) AS cost FROM product_dimension;
  cost
---------
 9042850
(1 row)
```

See Also

AVG (page 118)

COUNT (page 123)

Numeric Data Types (page 103)

Using SQL Analytics in the Programmer's Guide

SUM_FLOAT [Aggregate]

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.

Behavior Type

Immutable

Syntax

SUM_FLOAT ([ALL | DISTINCT] *expression*)

Parameters

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	Can be any expression resulting in DOUBLE PRECISION.

Example

The following example returns the floating point sum of the average price from the product table:

```
=> SELECT SUM_FLOAT(average_competitor_price) AS cost FROM product_dimension;
   cost
----------
 18181102
(1 row)
```

VAR_POP [Aggregate]

Evaluates the 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.

```
(SUM(expression*expression) - SUM(expression)*SUM(expression) /
COUNT(expression)) / COUNT(expression)
```

Behavior Type

Immutable

Syntax

```
VAR_POP ( expression )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Notes

The VAR_POP() aggregate function is different from the VAR_POP() analytic function, which computes the population variance of the current row with respect to the group of rows within a window.

Examples

The following example returns the population variance for each household ID in the customer table.

```
=> SELECT VAR_POP(household_id) FROM customer_dimension;
     var_pop
------------------
 74847050.0168393
(1 row)
```

VAR_SAMP [Aggregate]

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).

```
(SUM(expression*expression) - SUM(expression) *SUM(expression) /
COUNT(expression)) / (COUNT(expression) -1)
```

Behavior Type

Immutable

Syntax

```
VAR_SAMP ( expression )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Notes

- `VAR_SAMP()` is semantically identical to the non-standard function, `VARIANCE` (page 140)`()`, which is provided for compatibility with other databases.
- The `VAR_SAMP()` aggregate function is different from the `VAR_SAMP()` analytic function, which computes the sample variance of the current row with respect to the group of rows within a window.

Examples

The following example returns the sample variance for each household ID in the `customer` table.

```
=> SELECT VAR_SAMP(household_id) FROM customer_dimension;
     var_samp
------------------
 74848598.0106764
(1 row)
```

See Also

Analytic Functions (page 141)

VARIANCE (page 140)

Using SQL Analytics in the Programmer's Guide

VARIANCE [Aggregate]

Note: The non-standard function `VARIANCE()` is provided for compatibility with other databases. It is semantically identical to `VAR_SAMP()` (page 139).

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).

```
(SUM(expression*expression) - SUM(expression) *SUM(expression) /
```

```
COUNT(expression)) / (COUNT(expression) -1)
```

Behavior Type

Immutable

Syntax

```
VARIANCE ( expression )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Notes

The `VARIANCE()` aggregate function is different from the `VARIANCE()` analytic function, which computes the sample variance of the current row with respect to the group of rows within a window.

Examples

The following example returns the sample variance for each household ID in the `customer` table.

```
=> SELECT VARIANCE(household_id) FROM customer_dimension;
     variance
------------------
 74848598.0106764
(1 row)
```

See Also

Analytic Functions (page 141)

VAR_SAMP (page 139)

Using SQL Analytics in the Programmer's Guide

Analytic Functions

Note: All analytic functions in this section that have an aggregate counterpart are appended with [Analytics] in the heading to avoid confusion between the two.

HP Vertica analytics are SQL functions based on the ANSI 99 standard. These functions handle complex analysis and reporting tasks such as:

- Rank the longest-standing customers in a particular state
- Calculate the moving average of retail volume over a specified time
- Find the highest score among all students in the same grade

- Compare the current sales bonus each salesperson received against his or her previous bonus

Analytic functions return aggregate results but they do not group the result set. They return the group value multiple times, once per record.

You can sort these group values, or partitions, using a window `ORDER BY` clause, but the order affects only the function result set, not the entire query result set. This ordering concept is described more fully later.

Analytic function syntax

```
ANALYTIC_FUNCTION( argument-1, ..., argument-n )
  OVER( [ window_partition_clause (on page 143) ]
        [ window_order_clause (on page 144) ]
        [ window_frame_clause (on page 145) ] )
```

Analytic syntactic construct

`ANALYTIC_FUNCTION()`	HP Vertica provides a number of analytic functions that allow advanced data manipulation and analysis. Each of these functions takes one or more arguments.
`OVER(...)`	Specifies partitioning, ordering, and window framing for the function—important elements that determine what data the analytic function takes as input with respect to the current row. The `OVER()` clause is evaluated after the `FROM`, `WHERE`, `GROUP BY`, and `HAVING` clauses. The SQL `OVER()` clause must follow the analytic function.
`window_partition_clause`	Groups the rows in the input table by a given list of columns or expressions. The `window_partition_clause` is optional; if you omit it, the rows are not grouped, and the analytic function applies to all rows in the input set as a single partition. See `window_partition_clause` (page 143).
`window_order_clause`	Sorts the rows specified by the OVER() operator and supplies the ordered set of rows to the analytic function. If the partition clause is present, the `window_order_clause` applies within each partition. The order clause is optional. If you do not use it, the selection set is not sorted. See `window_order_clause` (page 144).
`window_frame_clause`	Used by only some analytic functions. If you include the frame clause in the OVER() statement, which specifies the beginning and end of the window relative to the current row, the analytic function applies only to a subset of the rows defined by the partition clause. This subset changes as the rows in the partition change (called a moving window). See `window_frame_clause`. (page 145)

Notes

Analytic functions:

- Require the OVER() clause. However, depending on the function, the window_frame_clause and window_order_clause might not apply. For example, when used with analytic aggregate functions like SUM(*x*), you can use the OVER() clause without supplying any of the windowing clauses; in this case, the aggregate returns the same aggregated value for each row of the result set.
- Are allowed only in the SELECT and ORDER BY clauses.
- Can be used in a subquery or in the parent query but *cannot* be nested; for example, the following query is not allowed:
  `=> SELECT MEDIAN(RANK() OVER(ORDER BY sal) OVER()).`
- WHERE, GROUP BY and HAVING operators are technically not part of the analytic function; however, they determine on which rows the analytic functions operate.

See Also

Using SQL Analytics and Sort Optimizations in the Programmer's Guide

window_partition_clause

Window partitioning is optional. When specified, the window_partition_clause divides the rows in the input based on user-provided expressions, such as aggregation functions like SUM(*x*). Window partitioning is similar to the GROUP BY clause except that it returns only one result row per input row. If you omit the window_partition_clause, all input rows are treated as a single partition.

The analytic function is computed per partition and starts over again (resets) at the beginning of each subsequent partition. The `window_partition_clause i`s specified within the `OVER()` clause.

Syntax

```
OVER ( PARTITION BY expression [ , ... ] )
```

Parameters

expression	Expression on which to to sort the partition on. May involve columns, constants or an arbitrary expression formed on columns.

For examples, see Window Partitioning in the Programmer's Guide.

window_order_clause

Sorts the rows specified by the OVER() clause and specifies whether data is sorted in ascending or descending order as well as the placement of null values; for example: ORDER BY expr_list [ASC | DESC] [NULLS { FIRST | LAST | AUTO]. The ordering of the data affects the results.

Using ORDER BY in an OVER clause changes the default window to RANGE UNBOUNDED PRECEDING AND CURRENT ROW, which is described in the ***window_frame_clause*** (page 145).

The following table shows the default null placement, with bold clauses to indicate what is implicit:

Ordering	Null placement
ORDER BY column1	ORDER BY a **ASC NULLS LAST**
ORDER BY column1 ASC	ORDER BY a **ASC NULLS LAST**
ORDER BY column1 DESC	ORDER BY a **DESC NULLS FIRST**

Because the window_order_clause is different from a query's final ORDER BY clause, window ordering might not guarantee the final result order; it specifies only the order within a window result set, supplying the ordered set of rows to the window_frame_clause (if present), to the analytic function, or to both. Use the ***SQL ORDER BY clause*** (page 893) to guarantee ordering of the final result set.

Syntax

```
OVER ( ORDER BY expression  [ { ASC | DESC } ]
...  [ NULLS { FIRST | LAST | AUTO } ] [, expression ...] )
```

Parameters

expression	Expression on which to sort the partition, which may involve columns, constants, or an arbitrary expression formed on columns.
ASC | DESC	Specifies the ordering sequence as ascending (default) or descending.
NULLS { FIRST | LAST | AUTO }	Indicates the position of nulls in the ordered sequence as either first or last. The order makes nulls compare either high or low with respect to non-null values. If the sequence is specified as ascending order, ASC NULLS FIRST implies that nulls are smaller than other non-null values. ASC NULLS LAST implies that nulls are larger than non-null values. The opposite is true for descending order. If you specify NULLS AUTO, HP Vertica chooses the most efficient placement of nulls (for example, either NULLS FIRST or NULLS LAST) based on your query. The default is ASC NULLS LAST

	and `DESC NULLS FIRST`. See also Analytics Null Placement and Minimizing Sort Operations in the Programmer's Guide.

The following analytic functions require the window_order_clause:

- ***RANK()*** (page 182) / ***DENSE_RANK()*** (page 156)
- ***LEAD()*** (page 168) / ***LAG()*** (page 163)
- ***PERCENT_RANK()*** (page 176) / ***CUME_DIST()*** (page 155)
- ***NTILE()*** (page 175)

You can also use the window_order_clause with aggregation functions, such as SUM(x).

The ORDER BY clause is optional for the ***ROW_NUMBER()*** (page 184) function.

The ORDER BY clause is not allowed with the following functions:

- ***PERCENTILE_CONT()*** (page 178) / ***PERCENTILE_DISC()*** (page 180)
- ***MEDIAN()*** (page 172)

For examples, see Window Ordering in the Programmer's Guide.

window_frame_clause

Allowed for some analytic functions in the analytic OVER() clause, window framing represents a unique construct, called a moving window. It defines which values in the partition are evaluated relative to the current row. You specify a window frame in terms of either logical intervals (such as time) using the RANGE keyword or on a physical number of rows before and/or after the current row using the ROWS keyword. The CURRENT ROW is the next row for which the analytic function computes results.

As the current row advances, the window boundaries are recomputed (move) along with it, determining which rows fall into the current window.

An analytic function with a window frame specification is computed for each row based on the rows that fall into the window relative to that row.

An analytic function with a window frame specification is computed for each row based on the rows that fall into the window relative to that row. If you omit the `window_frame_clause`, the default window is `RANGE UNBOUNDED PRECEDING AND CURRENT ROW`.

Syntax

```
{ ROWS | RANGE }
{
  {
      BETWEEN
      { UNBOUNDED PRECEDING
      | CURRENT ROW
      | constant-value { PRECEDING | FOLLOWING }
      }
      AND
      { UNBOUNDED FOLLOWING
      | CURRENT ROW
      | constant-value  { PRECEDING | FOLLOWING }
      }
  }
|
  {
      { UNBOUNDED PRECEDING
      | CURRENT ROW
      | constant-value  PRECEDING
      }
  }
}
```

Parameters

ROWS | RANGE	The ROWS and RANGE keywords define the window frame type. ROWS specifies a window as a physical offset and defines the window's start and end point by the number of rows before or after the current row. The value can be INTEGER data type only. RANGE specifies the window as a logical offset, such as time. The range value must match the window_order_clause data type, which can be NUMERIC, DATE/TIME, FLOAT or INTEGER. **Note**: The value returned by an analytic function with a logical offset is always deterministic. However, the value returned by an analytic function with a physical offset could produce nondeterministic results unless the ordering expression results in a unique ordering. You might have to specify multiple columns in the window_order_clause (on page 144) to achieve this unique ordering.
BETWEEN ... AND	Specifies a start point and end point for the window. The first expression (before AND) defines the start point and the second expression (after AND) defines the end point. **Note**: If you use the keyword BETWEEN, you must also use AND.
UNBOUNDED PRECEDING	Within a partition, indicates that the window frame starts at the first row of the partition. This start-point specification cannot be used as an end-point specification, and the default is RANGE UNBOUNDED PRECEDING AND CURRENT ROW
UNBOUNDED FOLLOWING	Within a partition, indicates that the window frame ends at the last

	row of the partition. This end-point specification cannot be used as a start-point specification.
CURRENT ROW	As a start point, CURRENT ROW specifies that the window begins at the current row or value, depending on whether you have specified ROW or RANGE, respectively. In this case, the end point cannot be *constant-value* PRECEDING. As an end point, CURRENT ROW specifies that the window ends at the current row or value, depending on whether you have specified ROW or RANGE, respectively. In this case the start point cannot be *constant-value* FOLLOWING.
constant-value { PRECEDING | FOLLOWING }	For RANGE or ROW: ▪ If *constant-value* FOLLOWING is the start point, the end point must be *constant-value* FOLLOWING. ▪ If *constant-value* PRECEDING is the end point, the start point must be *constant-value* PRECEDING. ▪ If you specify a logical window that is defined by a time interval in NUMERIC format, you might need to use conversion functions. If you specified ROWS: ▪ *constant-value* is a physical offset. It must be a constant or expression and must evaluate to an INTEGER data type value. ▪ If *constant-value* is part of the start point, it must evaluate to a row before the end point. If you specified RANGE: ▪ *constant-value* is a logical offset. It must be a constant or expression that evaluates to a positive numeric value or an INTERVAL literal. ▪ If *constant-value* evaluates to a NUMERIC value, the ORDER BY column type must be a NUMERIC data type.. ▪ If the *constant-value* evaluates to an INTERVAL DAY TO SECOND subtype, the ORDER BY column type can only be TIMESTAMP, TIME, DATE, or INTERVAL DAY TO SECOND. ▪ If the *constant-value* evaluates to an INTERVAL YEAR TO MONTH, the ORDER BY column type can only be TIMESTAMP, DATE, or INTERVAL YEAR TO MONTH. ▪ You can specify only one expression in the window_order_clause.

Window Aggregates

Analytic functions that take the window_frame_clause are called window aggregates, and they return information such as moving averages and cumulative results. To use the following functions as window (analytic) aggregates, instead of basic aggregates, specify both an ORDER BY clause (window_order_clause) and a moving window (window_frame_clause) in the OVER() clause.

Within a partition, UNBOUNDED PRECEDING/FOLLOWING means beginning/end of partition. If you omit the `window_frame_clause` but you specify the `window_order_clause`, the system provides the default window of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

The following analytic functions take the window_frame_clause:

- ***AVG()*** (page 150)
- ***COUNT()*** (page 153)
- ***MAX()*** (page 171) and ***MIN()*** (page 174)
- ***SUM()*** (page 189)
- ***STDDEV()*** (page 186), ***STDDEV_POP()*** (page 186), and ***STDDEV_SAMP()*** (page 186)
- ***VARIANCE()*** (page 193), ***VAR_POP()*** (page 191), and ***VAR_SAMP()*** (page 192)

Note: ***FIRST_VALUE*** (page 160) and ***LAST_VALUE*** (page 166) functions also accept the window_frame_clause, but they are analytic functions only and have no aggregate counterpart. ***EXPONENTIAL_MOVING_AVERAGE*** (page 158), ***LAG*** (page 163), and ***LEAD*** (page 168) analytic functions do not take the window_frame_clause.

If you use a window aggregate with an empty OVER() clause, there is no moving window, and the function is used as a reporting function, where the entire input is treated as one partition.

The value returned by an analytic function with a *logical* offset is always deterministic. However, the value returned by an analytic function with a *physical* offset could produce nondeterministic results unless the ordering expression results in a unique ordering. You might have to specify multiple columns in the `window_order_clause` to achieve this unique ordering.

See Window Framing in the Programmer's Guide for examples.

named_windows

You can use the WINDOW clause to name your windows and avoid typing long `OVER()` clause syntax.

The `window_partition_clause` is defined in the named window specification, not in the `OVER()` clause, and a window definition cannot contain a `window_frame_clause`.

Each window defined in the `window_definition_clause` must have a unique name.

Syntax

```
WINDOW window_name AS ( window_definition_clause );
```

Parameters

window_name	User-supplied name of the analytics window.

window_definition_clause	[window_partition_clause (on page 143)] Groups the rows in the input table by a given list of columns or expressions. The window_partition_clause is optional; if you omit it, the rows are not grouped, and the analytic function applies to all rows in the input set as a single partition. See window_partition_clause (page 143). [window_order_clause (on page 144)] Sorts the rows specified by the window_partition_clause and supplies an ordered set of rows to the window_frame_clause (if present), to the analytic function, or to both. The window_order_clause specifies whether data is returned in ascending or descending order and specifies where null values appear in the sorted result as either first or last. The ordering of the data affects the results. Note: The window_order_clause does not guarantee the order of the SQL result. Use the ***SQL ORDER BY clause*** (page 893) to guarantee the ordering of the final result set.

Examples

In the following example, RANK() and DENSE_RANK() use the partitioning and ordering specifications in the window definition for a window named w:

```
=> SELECT RANK() OVER w , DENSE_RANK() OVER w
   FROM employee_dimension
   WINDOW w AS (PARTITION BY employee_region ORDER by annual_salary);
```

Though analytic functions can reference a named window to inherit the window_partition_clause (page 143), you can use OVER() to define your own window_order_clause (page 144), but only if the window_definition_clause did not already define it. Because ORDER by annual_salary was already defined in the WINDOW clause in the previous example, the following query would return an error.

```
=> SELECT RANK() OVER(w ORDER BY annual_salary ASC),
       DENSE_RANK() OVER(w ORDER BY annual_salary DESC)
   FROM employee_dimension
   WINDOW w AS (PARTITION BY employee_region);
   ERROR:  cannot override ORDER BY clause of window "w"
```

You can reference window names within their scope only. For example, because named window w1 in the following query is defined before w2, w2 is within the scope of w1:

```
=> SELECT RANK() OVER(w1 ORDER BY sal DESC), RANK() OVER w2
   FROM EMP
   WINDOW w1 AS (PARTITION BY deptno), w2 AS (w1 ORDER BY sal);
```

AVG [Analytic]

Computes an average of an expression in a group within a window.

Behavior Type

Immutable

Syntax

```
AVG ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	The value whose average is calculated over a set of rows. Can be any expression resulting in `DOUBLE PRECISION`.
`OVER(...)`	See ***Analytic Functions***. (page 141)

Notes

`AVG()` takes as an argument any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the argument's numeric data type.

Examples

The following query finds the sales for that calendar month and returns a running/cumulative average (sometimes called a moving average) using the default window of `RANGE UNBOUNDED PRECEDING AND CURRENT ROW`:

```
=> SELECT calendar_month_number_in_year, SUM(product_price) AS sales,
   AVG(SUM(product_price)) OVER (ORDER BY calendar_month_number_in_year)
   FROM product_dimension, date_dimension, inventory_fact
   WHERE date_dimension.date_key = inventory_fact.date_key
   AND product_dimension.product_key = inventory_fact.product_key
   GROUP BY calendar_month_number_in_year;

 calendar_month_number_in_year |  sales   |     ?column?
-------------------------------+----------+------------------
                             1 | 23869547 |         23869547
                             2 | 19604661 |         21737104
                             3 | 22877913 | 22117373.6666667
                             4 | 22901263 |         22313346
                             5 | 23670676 |         22584812
                             6 | 22507600 | 22571943.3333333
                             7 | 21514089 | 22420821.2857143
                             8 | 24860684 |     22725804.125
                             9 | 21687795 | 22610469.7777778
                            10 | 23648921 |       22714314.9
```

```
                          11 | 21115910 | 22569005.3636364
                          12 | 24708317 | 22747281.3333333
(12 rows)
```

To return a moving average that is not a running (cumulative) average, the window should specify `ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING`:

```
=> SELECT calendar_month_number_in_year, SUM(product_price) AS sales,
   AVG(SUM(product_price)) OVER (ORDER BY calendar_month_number_in_year
     ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)
   FROM product_dimension, date_dimension, inventory_fact
   WHERE date_dimension.date_key = inventory_fact.date_key
   AND product_dimension.product_key = inventory_fact.product_key
   GROUP BY calendar_month_number_in_year;
```

See Also

AVG (page 118) aggregate function

COUNT (page 153) and ***SUM*** (page 189) analytic functions

Using SQL Analytics in the Programmer's Guide

CONDITIONAL_CHANGE_EVENT [Analytic]

Assigns an event window number to each row, starting from 0, and increments by 1 when the result of evaluating the argument expression on the current row differs from that on the previous row.

Behavior Type

Immutable

Syntax

```
CONDITIONAL_CHANGE_EVENT ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

expression	Is a SQL scalar expression that is evaluated on an input record. The result of *expression* can be of any data type.
`OVER(...)`	See ***Analytic Functions***. (page 141)

Notes

The analytic `window_order_clause` is required but the `window_partition_clause` is optional.

Example

```
=> SELECT CONDITIONAL_CHANGE_EVENT(bid)
```

```
   OVER (PARTITION BY symbol ORDER BY ts) AS cce
   FROM TickStore;
```

The system returns an error when no `ORDER BY` is present:

```
=> SELECT CONDITIONAL_CHANGE_EVENT(bid)
   OVER (PARTITION BY symbol) AS cce
   FROM TickStore;

   ERROR:  conditional_change_event must contain an ORDER BY clause within
   its analytic clause
```

For more examples, see Event-based Windows in the Programmer's Guide.

See Also

CONDITIONAL_TRUE_EVENT (page 152)

ROW_NUMBER (page 184)

Using Time Series Analytics and Event-based Windows in the Programmer's Guide

CONDITIONAL_TRUE_EVENT [Analytic]

Assigns an event window number to each row, starting from 0, and increments the number by 1 when the result of the boolean argument expression evaluates true. For example, given a sequence of values for column a:

```
( 1, 2, 3, 4, 5, 6 )
```

`CONDITIONAL_TRUE_EVENT(a > 3)` returns `0, 0, 0, 1, 2, 3`.

Behavior Type:

Immutable

Syntax

```
CONDITIONAL_TRUE_EVENT ( boolean-expression ) OVER
... ( [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

boolean-expression	Is a SQL scalar expression that is evaluated on an input record. The result of *boolean-expression* is boolean type.
`OVER(...)`	See ***Analytic Functions*** (page 141).

Notes

The analytic `window_order_clause` is required but the `window_partition_clause` is optional.

Example

```
=> SELECT CONDITIONAL_TRUE_EVENT(bid > 10.6)
     OVER(PARTITION BY bid ORDER BY ts) AS cte
   FROM Tickstore;
```

The system returns an error if the ORDER BY clause is omitted:

```
=> SELECT CONDITIONAL_TRUE_EVENT(bid > 10.6)
     OVER(PARTITION BY bid) AS cte
   FROM Tickstore;
   ERROR:  conditional_true_event must contain an ORDER BY clause within its
   analytic clause
```

For more examples, see Event-based Windows in the Programmer's Guide.

See Also

CONDITIONAL_CHANGE_EVENT (page 151)

Using Time Series Analytics and Event-based Windows in the Programmer's Guide

COUNT [Analytic]

Counts occurrences within a group within a window. If you specify * or some non-null constant, COUNT() counts all rows.

Behavior Type

Immutable

Syntax

```
COUNT ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Returns the number of rows in each group for which the *expression* is not null. Can be any expression resulting in BIGINT.
OVER(...)	See ***Analytic Functions.*** (page 141)

Example

Using the schema defined in Window Framing in the Programmer's Guide, the following COUNT function does not specify an order_clause or a frame_clause; otherwise it would be treated as a window aggregate. Think of the window of reporting aggregates as UNBOUNDED PRECEDING and UNBOUNDED FOLLOWING.

```
=> SELECT deptno, sal, empno, COUNT(sal)
```

```
     OVER (PARTITION BY deptno) AS count FROM emp;

 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     2
     10 | 104 |     4 |     2
     20 | 110 |    10 |     6
     20 | 110 |     9 |     6
     20 | 109 |     7 |     6
     20 | 109 |     6 |     6
     20 | 109 |     8 |     6
     20 | 109 |    11 |     6
     30 | 105 |     5 |     3
     30 | 103 |     3 |     3
     30 | 102 |     2 |     3
```

Using `ORDER BY sal` creates a moving window query with default window: RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

```
=> SELECT deptno, sal, empno, COUNT(sal)
     OVER (PARTITION BY deptno ORDER BY sal) AS count
   FROM emp;

 deptno | sal | empno | count
--------+-----+-------+-------
     10 | 101 |     1 |     1
     10 | 104 |     4 |     2
     20 | 100 |    11 |     1
     20 | 109 |     7 |     4
     20 | 109 |     6 |     4
     20 | 109 |     8 |     4
     20 | 110 |    10 |     6
     20 | 110 |     9 |     6
     30 | 102 |     2 |     1
     30 | 103 |     3 |     2
     30 | 105 |     5 |     3
```

Using the VMart schema, the following query finds the number of employees who make less than or equivalent to the hourly rate of the current employee. The query returns a running/cumulative average (sometimes called a moving average) using the default window of `RANGE UNBOUNDED PRECEDING AND CURRENT ROW`:

```
=> SELECT employee_last_name AS "last_name", hourly_rate, COUNT(*)
   OVER (ORDER BY hourly_rate) AS moving_count from employee_dimension;

 last_name  | hourly_rate | moving_count
------------+-------------+--------------
 Gauthier   |           6 |            4
 Taylor     |           6 |            4
 Jefferson  |           6 |            4
 Nielson    |           6 |            4
 McNulty    |        6.01 |           11
 Robinson   |        6.01 |           11
```

```
 Dobisz     |        6.01 |           11
 Williams   |        6.01 |           11
 Kramer     |        6.01 |           11
 Miller     |        6.01 |           11
 Wilson     |        6.01 |           11
 Vogel      |        6.02 |           14
 Moore      |        6.02 |           14
 Vogel      |        6.02 |           14
 Carcetti   |        6.03 |           19
...
```

To return a moving average that is not also a running (cumulative) average, the window should specify `ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING`:

```
=> SELECT employee_last_name AS "last_name", hourly_rate, COUNT(*)
      OVER (ORDER BY hourly_rate ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)
   AS moving_count from employee_dimension;
```

See Also

COUNT (page 123) aggregate function

AVG (page 150) and ***SUM*** (page 189) analytic functions

Using SQL Analytics in the Programmer's Guide

CUME_DIST [Analytic]

Calculates the cumulative distribution, or relative rank, of the current row with regard to other rows in the same partition within a window.

`CUME_DIST()` returns a number greater then 0 and less then or equal to 1, where the number represents the relative position of the specified row within a group of *N* rows. For a row `x` (assuming `ASC` ordering), the `CUME_DIST` of `x` is the number of rows with values lower than or equal to the value of `x`, divided by the number of rows in the partition. In a group of three rows, for example, the cumulative distribution values returned would be 1/3, 2/3, and 3/3.

Note: Because the result for a given row depends on the number of rows preceding that row in the same partition, HP recommends that you always specify a `window_order_clause` when you call this function.

Behavior Type

Immutable

Syntax

```
CUME_DIST ( ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

`OVER(...)`	See ***Analytic Functions.*** (page 141)

Notes

The analytic `window_order_clause` is required but the `window_partition_clause` is optional.

Examples

The following example returns the cumulative distribution of sales for different transaction types within each month of the first quarter.

```
=> SELECT calendar_month_name AS month, tender_type, SUM(sales_quantity),
       CUME_DIST()
   OVER (PARTITION BY calendar_month_name ORDER BY SUM(sales_quantity)) AS
CUME_DIST
   FROM store.store_sales_fact JOIN date_dimension
   USING(date_key) WHERE calendar_month_name IN ('January','February','March')
   AND tender_type NOT LIKE 'Other'
   GROUP BY calendar_month_name, tender_type;

  month   | tender_type |  SUM   | CUME_DIST
----------+-------------+--------+-----------
 March    | Credit      | 469858 |      0.25
 March    | Cash        | 470449 |       0.5
 March    | Check       | 473033 |      0.75
 March    | Debit       | 475103 |         1
 January  | Cash        | 441730 |      0.25
 January  | Debit       | 443922 |       0.5
 January  | Check       | 446297 |      0.75
 January  | Credit      | 450994 |         1
 February | Check       | 425665 |      0.25
 February | Debit       | 426726 |       0.5
 February | Credit      | 430010 |      0.75
 February | Cash        | 430767 |         1
(12 rows)
```

See Also

PERCENT_RANK (page 176)

PERCENTILE_DISC (page 180)

Using SQL Analytics in the Programmer's Guide

DENSE_RANK [Analytic]

Computes the relative rank of each row returned from a query with respect to the other rows, based on the values of the expressions in the window `ORDER BY` clause.

The data within a group is sorted by the ORDER BY clause and then a numeric ranking is assigned to each row in turn starting with 1 and continuing from there. The rank is incremented every time the values of the ORDER BY expressions change. Rows with equal values receive the same rank (nulls are considered equal in this comparison). A DENSE_RANK() function returns a ranking number without any gaps, which is why it is called "DENSE."

Behavior Type

Immutable

Syntax

```
DENSE_RANK ( ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- The analytic window_order_clause is required but the window_partition_clause is optional.
- The ranks are consecutive integers beginning with 1. The largest rank value is the number of unique values returned by the query.
- The primary difference between DENSE_RANK() and RANK() (page 182) is that RANK leaves gaps when ranking records whereas DENSE_RANK leaves no gaps. For example, N records occupy a particular position (say, a tie for rank X), RANK assigns all those records with rank X and skips the next N ranks, therefore the next assigned rank is X+N. DENSE_RANK places all the records in that position only—it does not skip any ranks.

  If there is a tie at the third position with two records having the same value, RANK and DENSE_RANK place both the records in the third position, but RANK places the next record at the fifth position, while DENSE_RANK places the next record at the fourth position.
- If you omit NULLS FIRST | LAST | AUTO, the ordering of the NULL values depends on the ASC or DESC arguments. NULL values are considered larger than any other value. If the ordering sequence is ASC, then nulls appear last; nulls appear first otherwise. Nulls are considered equal to other nulls and, therefore, the order in which nulls are presented is non-deterministic.

Examples

The following example shows the difference between RANK and DENSE_RANK when ranking customers by their annual income. Notice that RANK has a tie at 10 and skips 11, while DENSE_RANK leaves no gaps in the ranking sequence:

```
=> SELECT customer_name, SUM(annual_income),
   RANK () OVER (ORDER BY TO_CHAR(SUM(annual_income),'100000') DESC) rank,
```

```
   DENSE_RANK () OVER (ORDER BY TO_CHAR(SUM(annual_income),'100000') DESC)
dense_rank
   FROM customer_dimension GROUP BY customer_name LIMIT 15;

    customer_name    |  sum  | rank | dense_rank
---------------------+-------+------+------------
 Brian M. Garnett    | 99838 |    1 |          1
 Tanya A. Brown      | 99834 |    2 |          2
 Tiffany P. Farmer   | 99826 |    3 |          3
 Jose V. Sanchez     | 99673 |    4 |          4
 Marcus D. Rodriguez | 99631 |    5 |          5
 Alexander T. Nguyen | 99604 |    6 |          6
 Sarah G. Lewis      | 99556 |    7 |          7
 Ruth Q. Vu          | 99542 |    8 |          8
 Theodore T. Farmer  | 99532 |    9 |          9
 Daniel P. Li        | 99497 |   10 |         10
 Seth E. Brown       | 99497 |   10 |         10
 Matt X. Gauthier    | 99402 |   12 |         11
 Rebecca W. Lewis    | 99296 |   13 |         12
 Dean L. Wilson      | 99276 |   14 |         13
 Tiffany A. Smith    | 99257 |   15 |         14
(15 rows)
```

See Also

RANK (page 182)

Using SQL Analytics in the Programmer's Guide

EXPONENTIAL_MOVING_AVERAGE [Analytic]

Calculates the exponential moving average of expression *E* with smoothing factor *X*.

The exponential moving average (EMA) is calculated by adding the previous EMA value to the current data point scaled by the smoothing factor, as in the following formula, where:

- EMA0 is the previous row's EMA value
- *X* is the smoothing factor
- *E* is the current data point: EMA = EMA0 + (*X* * (*E* - EMA0))

EXPONENTIAL_MOVING_AVERAGE() is different from a simple moving average in that it provides a more stable picture of changes to data over time.

Behavior Type

Immutable

Syntax

```
EXPONENTIAL_MOVING_AVERAGE ( E , X ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

E	The value whose average is calculated over a set of rows. Can be INTEGER, FLOAT or NUMERIC type and must be a constant.
X	A positive FLOAT value between 0 and 1 that is used as the smoothing factor.
OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- The analytic window_order_clause is required but the window_partition_clause is optional.
- There is no [Aggregate] equivalent of this function because of its unique semantics.
- The EXPONENTIAL_MOVING_AVERAGE() function also works at the row level; for example, EMA assumes the data in a given column is sampled at uniform intervals. If the users' data points are sampled at non-uniform intervals, they should run the time series gap filling and interpolation (GFI) operations before EMA().

Examples

The following example uses time series gap filling and interpolation (GFI) first in a subquery, and then performs an EXPONENTIAL_MOVING_AVERAGE operation on the subquery result.

Create a simple 4-column table:

```
=> CREATE TABLE ticker(
     time TIMESTAMP,
     symbol VARCHAR(8),
     bid1 FLOAT,
     bid2 FLOAT );
```

Now insert some data, including nulls, so GFI can do its interpolation and gap filling:

```
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:00', 'ABC', 60.45, 60.44);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:01', 'ABC', 60.49, 65.12);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:02', 'ABC', 57.78, 59.25);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:03', 'ABC', null, 65.12);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:04', 'ABC', 67.88, null);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:00', 'XYZ', 47.55, 40.15);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:01', 'XYZ', 44.35, 46.78);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:02', 'XYZ', 71.56, 75.78);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:03', 'XYZ', 85.55, 70.21);
=> INSERT INTO ticker VALUES ('2009-07-12 03:00:04', 'XYZ', 45.55, 58.65);
=> COMMIT;
```

Note: During gap filling and interpolation, HP Vertica takes the closest non null value on either side of the time slice and uses that value. For example, if you use a linear interpolation scheme and you do not specify IGNORE NULLS, and your data has one real value and one null, the result is null. If the value on either side is null, the result is null. See When Time Series Data Contains Nulls in the Programmer's Guide for details.

Query the table you just created to you can see the output:

```
=> SELECT * FROM ticker;
        time         | symbol | bid1  | bid2
---------------------+--------+-------+-------
 2009-07-12 03:00:00 | ABC    | 60.45 | 60.44
 2009-07-12 03:00:01 | ABC    | 60.49 | 65.12
 2009-07-12 03:00:02 | ABC    | 57.78 | 59.25
 2009-07-12 03:00:03 | ABC    |       | 65.12
 2009-07-12 03:00:04 | ABC    | 67.88 |
 2009-07-12 03:00:00 | XYZ    | 47.55 | 40.15
 2009-07-12 03:00:01 | XYZ    | 44.35 | 46.78
 2009-07-12 03:00:02 | XYZ    | 71.56 | 75.78
 2009-07-12 03:00:03 | XYZ    | 85.55 | 70.21
 2009-07-12 03:00:04 | XYZ    | 45.55 | 58.65
(10 rows)
```

The following query processes the first and last values that belong to each 2-second time slice in table `trades`' column `a`. The query then calculates the exponential moving average of expression fv and lv with a smoothing factor of 50%:

```
=> SELECT symbol, slice_time, fv, lv,
     EXPONENTIAL_MOVING_AVERAGE(fv, 0.5)
       OVER (PARTITION BY symbol ORDER BY slice_time) AS ema_first,
   EXPONENTIAL_MOVING_AVERAGE(lv, 0.5)
       OVER (PARTITION BY symbol ORDER BY slice_time) AS ema_last
   FROM (
     SELECT symbol, slice_time,
        TS_FIRST_VALUE(bid1 IGNORE NULLS) as fv,
        TS_LAST_VALUE(bid2 IGNORE NULLS) AS lv
      FROM ticker TIMESERIES slice_time AS '2 seconds'
      OVER (PARTITION BY symbol ORDER BY time) ) AS sq;

 symbol |     slice_time      |  fv   |  lv   | ema_first | ema_last
--------+---------------------+-------+-------+-----------+----------
 ABC    | 2009-07-12 03:00:00 | 60.45 | 65.12 |     60.45 |    65.12
 ABC    | 2009-07-12 03:00:02 | 57.78 | 65.12 |    59.115 |    65.12
 ABC    | 2009-07-12 03:00:04 | 67.88 | 65.12 |   63.4975 |    65.12
 XYZ    | 2009-07-12 03:00:00 | 47.55 | 46.78 |     47.55 |    46.78
 XYZ    | 2009-07-12 03:00:02 | 71.56 | 70.21 |    59.555 |   58.495
 XYZ    | 2009-07-12 03:00:04 | 45.55 | 58.65 |   52.5525 |  58.5725
(6 rows)
```

See Also

TIMESERIES Clause (page 894)

Using Time Series Analytics and Using SQL Analytics in the Programmer's Guide

FIRST_VALUE [Analytic]

Allows the selection of the first value of a table or partition without having to use a self-join. If no window is specified for the current row, the default window is `UNBOUNDED PRECEDING AND CURRENT ROW`.

Behavior Type

Immutable

Syntax

```
FIRST_VALUE ( expression [ IGNORE NULLS ] ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Is the expression to evaluate; for example, a constant, column, nonanalytic function, function expression, or expressions involving any of these.
IGNORE NULLS	Returns the first non-null value in the set, or NULL if all values are NULL.
OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- The FIRST_VALUE() function lets you select a table's first value (determined by the window_order_clause) without having to use a self join. This function is useful when you want to use the first value as a baseline in calculations.
- HP recommends that you use FIRST_VALUE with the window_order_clause to produce deterministic results.
- If the first value in the set is null, then the function returns NULL unless you specify IGNORE NULLS. If you specify IGNORE NULLS, FIRST_VALUE returns the first non-null value in the set, or NULL if all values are null.

Examples

The following query, which asks for the first value in the partitioned day of week, illustrates the potential nondeterministic nature of the FIRST_VALUE function:

```
=> SELECT calendar_year, date_key, day_of_week, full_date_description,
   FIRST_VALUE(full_date_description)
     OVER(PARTITION BY calendar_month_number_in_year ORDER BY day_of_week) AS "first_value"
   FROM date_dimension
   WHERE calendar_year=2003 AND calendar_month_number_in_year=1;
```

The first value returned is January 31, 2003; however, the next time the same query is run, the first value could be January 24 or January 3, or the 10th or 17th. The reason is because the analytic ORDER BY column (day_of_week) returns rows that contain ties (multiple Fridays). These repeated values make the ORDER BY evaluation result nondeterministic, because rows that contain ties can be ordered in any way, and any one of those rows qualifies as being the first value of day_of_week.

```
 calendar_year | date_key | day_of_week | full_date_description |   first_value
---------------+----------+-------------+-----------------------+-------------
-----
```

```
          2003 |       31 | Friday      | January 31, 2003      | January 31, 2003
          2003 |       24 | Friday      | January 24, 2003      | January 31, 2003
          2003 |        3 | Friday      | January 3, 2003       | January 31, 2003
          2003 |       10 | Friday      | January 10, 2003      | January 31, 2003
          2003 |       17 | Friday      | January 17, 2003      | January 31, 2003
          2003 |        6 | Monday      | January 6, 2003       | January 31, 2003
          2003 |       27 | Monday      | January 27, 2003      | January 31, 2003
          2003 |       13 | Monday      | January 13, 2003      | January 31, 2003
          2003 |       20 | Monday      | January 20, 2003      | January 31, 2003
          2003 |       11 | Saturday    | January 11, 2003      | January 31, 2003
          2003 |       18 | Saturday    | January 18, 2003      | January 31, 2003
          2003 |       25 | Saturday    | January 25, 2003      | January 31, 2003
          2003 |        4 | Saturday    | January 4, 2003       | January 31, 2003
          2003 |       12 | Sunday      | January 12, 2003      | January 31, 2003
          2003 |       26 | Sunday      | January 26, 2003      | January 31, 2003
          2003 |        5 | Sunday      | January 5, 2003       | January 31, 2003
          2003 |       19 | Sunday      | January 19, 2003      | January 31, 2003
          2003 |       23 | Thursday    | January 23, 2003      | January 31, 2003
          2003 |        2 | Thursday    | January 2, 2003       | January 31, 2003
          2003 |        9 | Thursday    | January 9, 2003       | January 31, 2003
          2003 |       16 | Thursday    | January 16, 2003      | January 31, 2003
          2003 |       30 | Thursday    | January 30, 2003      | January 31, 2003
          2003 |       21 | Tuesday     | January 21, 2003      | January 31, 2003
          2003 |       14 | Tuesday     | January 14, 2003      | January 31, 2003
          2003 |        7 | Tuesday     | January 7, 2003       | January 31, 2003
          2003 |       28 | Tuesday     | January 28, 2003      | January 31, 2003
          2003 |       22 | Wednesday   | January 22, 2003      | January 31, 2003
          2003 |       29 | Wednesday   | January 29, 2003      | January 31, 2003
          2003 |       15 | Wednesday   | January 15, 2003      | January 31, 2003
          2003 |        1 | Wednesday   | January 1, 2003       | January 31, 2003
          2003 |        8 | Wednesday   | January 8, 2003       | January 31, 2003
(31 rows)
```

Note: The `day_of_week` results are returned in alphabetical order because of lexical rules. The fact that each day does not appear ordered by the 7-day week cycle (for example, starting with Sunday followed by Monday, Tuesday, and so on) has no affect on results.

To return deterministic results, modify the query so that it performs its analytic `ORDER BY` operations on a **unique** field, such as `date_key`:

```
=> SELECT calendar_year, date_key, day_of_week, full_date_description,
   FIRST_VALUE(full_date_description) OVER
     (PARTITION BY calendar_month_number_in_year ORDER BY date_key) AS "first_value"
   FROM date_dimension WHERE calendar_year=2003;
```

Notice that the results return a first value of January 1 for the January partition and the first value of February 1 for the February partition. Also, there are no ties in the `full_date_description` column:

```
 calendar_year | date_key | day_of_week | full_date_description |    first_value
---------------+----------+-------------+-----------------------+-------------
------
          2003 |        1 | Wednesday   | January 1, 2003       | January 1, 2003
          2003 |        2 | Thursday    | January 2, 2003       | January 1, 2003
          2003 |        3 | Friday      | January 3, 2003       | January 1, 2003
          2003 |        4 | Saturday    | January 4, 2003       | January 1, 2003
```

```
          2003 |         5 | Sunday      | January 5, 2003       | January 1, 2003
          2003 |         6 | Monday      | January 6, 2003       | January 1, 2003
          2003 |         7 | Tuesday     | January 7, 2003       | January 1, 2003
          2003 |         8 | Wednesday   | January 8, 2003       | January 1, 2003
          2003 |         9 | Thursday    | January 9, 2003       | January 1, 2003
          2003 |        10 | Friday      | January 10, 2003      | January 1, 2003
          2003 |        11 | Saturday    | January 11, 2003      | January 1, 2003
          2003 |        12 | Sunday      | January 12, 2003      | January 1, 2003
          2003 |        13 | Monday      | January 13, 2003      | January 1, 2003
          2003 |        14 | Tuesday     | January 14, 2003      | January 1, 2003
          2003 |        15 | Wednesday   | January 15, 2003      | January 1, 2003
          2003 |        16 | Thursday    | January 16, 2003      | January 1, 2003
          2003 |        17 | Friday      | January 17, 2003      | January 1, 2003
          2003 |        18 | Saturday    | January 18, 2003      | January 1, 2003
          2003 |        19 | Sunday      | January 19, 2003      | January 1, 2003
          2003 |        20 | Monday      | January 20, 2003      | January 1, 2003
          2003 |        21 | Tuesday     | January 21, 2003      | January 1, 2003
          2003 |        22 | Wednesday   | January 22, 2003      | January 1, 2003
          2003 |        23 | Thursday    | January 23, 2003      | January 1, 2003
          2003 |        24 | Friday      | January 24, 2003      | January 1, 2003
          2003 |        25 | Saturday    | January 25, 2003      | January 1, 2003
          2003 |        26 | Sunday      | January 26, 2003      | January 1, 2003
          2003 |        27 | Monday      | January 27, 2003      | January 1, 2003
          2003 |        28 | Tuesday     | January 28, 2003      | January 1, 2003
          2003 |        29 | Wednesday   | January 29, 2003      | January 1, 2003
          2003 |        30 | Thursday    | January 30, 2003      | January 1, 2003
          2003 |        31 | Friday      | January 31, 2003      | January 1, 2003
          2003 |        32 | Saturday    | February 1, 2003      | February 1, 2003
          2003 |        33 | Sunday      | February 2, 2003      | February 1, 2003
          ...
(365 rows)
```

See Also

LAST_VALUE (page 166)

TIME_SLICE (page 240)

Using SQL Analytics in the Programmer's Guide

LAG [Analytic]

Returns the value of the input expression at the given offset *before* the current row within a window.

Behavior Type

Immutable

Syntax

```
LAG ( expression [, offset ] [, default ] ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

expression	Is the expression to evaluate; for example, a constant, column, non-analytic function, function expression, or expressions involving any of these.
offset	[Optional] Indicates how great is the lag. The default value is 1 (the previous row). The *offset* parameter must be (or can be evaluated to) a constant positive integer.
default	Is `NULL`. This optional parameter is the value returned if *offset* falls outside the bounds of the table or partition. **Note:** The default input argument must be a constant value or an expression that can be evaluated to a constant; its data type is coercible to that of the first argument.
`OVER(...)`	See ***Analytic Functions.*** (page 141)

Notes

- The analytic `window_order_clause` is required but the `window_partition_clause` is optional.
- The `LAG()` function returns values from the row before the current row, letting you access more than one row in a table at the same time. This is useful for comparing values when the relative positions of rows can be reliably known. It also lets you avoid the more costly self join, which enhances query processing speed.
- See ***LEAD()*** (page 168) for how to get the *next* rows.
- Analytic functions, such as `LAG()`, cannot be nested within aggregate functions.

Examples

This example sums the current balance by date in a table and also sums the previous balance from the last day. Given the inputs that follow, the data satisfies the following conditions:

- For each `some_id`, there is exactly 1 row for each date represented by `month_date`.
- For each `some_id`, the set of dates is consecutive; that is, if there is a row for February 24 and a row for February 26, there would also be a row for February 25.
- Each `some_id` has the same set of dates.

```
=> CREATE TABLE balances (
       month_date DATE,
       current_bal INT,
       some_id INT);

=> INSERT INTO balances values ('2009-02-24', 10, 1);
=> INSERT INTO balances values ('2009-02-25', 10, 1);
=> INSERT INTO balances values ('2009-02-26', 10, 1);
=> INSERT INTO balances values ('2009-02-24', 20, 2);
=> INSERT INTO balances values ('2009-02-25', 20, 2);
=> INSERT INTO balances values ('2009-02-26', 20, 2);
=> INSERT INTO balances values ('2009-02-24', 30, 3);
```

```
   => INSERT INTO balances values ('2009-02-25', 20, 3);
   => INSERT INTO balances values ('2009-02-26', 30, 3);
```

Now run the `LAG()` function to sum the current balance for each date and sum the previous balance from the last day:

```
=> SELECT month_date,
     SUM(current_bal) as current_bal_sum,
     SUM(previous_bal) as previous_bal_sum FROM
       (SELECT month_date, current_bal,
     LAG(current_bal, 1, 0) OVER
       (PARTITION BY some_id ORDER BY month_date)
     AS previous_bal FROM balances) AS subQ
     GROUP BY month_date ORDER BY month_date;

month_date  | current_bal_sum | previous_bal_sum
------------+-----------------+------------------
 2009-02-24 |              60 |                0
 2009-02-25 |              50 |               60
 2009-02-26 |              60 |               50
(3 rows)
```

Using the same example data, the following query would not be allowed because `LAG()` is nested inside an aggregate function:

```
=> SELECT month_date,
   SUM(current_bal) as current_bal_sum,
   SUM(LAG(current_bal, 1, 0) OVER
      (PARTITION BY some_id ORDER BY month_date)) AS previous_bal_sum
   FROM some_table GROUP BY month_date ORDER BY month_date;
```

In the next example, which uses the VMart example database, the `LAG()` function first returns the annual income from the previous row, and then it calculates the difference between the income in the current row from the income in the previous row. Note: The vmart example database returns over 50,000 rows, so we'll limit the results to 20 records:

```
=> SELECT occupation, customer_key, customer_name, annual_income,
   LAG(annual_income, 1, 0) OVER (PARTITION BY occupation  ORDER BY annual_income) AS prev_income,
annual_income -
   LAG(annual_income, 1, 0) OVER (PARTITION BY occupation  ORDER BY annual_income) AS difference
   FROM customer_dimension ORDER BY occupation, customer_key LIMIT 20;

 occupation | customer_key |    customer_name     | annual_income | prev_income | difference
------------+--------------+----------------------+---------------+-------------+------------
 Accountant |           15 | Midori V. Peterson   |        692610 |      692535 |         75
 Accountant |           43 | Midori S. Rodriguez  |        282359 |      280976 |       1383
 Accountant |           93 | Robert P. Campbell   |        471722 |      471355 |        367
 Accountant |          102 | Sam T. McNulty       |        901636 |      901561 |         75
 Accountant |          134 | Martha B. Overstreet |        705146 |      704335 |        811
 Accountant |          165 | James C. Kramer      |        376841 |      376474 |        367
 Accountant |          225 | Ben W. Farmer        |         70574 |       70449 |        125
 Accountant |          270 | Jessica S. Lang      |        684204 |      682274 |       1930
 Accountant |          273 | Mark X. Lampert      |        723294 |      722737 |        557
 Accountant |          295 | Sharon K. Gauthier   |         29033 |       28412 |        621
 Accountant |          338 | Anna S. Jackson      |        816858 |      815557 |       1301
 Accountant |          377 | William I. Jones     |        915149 |      914872 |        277
 Accountant |          438 | Joanna A. McCabe     |        147396 |      144482 |       2914
 Accountant |          452 | Kim P. Brown         |        126023 |      124797 |       1226
 Accountant |          467 | Meghan K. Carcetti   |        810528 |      810284 |        244
 Accountant |          478 | Tanya E. Greenwood   |        639649 |      639029 |        620
 Accountant |          511 | Midori P. Vogel      |        187246 |      185539 |       1707
```

```
 Accountant |          525 | Alexander K. Moore   |        677433 |      677050 |        383
 Accountant |          550 | Sam P. Reyes         |        735691 |      735355 |        336
 Accountant |          577 | Robert U. Vu         |        616101 |      615439 |        662
(20 rows)
```

Continuing with the Vmart database, the next example uses both `LEAD()` and `LAG()` to return the third row after the salary in the current row and fifth salary before the salary in the current row.

```
=> SELECT hire_date, employee_key, employee_last_name,
   LEAD(hire_date, 1) OVER (ORDER BY hire_date) AS "next_hired" ,
   LAG(hire_date, 1) OVER (ORDER BY hire_date) AS "last_hired"
   FROM employee_dimension ORDER BY hire_date, employee_key;

 hire_date  | employee_key | employee_last_name | next_hired | last_hired
------------+--------------+--------------------+------------+------------
 1956-04-11 |         2694 | Farmer             | 1956-05-12 |
 1956-05-12 |         5486 | Winkler            | 1956-09-18 | 1956-04-11
 1956-09-18 |         5525 | McCabe             | 1957-01-15 | 1956-05-12
 1957-01-15 |          560 | Greenwood          | 1957-02-06 | 1956-09-18
 1957-02-06 |         9781 | Bauer              | 1957-05-25 | 1957-01-15
 1957-05-25 |         9506 | Webber             | 1957-07-04 | 1957-02-06
 1957-07-04 |         6723 | Kramer             | 1957-07-07 | 1957-05-25
 1957-07-07 |         5827 | Garnett            | 1957-11-11 | 1957-07-04
 1957-11-11 |          373 | Reyes              | 1957-11-21 | 1957-07-07
 1957-11-21 |         3874 | Martin             | 1958-02-06 | 1957-11-11
(10 rows)
```

The following example specifies arguments that use different data types; for example `annual_income(INT) and occupation(VARCHAR)`. The query returns an error:

```
=> SELECT customer_key, customer_name, occupation, annual_income,
   LAG (annual_income, 1, occupation) OVER
      (PARTITION BY occupation ORDER BY customer_key) LAG1
   FROM customer_dimension ORDER BY 3, 1;

   ERROR:  Third argument of lag could not be converted from type character varying
to type int8
   HINT:  You may need to add explicit type cast.
```

See Also

LEAD (page 168)

Using SQL Analytics in the Programmer's Guide

LAST_VALUE [Analytic]

Returns values of the expression from the last row of a window for the current row. If no window is specified for the current row, the default window is `UNBOUNDED PRECEDING AND CURRENT ROW`.

Behavior Type

Immutable

Syntax

```
LAST_VALUE ( expression [ IGNORE NULLS ] ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Is the expression to evaluate; for example, a constant, column, nonanalytic function, function expression, or expressions involving any of these.
IGNORE NULLS	Returns the last non-null value in the set, or NULL if all values are NULL.
OVER(...)	See ***Analytic Functions.*** (page 141)

Notes

- The LAST_VALUE() function lets you select a window's last value (determined by the window_order_clause), without having to use a self join. This function is useful when you want to use the last value as a baseline in calculations.
- LAST_VALUE() takes the last record from the partition after the analytic window_order_clause. The expression is then computed against the last record, and results are returned.
- HP recommends that you use LAST_VALUE with the window_order_clause to produce deterministic results.

  TIP: Due to default window semantics, LAST_VALUE does not always return the last value of a partition. If you omit the *window_frame_clause* from the analytic clause, LAST_VALUE operates on this default window. Although results can seem non-intuitive by not returning the bottom of the current partition, it returns the bottom of the window, which continues to change along with the current input row being processed. If you want to return the last value of a partition, use UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. See examples below.
- If the last value in the set is null, then the function returns NULL unless you specify IGNORE NULLS. If you specify IGNORE NULLS, LAST_VALUE returns the fist non-null value in the set, or NULL if all values are null.

Example

Using the schema defined in Window Framing in the Programmer's Guide, the following query does not show the highest salary value by department; instead it shows the highest salary value by department by salary.

```
=> SELECT deptno, sal, empno, LAST_VALUE(sal)
      OVER (PARTITION BY deptno ORDER BY sal) AS lv
   FROM emp;

 deptno | sal | empno |    lv
--------+-----+-------+--------
     10 | 101 |     1 |     101
```

```
     10 | 104 |      4 |     104
     20 | 100 |     11 |     100
     20 | 109 |      7 |     109
     20 | 109 |      6 |     109
     20 | 109 |      8 |     109
     20 | 110 |     10 |     110
     20 | 110 |      9 |     110
     30 | 102 |      2 |     102
     30 | 103 |      3 |     103
     30 | 105 |      5 |     105
```

If you include the window_frame clause `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`, the LAST_VALUE() function will return the highest salary by department, an accurate representation of the information.

```
=> SELECT deptno, sal, empno, LAST_VALUE(sal)
      OVER (PARTITION BY deptno ORDER BY sal
            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
   FROM emp;

 deptno | sal | empno |    lv
--------+-----+-------+--------
     10 | 101 |     1 |     104
     10 | 104 |     4 |     104
     20 | 100 |    11 |     110
     20 | 109 |     7 |     110
     20 | 109 |     6 |     110
     20 | 109 |     8 |     110
     20 | 110 |    10 |     110
     20 | 110 |     9 |     110
     30 | 102 |     2 |     105
     30 | 103 |     3 |     105
     30 | 105 |     5 |     105
```

For additional examples, see ***FIRST_VALUE()*** (page 160).

See Also

FIRST_VALUE (page 160)

TIME_SLICE (page 240)

Using SQL for Analytics in the Programmer's Guide

LEAD [Analytic]

Returns the value of the input expression at the given offset *after* the current row within a window.

Behavior Type

Immutable

Syntax

```
LEAD ( expression [, offset ] [, default ] ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

expression	Is the expression to evaluate; for example, a constant, column, nonanalytic function, function expression, or expressions involving any of these.
offset	Is an optional parameter that defaults to 1 (the next row). The *offset* parameter must be (or can be evaluated to) a constant positive integer.
default	Is NULL. This optional parameter is the value returned if *offset* falls outside the bounds of the table or partition. **Note:** The third input argument must be a constant value or an expression that can be evaluated to a constant; its data type is coercible to that of the first argument.
OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- The analytic `window_order_clause` is required but the `window_partition_clause` is optional.
- The `LEAD()` function returns values from the row after the current row, letting you access more than one row in a table at the same time. This is useful for comparing values when the relative positions of rows can be reliably known. It also lets you avoid the more costly self join, which enhances query processing speed.
- Analytic functions, such as `LEAD()`, cannot be nested within aggregate functions.

Examples

In this example, the `LEAD()` function finds the hire date of the employee hired just after the current row:

```
=> SELECT employee_region, hire_date, employee_key, employee_last_name,
   LEAD(hire_date, 1) OVER (PARTITION BY employee_region ORDER BY hire_date) AS "next_hired"
   FROM employee_dimension ORDER BY employee_region, hire_date, employee_key;

  employee_region   | hire_date  | employee_key | employee_last_name | next_hired
--------------------+------------+--------------+--------------------+---------
---
 East               | 1956-04-08 |         9218 | Harris             | 1957-02-06
 East               | 1957-02-06 |         7799 | Stein              | 1957-05-25
 East               | 1957-05-25 |         3687 | Farmer             | 1957-06-26
 East               | 1957-06-26 |         9474 | Bauer              | 1957-08-18
 East               | 1957-08-18 |          570 | Jefferson          | 1957-08-24
 East               | 1957-08-24 |         4363 | Wilson             | 1958-02-17
 East               | 1958-02-17 |         6457 | McCabe             | 1958-06-26
 East               | 1958-06-26 |         6196 | Li                 | 1958-07-16
```

```
 East              | 1958-07-16 |         7749 | Harris             | 1958-09-18
 East              | 1958-09-18 |         9678 | Sanchez            | 1958-11-10
(10 rows)
```

The next example uses both `LEAD()` and `LAG()` to return the third row after the salary in the current row and fifth salary before the salary in the current row.

```
=> SELECT hire_date, employee_key, employee_last_name,
   LEAD(hire_date, 1) OVER (ORDER BY hire_date) AS "next_hired" ,
   LAG(hire_date, 1) OVER (ORDER BY hire_date) AS "last_hired"
   FROM employee_dimension ORDER BY hire_date, employee_key;

 hire_date  | employee_key | employee_last_name | next_hired | last_hired
------------+--------------+--------------------+------------+------------
 1956-04-11 |         2694 | Farmer             | 1956-05-12 |
 1956-05-12 |         5486 | Winkler            | 1956-09-18 | 1956-04-11
 1956-09-18 |         5525 | McCabe             | 1957-01-15 | 1956-05-12
 1957-01-15 |          560 | Greenwood          | 1957-02-06 | 1956-09-18
 1957-02-06 |         9781 | Bauer              | 1957-05-25 | 1957-01-15
 1957-05-25 |         9506 | Webber             | 1957-07-04 | 1957-02-06
 1957-07-04 |         6723 | Kramer             | 1957-07-07 | 1957-05-25
 1957-07-07 |         5827 | Garnett            | 1957-11-11 | 1957-07-04
 1957-11-11 |          373 | Reyes              | 1957-11-21 | 1957-07-07
 1957-11-21 |         3874 | Martin             | 1958-02-06 | 1957-11-11
(10 rows)
```

The following example returns employee name and salary, along with the next highest and lowest salaries.

```
=> SELECT employee_last_name, annual_salary,
       NVL(LEAD(annual_salary) OVER (ORDER BY annual_salary),
         MIN(annual_salary) OVER()) "Next Highest",
       NVL(LAG(annual_salary) OVER (ORDER BY annual_salary),
         MAX(annual_salary)  OVER()) "Next Lowest"
   FROM employee_dimension;

 employee_last_name | annual_salary | Next Highest | Next Lowest
--------------------+---------------+--------------+-------------
 Nielson            |          1200 |         1200 |      995533
 Lewis              |          1200 |         1200 |        1200
 Harris             |          1200 |         1202 |        1200
 Robinson           |          1202 |         1202 |        1200
 Garnett            |          1202 |         1202 |        1202
 Weaver             |          1202 |         1202 |        1202
 Nielson            |          1202 |         1202 |        1202
 McNulty            |          1202 |         1204 |        1202
 Farmer             |          1204 |         1204 |        1202
 Martin             |          1204 |         1204 |        1204
(10 rows)
```

The next example returns, for each assistant director in the employees table, the hire date of the director hired just after the director on the current row. For example, Jackson was hired on 2007-12-28, and the next director hired was Bauer:

```
=> SELECT employee_last_name, hire_date,
      LEAD(hire_date, 1) OVER (ORDER BY hire_date DESC) as "NextHired"
```

```
   FROM employee_dimension WHERE job_title = 'Assistant Director';
 employee_last_name | hire_date  | NextHired
--------------------+------------+------------
 Jackson            | 2007-12-28 | 2007-12-26
 Bauer              | 2007-12-26 | 2007-12-11
 Miller             | 2007-12-11 | 2007-12-07
 Fortin             | 2007-12-07 | 2007-11-27
 Harris             | 2007-11-27 | 2007-11-15
 Goldberg           | 2007-11-15 |

(5 rows)
```

See Also

LAG (page 163)

Using SQL for Analytics in the Programmer's Guide

MAX [Analytic]

Returns the maximum value of an expression within a window. The return value is the same as the expression data type.

Behavior Type

Immutable

Syntax

```
MAX ( [ DISTINCT ] expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

`DISTINCT`	Is meaningless in this context.
expression	Can be any expression for which the maximum value is calculated, typically a ***column reference*** (see "***Column References***" on page 54).
`OVER(...)`	See ***Analytic Functions***. (page 141)

Example

The following query computes the deviation between the employees' annual salary and the maximum annual salary in Massachusetts:

```
=> SELECT employee_state, annual_salary,
     MAX(annual_salary)
       OVER(PARTITION BY employee_state ORDER BY employee_key) max,
```

```
          annual_salary- MAX(annual_salary)
       OVER(PARTITION BY employee_state ORDER BY employee_key) diff
   FROM employee_dimension
   WHERE employee_state = 'MA';

 employee_state | annual_salary |  max   |  diff
----------------+---------------+--------+---------
 MA             |          1918 | 995533 | -993615
 MA             |          2058 | 995533 | -993475
 MA             |          2586 | 995533 | -992947
 MA             |          2500 | 995533 | -993033
 MA             |          1318 | 995533 | -994215
 MA             |          2072 | 995533 | -993461
 MA             |          2656 | 995533 | -992877
 MA             |          2148 | 995533 | -993385
 MA             |          2366 | 995533 | -993167
 MA             |          2664 | 995533 | -992869
(10 rows)
```

See Also

MAX (page 128) aggregate function

MIN (page 174) analytic function

Using SQL Analytics in the Programmer's Guide

MEDIAN [Analytic]

A numerical value of an expression in a result set within a window, which separates the higher half of a sample from the lower half. For example, a query can retrieve the median of a finite list of numbers by arranging all observations from lowest value to highest value and then picking the middle one.

If there is an even number of observations, then there is no single middle value; thus, the median is defined to be the mean (average) of the two middle values

MEDIAN() is an alias for 50% PERCENTILE(); for example:

```
PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY expression)
```

Behavior Type

Immutable

Syntax

```
MEDIAN ( expression ) OVER ( [ window_partition_clause (page 143) ] )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the middle value or an interpolated value that would be the middle value once the values are sorted. Null values are ignored in the calculation.
OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- For each row, MEDIAN() returns the value that would fall in the middle of a value set within each partition.
- HP Vertica determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.
- MEDIAN() does not allow the window_order_clause or window_frame_clause.

Examples

The following query computes the median annual income for first 500 customers in Wisconsin and in the District of Columbia. Note that median is reported for every row in each partitioned result set:

```
=> SELECT customer_state, annual_income,
     MEDIAN(annual_income) OVER (PARTITION BY customer_state) AS MEDIAN
   FROM customer_dimension
   WHERE customer_state IN ('DC','WI')
   ORDER BY customer_state;

 customer_state | customer_key | annual_income | MEDIAN
----------------+--------------+---------------+----------
 DC             |          120 |        299768 |   535413
 DC             |          113 |        535413 |   535413
 DC             |          130 |        848360 |   535413
-------------------------------------------------------------
 WI             |          372 |         34962 |   668147
 WI             |          437 |         47128 |   668147
 WI             |          435 |         67770 |   668147
 WI             |          282 |        638054 |   668147
 WI             |          314 |        668147 |   668147
 WI             |          128 |        675608 |   668147
 WI             |          179 |        825304 |   668147
 WI             |          302 |        827618 |   668147
 WI             |           29 |        922760 |   668147
(12 rows)
```

See Also

PERCENTILE_CONT (page 178)

Using SQL Analytics in the Programmer's Guide

MIN [Analytic]

Returns the minimum value of an expression within a window. The return value is the same as the expression data type.

Behavior Type

Immutable

Syntax

```
MIN ( [ DISTINCT ] expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

DISTINCT	Is meaningless in this context.
expression	Can be any expression for which the minimum value is calculated, typically a ***column reference*** (see "***Column References***" on page 54).
OVER(...)	See ***Analytic Functions***. (page 141)

Examples

The following query computes the deviation between the employees' annual salary and the minimum annual salary in Massachusetts:

```
=> SELECT employee_state, annual_salary,
      MIN(annual_salary)
      OVER(PARTITION BY employee_state ORDER BY employee_key) min,
        annual_salary- MIN(annual_salary)
      OVER(PARTITION BY employee_state ORDER BY employee_key) diff
   FROM employee_dimension
   WHERE employee_state = 'MA';

 employee_state | annual_salary | min  | diff
----------------+---------------+------+------
 MA             |          1918 | 1204 |  714
 MA             |          2058 | 1204 |  854
 MA             |          2586 | 1204 | 1382
 MA             |          2500 | 1204 | 1296
 MA             |          1318 | 1204 |  114
 MA             |          2072 | 1204 |  868
 MA             |          2656 | 1204 | 1452
 MA             |          2148 | 1204 |  944
 MA             |          2366 | 1204 | 1162
 MA             |          2664 | 1204 | 1460
(10 rows)
```

See Also

MIN (page 129) aggregate function

MAX (page 171) analytic function

Using SQL Analytics in the Programmer's Guide

NTILE [Analytic]

Equally divides an ordered data set (partition) into a *{value}* number of subsets within a window, with buckets (subsets) numbered 1 through *constant-value*. For example, if *constant-value* = 4, then each row in the partition is assigned a number from 1 to 4. If the partition contains 20 rows, the first 5 would be assigned 1, the next 5 would be assigned 2, and so on.

Behavior Type

Immutable

Syntax

```
NTILE ( constant-value ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144) )
```

Parameters

constant-value	Represents the number of subsets and must resolve to a positive constant for each partition.
OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- The analytic `window_order_clause` is required but the `window_partition_clause` is optional.
- If the number of subsets is greater than the number of rows, then a number of subsets equal to the number of rows is filled, and the remaining subsets are empty.
- In the event the cardinality of the partition is not evenly divisible by the number of subsets, the rows are distributed so no subset has more than 1 row more then any other subset, and the lowest subsets are the ones that have extra rows. For example, using constant-value = 4 again and the number of rows = 21, subset = 1 has 6 rows, subset = 2 has 5, and so on.
- Analytic functions, such as `NTILE()`, cannot be nested within aggregate functions.

Examples

The following query assigns each month's sales total into one of four subsets:

```
=> SELECT calendar_month_name AS MONTH, SUM(sales_quantity),
     NTILE(4) OVER (ORDER BY SUM(sales_quantity)) AS NTILE
```

```
   FROM store.store_sales_fact JOIN date_dimension
   USING(date_key)
   GROUP BY calendar_month_name
   ORDER BY NTILE;

   MONTH    | SUM   | NTILE
------------+------+-------
 February   |  755 |      1
 June       |  842 |      1
 September  |  849 |      1
 January    |  881 |      2
 May        |  882 |      2
 July       |  894 |      2
 August     |  921 |      3
 April      |  952 |      3
 March      |  987 |      3
 October    | 1010 |      4
 November   | 1026 |      4
 December   | 1094 |      4
(12 rows)
```

See Also

PERCENTILE_CONT (page 178)

WIDTH_BUCKET (page 319)

Using SQL Analytics in the Programmer's Guide

PERCENT_RANK [Analytic]

Calculates the relative rank of a row for a given row in a group within a window by dividing that row's rank less 1 by the number of rows in the partition, also less 1. This function always returns values from 0 to 1 inclusive. The first row in any set has a `PERCENT_RANK()` of 0. The return value is `NUMBER`.

```
( rank - 1 ) / ( [ rows ] - 1 )
```

In the above formula, `rank` is the rank position of a row in the group and `rows` is the total number of rows in the partition defined by the `OVER()` clause.

Behavior Type

Immutable

Syntax

```
PERCENT_RANK ( ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

OVER(...)	See ***Analytic Functions***. (page 141)

Notes

The window_order_clause is required but the window_partition_clause is optional.

Examples

The following example finds the percent rank of gross profit for different states within each month of the first quarter:

```
=> SELECT calendar_month_name AS MONTH, store_state ,
      SUM(gross_profit_dollar_amount),
      PERCENT_RANK() OVER (PARTITION BY calendar_month_name
      ORDER BY SUM(gross_profit_dollar_amount)) AS PERCENT_RANK
   FROM store.store_sales_fact JOIN date_dimension
   USING(date_key)
   JOIN store.store_dimension
   USING (store_key)
   WHERE calendar_month_name IN ('January','February','March')
   AND store_state IN ('OR','IA','DC','NV','WI')
   GROUP BY calendar_month_name, store_state
   ORDER BY calendar_month_name, PERCENT_RANK;

  MONTH   | store_state | SUM  |   PERCENT_RANK
----------+-------------+------+-------------------
 February | OR          |   16 |                 0
 February | IA          |   47 |              0.25
 February | DC          |   94 |               0.5
 February | NV          |  113 |              0.75
 February | WI          |  119 |                 1
 January  | IA          | -263 |                 0
 January  | OR          |   91 | 0.333333333333333
 January  | NV          |  372 | 0.666666666666667
 January  | DC          |  497 |                 1
 March    | NV          | -141 |                 0
 March    | OR          |  224 |                 1

(11 rows)
```

The following example calculates, for each employee, the percent rank of the employee's salary by their job title:

```
=> SELECT job_title, employee_last_name, annual_salary,
      PERCENT_RANK()
      OVER (PARTITION BY job_title ORDER BY annual_salary DESC) AS percent_rank
   FROM employee_dimension
   ORDER BY percent_rank, annual_salary;

  job_title         | employee_last_name | annual_salary |   PERCENT_RANK
--------------------+--------------------+---------------+--------------------
-
```

```
 CEO                | Campbell           |         963914 |                     0
 Co-Founder         | Nguyen             |         968625 |                     0
 Founder            | Overstreet         |         995533 |                     0
 Greeter            | Peterson           |           3192 | 0.00113895216400911
 Greeter            | Greenwood          |           3192 | 0.00113895216400911
 Customer Service   | Peterson           |           3190 | 0.00121065375302663
 Delivery Person    | Rodriguez          |           3192 | 0.00121065375302663
 Shelf Stocker      | Martin             |           3194 | 0.00125786163522013
 Shelf Stocker      | Vu                 |           3194 | 0.00125786163522013
 Marketing          | Li                 |          99711 | 0.00190114068441065
 Assistant Director | Sanchez            |          99913 | 0.00190839694656489
 Branch Manager     | Perkins            |          99901 | 0.00192307692307692
 Advertising        | Lampert            |          99809 | 0.00204918032786885
 Sales              | Miller             |          99727 | 0.00211416490486258
 Shift Manager      | King               |          99904 | 0.00215982721382289
 Custodian          | Bauer              |           3196 | 0.00235849056603774
 Custodian          | Goldberg           |           3196 | 0.00235849056603774
 Customer Service   | Fortin             |           3184 | 0.00242130750605327
 Delivery Person    | Greenwood          |           3186 | 0.00242130750605327
 Cashier            | Overstreet         |           3178 | 0.00243605359317905
 Regional Manager   | McCabe             |         199688 | 0.00306748466257669
 VP of Sales        | Li                 |         199309 | 0.00313479623824451
 Director of HR     | Goldberg           |         199592 | 0.00316455696202532
 Head of Marketing  | Stein              |         199941 | 0.00317460317460317
 VP of Advertising  | Goldberg           |         199036 | 0.00323624595469256
 Head of PR         | Stein              |         199767 | 0.00323624595469256
 Customer Service   | Rodriguez          |           3180 |  0.0036319612590799
 Delivery Person    | King               |           3184 |  0.0036319612590799
 Cashier            | Dobisz             |           3174 | 0.00365408038976857
 Cashier            | Miller             |           3174 | 0.00365408038976857
 Marketing          | Dobisz             |          99655 | 0.00380228136882129
 Branch Manager     | Gauthier           |          99082 |               0.025
 Branch Manager     | Moore              |          98415 |                0.05
...
```

See Also

CUME_DIST (page 155)

Using SQL Analytics in the Programmer's Guide

PERCENTILE_CONT [Analytic]

An inverse distribution function where, for each row, `PERCENTILE_CONT()` returns the value that would fall into the specified percentile among a set of values in each partition within a window. For example, if the argument to the function is 0.5, the result of the function is the median of the data set (the 50th percentile). `PERCENTILE_CONT()` assumes a continuous distribution data model. Nulls are ignored.

Behavior Type

Immutable

Syntax

```
PERCENTILE_CONT ( %_number ) WITHIN GROUP (
... ORDER BY expression [ ASC | DESC ] ) OVER (
... [ window_partition_clause (page 143) ] )
```

Parameters

%_number	Is the percentile value, which must be a `FLOAT` constant ranging from 0 to 1 (inclusive).
`WITHIN GROUP(ORDER BY` *expression*`)`	Specifies how the data is sorted within each group. `ORDER BY` takes only one column/expression that must be `INTEGER`, `FLOAT`, `INTERVAL`, or `NUMERIC` data type. Nulls are discarded. **Note:** The `WITHIN GROUP(ORDER BY)` clause does not guarantee the order of the SQL result. Use the ***SQL ORDER BY clause*** (page 893) to guarantee the ordering of the final result set.
`ASC | DESC`	Specifies the ordering sequence as ascending (default) or descending.
`OVER(...)`	See ***Analytic Functions***. (page 141)

Notes

- HP Vertica computes the percentile by first computing the row number where the percentile row would exist; for example:

  `ROW_NUMBER = 1 + PERCENTILE_VALUE * (NUMBER_OF_ROWS_IN_PARTITION -1)`

  If the `CEILING(ROW_NUMBER) = FLOOR(ROW_NUMBER)`, then the percentile is the value *at* the `ROW_NUMBER`. Otherwise there was an even number of rows, and HP Vertica interpolates the value between the rows. In this case, the percentile `CEILING_VAL` = get the value at the `CEILING(ROW_NUMBER)`. `FLOOR_VAL` = get the value at the `FLOOR(ROW_NUMBER)` would be `(CEILING(ROW_NUMBER) - ROW_NUMBER) * CEILING_VAL + (ROW_NUMBER - FLOOR(ROW_NUMBER) * FLOOR+VAL`.

  If `CEIL(num) = FLOOR(num) = num`, then retrieve the value in that row. Otherwise compute values at `[ CEIL(num) + FLOOR(num) ] / 2`
- Specifying `ASC` or `DESC` in the `WITHIN GROUP` clause affects results as long as the percentile parameter is not .5.
- The `MEDIAN()` function is a specific case of `PERCENTILE_CONT()` where the percentile value defaults to 0.5. For more information, see `MEDIAN()` (page 172).

Examples

This query computes the median annual income per group for the first 500 customers in Wisconsin and the District of Columbia.

```
=> SELECT customer_state, customer_key, annual_income,
      PERCENTILE_CONT(.5) WITHIN GROUP(ORDER BY annual_income)
      OVER (PARTITION BY customer_state) AS PERCENTILE_CONT
   FROM customer_dimension
   WHERE customer_state IN ('DC','WI')
```

```
   AND customer_key < 300
   ORDER BY customer_state, customer_key;

 customer_state | customer_key | annual_income | PERCENTILE_CONT
----------------+--------------+---------------+-----------------
 DC             |          104 |        658383 |          658383
 DC             |          168 |        417092 |          658383
 DC             |          245 |        670205 |          658383
 WI             |          106 |        227279 |          458607
 WI             |          127 |        703889 |          458607
 WI             |          209 |        458607 |          458607
(6 rows)
```

The median value for DC is 65838, and the median value for WI is 458607. Note that with a %_number of `.5` in the above query, `PERCENTILE_CONT()` returns the same result as `MEDIAN()` in the following query:

```
=> SELECT customer_state, customer_key, annual_income,
      MEDIAN(annual_income)
      OVER (PARTITION BY customer_state) AS MEDIAN
   FROM customer_dimension
   WHERE customer_state IN ('DC','WI')
   AND customer_key < 300
   ORDER BY customer_state, customer_key;

 customer_state | customer_key | annual_income | MEDIAN
----------------+--------------+---------------+--------
 DC             |          104 |        658383 | 658383
 DC             |          168 |        417092 | 658383
 DC             |          245 |        670205 | 658383
 WI             |          106 |        227279 | 458607
 WI             |          127 |        703889 | 458607
 WI             |          209 |        458607 | 458607
(6 rows)
```

See Also

MEDIAN (page 172)

Using SQL Analytics in the Programmer's Guide

PERCENTILE_DISC [Analytic]

An inverse distribution function where, for each row, `PERCENTILE_DISC()` returns the value that would fall into the specified percentile among a set of values in each partition within a window. `PERCENTILE_DISC()` assumes a discrete distribution data model. Nulls are ignored.

Behavior Type

Immutable

Syntax

```
PERCENTILE_DISC ( %_number ) WITHIN GROUP (
```

```
... ORDER BY expression [ ASC | DESC ] ) OVER (
... [ window_partition_clause (page 143) ] )
```

Parameters

%_number	Is the percentile value, which must be a FLOAT constant ranging from 0 to 1 (inclusive).
WITHIN GROUP(ORDER BY *expression*)	Specifies how the data is sorted within each group. ORDER BY takes only one column/expression that must be INTEGER, FLOAT, INTERVAL, or NUMERIC data type. Nulls are discarded. **Note:** The WITHIN GROUP(ORDER BY) clause does not guarantee the order of the SQL result. Use the ***SQL ORDER BY clause*** (page 893) to guarantee the ordering of the final result set.
ASC | DESC	Specifies the ordering sequence as ascending (default) or descending.
OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- PERCENTILE_DISC(%_number) examines the cumulative distribution values in each group until it finds one that is greater than or equal to *%_number*.
- HP Vertica computes the percentile where, for each row, PERCENTILE_DISC outputs the first value of the WITHIN GROUP(ORDER BY) column whose CUME_DIST (cumulative distribution) value is >= the argument FLOAT value (for example, .4). Specifically:

  ```
  PERCENTILE_DIST(.4) WITHIN GROUP (ORDER BY salary) OVER(PARTITION By deptno) ...
  ```

  If you write, for example, `SELECT CUME_DIST() OVER(ORDER BY salary) FROM table;` you notice that the smallest CUME_DIST value that is greater than .4 is also the PERCENTILE_DISC.

Examples

This query computes the 20th percentile annual income by group for first 500 customers in Wisconsin and the District of Columbia.

```
=> SELECT customer_state, customer_key, annual_income,
      PERCENTILE_DISC(.2) WITHIN GROUP(ORDER BY annual_income)
      OVER (PARTITION BY customer_state) AS PERCENTILE_DISC
   FROM customer_dimension
   WHERE customer_state IN ('DC','WI')
   AND customer_key < 300
   ORDER BY customer_state, customer_key;

 customer_state | customer_key | annual_income | PERCENTILE_DISC
----------------+--------------+---------------+-----------------
 DC             |          104 |        658383 |          417092
 DC             |          168 |        417092 |          417092
 DC             |          245 |        670205 |          417092
 WI             |          106 |        227279 |          227279
```

```
 WI              |          127 |        703889 |          227279
 WI              |          209 |        458607 |          227279
(6 rows)
```

See Also

CUME_DIST (page 155)

PERCENTILE_CONT (page 178)

Using SQL Analytics in the Programmer's Guide

RANK [Analytic]

Assigns a rank to each row returned from a query with respect to the other ordered rows, based on the values of the expressions in the window `ORDER BY` clause. The data within a group is sorted by the `ORDER BY` clause and then a numeric ranking is assigned to each row in turn, starting with 1, and continuing up. Rows with the same values of the `ORDER BY` expressions receive the same rank; however, if two rows receive the same rank (a tie), `RANK()` skips the ties. If, for example, two rows are numbered 1, `RANK()` skips number 2 and assigns 3 to the next row in the group. This is in contrast to `DENSE_RANK()` (page 156), which does not skip values.

Behavior Type

Immutable

Syntax

```
RANK ( ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

`OVER(...)`	See ***Analytic Functions***. (page 141)

Notes

- Ranking functions return a rank value for each row in a result set based on the order specified in the query. For example, a territory sales manager might want to identify the top or bottom ranking sales associates in a department or the highest/lowest-performing sales offices by region.
- `RANK()` requires an `OVER()` clause. The `window_partition_clause` is optional.
- In ranking functions, `OVER()` specifies the measures *expression* on which ranking is done and defines the order in which rows are sorted in each group (or partition). Once the data is sorted within each partition, ranks are given to each row starting from 1.

- The primary difference between RANK and DENSE_RANK is that RANK leaves gaps when ranking records; DENSE_RANK leaves no gaps. For example, if more than one record occupies a particular position (a tie), RANK places all those records in that position and it places the next record after a gap of the additional records (it skips one). DENSE_RANK places all the records in that position only—it does not leave a gap for the next rank.

  If there is a tie at the third position with two records having the same value, RANK and DENSE_RANK place both the records in the third position only, but RANK has the next record at the fifth position — leaving a gap of 1 position—while DENSE_RANK places the next record at the forth position (no gap).
- If you omit NULLS FIRST | LAST | AUTO, the ordering of the null values depends on the ASC or DESC arguments. Null values are considered larger than any other values. If the ordering sequence is ASC, then nulls appear last; nulls appear first otherwise. Nulls are considered equal to other nulls and, therefore, the order in which nulls are presented is non-deterministic.

Examples

This example ranks the longest-standing customers in Massachusetts. The query first computes the customer_since column by region, and then partitions the results by customers with businesses in MA. Then within each region, the query ranks customers over the age of 70.

```
=> SELECT customer_type, customer_name,
      RANK() OVER (PARTITION BY customer_region ORDER BY customer_since) as rank
   FROM customer_dimension
   WHERE customer_state = 'MA'
   AND customer_age > '70';

 customer_type | customer_name | rank
---------------+---------------+------
 Company       | Virtadata     |    1
 Company       | Evergen       |    2
 Company       | Infocore      |    3
 Company       | Goldtech      |    4
 Company       | Veritech      |    5
 Company       | Inishop       |    6
 Company       | Intracom      |    7
 Company       | Virtacom      |    8
 Company       | Goldcom       |    9
 Company       | Infostar      |   10
 Company       | Golddata      |   11
 Company       | Everdata      |   12
 Company       | Goldcorp      |   13
(13 rows)
```

The following example shows the difference between RANK and DENSE_RANK when ranking customers by their annual income. Notice that RANK has a tie at 10 and skips 11, while DENSE_RANK leaves no gaps in the ranking sequence:

```
=> SELECT customer_name, SUM(annual_income),
      RANK () OVER (ORDER BY TO_CHAR(SUM(annual_income),'100000') DESC) rank,
      DENSE_RANK () OVER (ORDER BY TO_CHAR(SUM(annual_income),'100000') DESC)
dense_rank
   FROM customer_dimension
```

```
   GROUP BY customer_name
   LIMIT 15;
    customer_name    |  sum   | rank | dense_rank
---------------------+-------+------+------------
 Brian M. Garnett    | 99838 |    1 |          1
 Tanya A. Brown      | 99834 |    2 |          2
 Tiffany P. Farmer   | 99826 |    3 |          3
 Jose V. Sanchez     | 99673 |    4 |          4
 Marcus D. Rodriguez | 99631 |    5 |          5
 Alexander T. Nguyen | 99604 |    6 |          6
 Sarah G. Lewis      | 99556 |    7 |          7
 Ruth Q. Vu          | 99542 |    8 |          8
 Theodore T. Farmer  | 99532 |    9 |          9
 Daniel P. Li        | 99497 |   10 |         10
 Seth E. Brown       | 99497 |   10 |         10
 Matt X. Gauthier    | 99402 |   12 |         11
 Rebecca W. Lewis    | 99296 |   13 |         12
 Dean L. Wilson      | 99276 |   14 |         13
 Tiffany A. Smith    | 99257 |   15 |         14
(15 rows)
```

See Also

DENSE_RANK (page 156)

Using SQL Analytics in the Programmer's Guide

ROW_NUMBER [Analytic]

Assigns a unique number, sequentially, starting from 1, to each row in a partition within a window.

Behavior Type

Immutable

Syntax

```
ROW_NUMBER ( ) OVER (
... [ window_partition_clause (page 143) ]
... window_order_clause (page 144)  )
```

Parameters

`OVER(...)`	See ***Analytic Functions.*** (page 141)

Notes

- `ROW_NUMBER()` is an HP Vertica extension, not part of the SQL-99 standard. It requires an OVER() clause. The `window_partition_clause` is optional.
- You can use the optional partition clause to group data into partitions before operating on it; for example:
  `SUM OVER (PARTITION BY col1, col2, ...)`

- You can substitute any `RANK()` example for `ROW_NUMBER()`. The difference is that `ROW_NUMBER` assigns a unique ordinal number, starting with 1, to each row in the ordered set.

Examples

The following query first partitions customers in the customer_dimension table by occupation and then ranks those customers based on the ordered set specified by the analytic partition_clause.

```
=> SELECT occupation, customer_key, customer_since, annual_income,
      ROW_NUMBER() OVER (PARTITION BY occupation) AS customer_since_row_num
   FROM public.customer_dimension
   ORDER BY occupation, customer_since_row_num;
```

```
     occupation     | customer_key | customer_since | annual_income | customer_since_row_num
--------------------+--------------+----------------+---------------+------------------------
 Accountant         |        19453 | 1973-11-06     |        602460 |                      1
 Accountant         |        42989 | 1967-07-09     |        850814 |                      2
 Accountant         |        24587 | 1995-05-18     |        180295 |                      3
 Accountant         |        26421 | 2001-10-08     |        126490 |                      4
 Accountant         |        37783 | 1993-03-16     |        790282 |                      5
 Accountant         |        39170 | 1980-12-21     |        823917 |                      6
 Banker             |        13882 | 1998-04-10     |         15134 |                      1
 Banker             |        14054 | 1989-03-16     |        961850 |                      2
 Banker             |        15850 | 1996-01-19     |        262267 |                      3
 Banker             |        29611 | 2004-07-14     |        739016 |                      4
 Doctor             |          261 | 1969-05-11     |        933692 |                      1
 Doctor             |         1264 | 1981-07-19     |        593656 |                      2
 Psychologist       |         5189 | 1999-05-04     |        397431 |                      1
 Psychologist       |         5729 | 1965-03-26     |        339319 |                      2
 Software Developer |         2513 | 1996-09-22     |        920003 |                      1
 Software Developer |         5927 | 2001-03-12     |        633294 |                      2
 Software Developer |         9125 | 1971-10-06     |        198953 |                      3
 Software Developer |        16097 | 1968-09-02     |        748371 |                      4
 Software Developer |        23137 | 1988-12-07     |         92578 |                      5
 Software Developer |        24495 | 1989-04-16     |        149371 |                      6
 Software Developer |        24548 | 1994-09-21     |        743788 |                      7
 Software Developer |        33744 | 2005-12-07     |        735003 |                      8
 Software Developer |         9684 | 1970-05-20     |        246000 |                      9
 Software Developer |        24278 | 2001-11-14     |        122882 |                     10
 Software Developer |        27122 | 1994-02-05     |        810044 |                     11
 Stock Broker       |         5950 | 1965-01-20     |        752120 |                      1
 Stock Broker       |        12517 | 2003-06-13     |        380102 |                      2
 Stock Broker       |        33010 | 1984-05-07     |        384463 |                      3
 Stock Broker       |        46196 | 1972-11-28     |        497049 |                      4
 Stock Broker       |         8710 | 2005-02-11     |         79387 |                      5
 Writer             |         3149 | 1998-11-17     |        643972 |                      1
 Writer             |        17124 | 1965-01-18     |        444747 |                      2
 Writer             |        20100 | 1994-08-13     |        106097 |                      3
 Writer             |        23317 | 2003-05-27     |        511750 |                      4
 Writer             |        42845 | 1967-10-23     |        433483 |                      5
 Writer             |        47560 | 1997-04-23     |        515647 |                      6
(39 rows)
```

See Also

RANK (page 182)

Using SQL for Analytics in the Programmer's Guide

STDDEV [Analytic]

Note: The non-standard function STDDEV() is provided for compatibility with other databases. It is semantically identical to STDDEV_SAMP() (page 188).

Computes the statistical sample standard deviation of the current row with respect to the group within a window. The STDDEV_SAMP() return value is the same as the square root of the variance defined for the VAR_SAMP() function:

```
STDDEV(expression) = SQRT(VAR_SAMP(expression))
```

When VAR_SAMP() returns null, this function returns null.

Behavior Type

Immutable

Syntax

```
STDDEV ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.
OVER(...)	See ***Analytic Functions***. (page 141)

Example

The following example returns the standard deviations of salaries in the employee dimension table by job title Assistant Director:

```
=> SELECT employee_last_name, annual_salary,
      STDDEV(annual_salary) OVER (ORDER BY hire_date) as "stddev"
   FROM employee_dimension
   WHERE job_title =  'Assistant Director';

 employee_last_name | annual_salary |      stddev
--------------------+---------------+------------------
 Goldberg           |         61859 |              NaN
 Miller             |         79582 | 12532.0534829692
 Goldberg           |         74236 | 9090.97147357388
 Campbell           |         66426 |  7909.9541665339
 Moore              |         66630 | 7068.30282316761
 Nguyen             |         53530 | 9154.14713486005
 Harris             |         74115 | 8773.54346886142
 Lang               |         59981 | 8609.60471031374
```

```
 Farmer              |         60597 | 8335.41158418579
 Nguyen              |         78941 | 8812.87941405456
 Smith               |         55018 |  9179.7672390773
...
```

See Also

STDDEV (page 134) and ***STDDEV_SAMP*** (page 136) aggregate functions

STDDEV_SAMP (page 188) analytic function

Using SQL Analytics in the Programmer's Guide

STDDEV_POP [Analytic]

Computes the statistical population standard deviation and returns the square root of the population variance within a window. The `STDDEV_POP()` return value is the same as the square root of the `VAR_POP()` function:

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

When `VAR_POP` returns null, this function returns null.

Behavior Type

Immutable

Syntax

```
STDDEV_POP ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Any `NUMERIC` ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.
`OVER(...)`	See ***Analytic Functions.*** (page 141)

Examples

The following example returns the population standard deviations of salaries in the employee dimension table by job title Assistant Director:

```
=> SELECT employee_last_name, annual_salary,
      STDDEV_POP(annual_salary) OVER (ORDER BY hire_date) as "stddev_pop"
   FROM employee_dimension WHERE job_title =  'Assistant Director';

 employee_last_name | annual_salary |    stddev_pop
--------------------+---------------+------------------
```

```
 Goldberg           |         61859 |                0
 Miller             |         79582 |           8861.5
 Goldberg           |         74236 | 7422.74712548456
 Campbell           |         66426 | 6850.22125098891
 Moore              |         66630 | 6322.08223926257
 Nguyen             |         53530 | 8356.55480080699
 Harris             |         74115 | 8122.72288970008
 Lang               |         59981 | 8053.54776538731
 Farmer             |         60597 | 7858.70140687825
 Nguyen             |         78941 | 8360.63150784682
```

See Also

STDDEV_POP (page 135) aggregate functions

Using SQL Analytics in the Programmer's Guide

STDDEV_SAMP [Analytic]

Computes the statistical sample standard deviation of the current row with respect to the group within a window. The `STDDEV_SAMP()` return value is the same as the square root of the variance defined for the `VAR_SAMP()` function:

```
STDDEV(expression) = SQRT(VAR_SAMP(expression))
```

When `VAR_SAMP()` returns null, this function returns null.

Behavior Type

Immutable

Syntax

```
STDDEV_SAMP ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Any `NUMERIC` ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument..
`OVER(...)`	See ***Analytic Functions***. (page 141)

Notes

`STDDEV_SAMP()` is semantically identical to the non-standard function, `STDDEV()` (page 134).

Examples

The following example returns the sample standard deviations of salaries in the `employee` dimension table by job title Assistant Director:

```
=> SELECT employee_last_name, annual_salary,
      STDDEV(annual_salary) OVER (ORDER BY hire_date) as "stddev_samp"
      FROM employee_dimension WHERE job_title =  'Assistant Director';

 employee_last_name | annual_salary |   stddev_samp
--------------------+---------------+-----------------
 Goldberg           |         61859 |             NaN
 Miller             |         79582 | 12532.0534829692
 Goldberg           |         74236 | 9090.97147357388
 Campbell           |         66426 |  7909.9541665339
 Moore              |         66630 | 7068.30282316761
 Nguyen             |         53530 | 9154.14713486005
 Harris             |         74115 | 8773.54346886142
 Lang               |         59981 | 8609.60471031374
 Farmer             |         60597 | 8335.41158418579
 Nguyen             |         78941 | 8812.87941405456
...
```

See Also

Analytic Functions (page 141)

STDDEV (page 186) analytic function

STDDEV (page 134) and ***STDDEV_SAMP*** (page 136) aggregate functions

Using SQL Analytics in the Programmer's Guide

SUM [Analytic]

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

Behavior Type

Immutable

Syntax

```
SUM ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.
OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- If you encounter data overflow when using SUM(), use SUM_FLOAT() (page 138) which converts data to a floating point.
- SUM() returns the sum of values of an expression.

Examples

The following query returns the cumulative sum all of the returns made to stores in January:

```
=> SELECT calendar_month_name AS month, transaction_type, sales_quantity,
      SUM(sales_quantity)
     OVER (PARTITION BY calendar_month_name ORDER BY date_dimension.date_key) AS
SUM
   FROM store.store_sales_fact JOIN date_dimension
   USING(date_key) WHERE calendar_month_name IN ('January')
   AND transaction_type= 'return';


  month   | transaction_type | sales_quantity | SUM
----------+------------------+----------------+------
 January | return           |              4 | 2338
 January | return           |              3 | 2338
 January | return           |              1 | 2338
 January | return           |              5 | 2338
 January | return           |              8 | 2338
 January | return           |              3 | 2338
 January | return           |              5 | 2338
 January | return           |             10 | 2338
 January | return           |              9 | 2338
 January | return           |             10 | 2338
(10 rows)
```

See Also

SUM (page 137) aggregate function

Numeric Data Types (page 103)

Using SQL Analytics in the Programmer's Guide

VAR_POP [Analytic]

Returns the statistical population variance of a non-null set of numbers (nulls are ignored) in a group within a window. Results are calculated by the sum of squares of the difference of *expression* from the mean of *expression*, divided by the number of rows remaining:

```
(SUM(expression*expression) - SUM(expression)*SUM(expression) /
  COUNT(expression)) / COUNT(expression)
```

Behavior Type

Immutable

Syntax

```
VAR_POP ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument
OVER(...)	See ***Analytic Functions***. (page 141)

Examples

The following example calculates the cumulative population in the store orders fact table of sales in December 2007:

```
=> SELECT date_ordered,
      VAR_POP(SUM(total_order_cost))
      OVER (ORDER BY date_ordered) "var_pop"
   FROM store.store_orders_fact s
   WHERE date_ordered BETWEEN '2007-12-01' AND '2007-12-31'
   GROUP BY s.date_ordered;

 date_ordered |     var_pop
--------------+-----------------
 2007-12-01   |               0
 2007-12-02   |       1129564881
 2007-12-03   | 1206008121.55542
 2007-12-04   | 26353624176.1875
 2007-12-05   | 21315288023.4402
 2007-12-06   | 21619271028.3333
 2007-12-07   | 19867030477.6328
 2007-12-08   |    19197735288.5
 2007-12-09   | 19100157155.2097
```

```
 2007-12-10   | 19369222968.0896
(10 rows)
```

See Also

VAR_POP (page 139) aggregate function

Using SQL Analytics in the Programmer's Guide

VAR_SAMP [Analytic]

Returns the sample variance of a non-null set of numbers (nulls in the set are ignored) for each row of the group within a window. Results are calculated by the sum of squares of the difference of *expression* from the mean of *expression,* divided by the number of rows remaining minus 1:

```
(SUM(expression*expression) - SUM(expression)*SUM(expression) /
  COUNT(expression)) / (COUNT(expression) - 1)
```

Behavior Type

Immutable

Syntax

```
VAR_SAMP ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Any NUMERIC ***data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument
OVER(...)	See ***Analytic Functions***. (page 141)

Notes

- VAR_SAMP() returns the sample variance of a set of numbers after it discards the nulls in the set.
- If the function is applied to an empty set, then it returns null.
- This function is similar to VARIANCE(), except that given an input set of one element, VARIANCE() returns 0 and VAR_SAMP() returns null.

Examples

The following example calculates the sample variance in the store orders fact table of sales in December 2007:

```
=> SELECT date_ordered,
      VAR_SAMP(SUM(total_order_cost))
```

```
      OVER (ORDER BY date_ordered) "var_samp"
   FROM store.store_orders_fact s
   WHERE date_ordered BETWEEN '2007-12-01' AND '2007-12-31'
   GROUP BY s.date_ordered;

 date_ordered |     var_samp
--------------+------------------
 2007-12-01   |              NaN
 2007-12-02   |       2259129762
 2007-12-03   | 1809012182.33301
 2007-12-04   |   35138165568.25
 2007-12-05   | 26644110029.3003
 2007-12-06   |      25943125234
 2007-12-07   | 23178202223.9048
 2007-12-08   | 21940268901.1431
 2007-12-09   | 21487676799.6108
 2007-12-10   | 21521358853.4331
(10 rows)
```

See Also

VARIANCE (page 193) analytic function

VAR_SAMP (page 139) aggregate function

Using SQL Analytics in the Programmer's Guide

VARIANCE [Analytic]

Note: The non-standard function `VARIANCE()` is provided for compatibility with other databases. It is semantically identical to `VAR_SAMP()` (page 192).

Returns the sample variance of a non-null set of numbers (nulls in the set are ignored) for each row of the group within a window. Results are calculated by the sum of squares of the difference of *expression* from the mean of *expression*, divided by the number of rows remaining minus 1:

```
(SUM(expression*expression) - SUM(expression)*SUM(expression) /
  COUNT(expression)) / (COUNT(expression) - 1)
```

Behavior Type

Immutable

Syntax

```
VAR_SAMP ( expression ) OVER (
... [ window_partition_clause (page 143) ]
... [ window_order_clause (page 144) ]
... [ window_frame_clause (page 145) ] )
```

Parameters

expression	Any ***NUMERIC data type*** (page 103) or any non-numeric data type that can be implicitly converted to a numeric data

	type. The function returns the same data type as the numeric data type of the argument.
`OVER(...)`	See ***Analytic Functions***. (page 141)

Notes

- `VARIANCE()` returns the variance of *expression*.
- The variance of expression is calculated as follows:
  - 0 if the number of rows in *expression* = 1
  - `VAR_SAMP()` if the number of rows in *expression* > 1

Examples

The following example calculates the cumulative variance in the store orders fact table of sales in December 2007:

```
=> SELECT date_ordered,
      VARIANCE(SUM(total_order_cost))
      OVER (ORDER BY date_ordered) "variance"
   FROM store.store_orders_fact s
   WHERE date_ordered BETWEEN '2007-12-01' AND '2007-12-31'
   GROUP BY s.date_ordered;

 date_ordered |     variance
--------------+------------------
 2007-12-01   |              NaN
 2007-12-02   |       2259129762
 2007-12-03   | 1809012182.33301
 2007-12-04   |   35138165568.25
 2007-12-05   | 26644110029.3003
 2007-12-06   |      25943125234
 2007-12-07   | 23178202223.9048
 2007-12-08   | 21940268901.1431
 2007-12-09   | 21487676799.6108
 2007-12-10   | 21521358853.4331
(10 rows)
```

See Also

VAR_SAMP (page 192) analytic function

VARIANCE (page 140) and ***VAR_SAMP*** (page 139) aggregate functions

Using SQL Analytics in the Programmer's Guide

Date/Time Functions

Date and time functions perform conversion, extraction, or manipulation operations on date and time data types and can return date and time information.

Usage

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'
```

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 timestamp. However, `TIMEOFDAY()` returns the wall-clock time and advances during transactions.

See Also

Template Patterns for Date/Time Formatting (page 265)

ADD_MONTHS

Takes a DATE, TIMESTAMP, or TIMESTAMPTZ argument and a number of months and returns a date. TIMESTAMPTZ arguments are implicitly cast to TIMESTAMP.

Behavior Type

Immutable if called with DATE or TIMESTAMP but stable with TIMESTAMPTZ in that its results can change based on TIMEZONE settings

Syntax

```
ADD_MONTHS ( d , n );
```

Parameters

d	Is the incoming DATE, TIMESTAMP, or TIMESTAMPTZ. If the start date falls on the last day of the month, or if the resulting month has fewer days than the given day of the month, then the result is the last day of the resulting month. Otherwise, the result has the same start day.
n	Can be any INTEGER.

Examples

The following example's results include a leap year:

```
SELECT ADD_MONTHS('31-Jan-08', 1) "Months";
   Months
------------
 2008-02-29
(1 row)
```

The next example adds four months to January and returns a date in May:

```
SELECT ADD_MONTHS('31-Jan-08', 4) "Months";
   Months
------------
 2008-05-31
(1 row)
```

This example subtracts 4 months from January, returning a date in September:

```
SELECT ADD_MONTHS('31-Jan-08', -4) "Months";
   Months
------------
 2007-09-30
(1 row)
```

Because the following example specifies NULL, the result set is empty:

```
SELECT ADD_MONTHS('31-Jan-03', NULL) "Months";
 Months
--------

(1 row)
```

This example provides no date argument, so even though the number of months specified is 1, the result set is empty:

```
SELECT ADD_MONTHS(NULL, 1) "Months";
 Months
--------

(1 row)
```

In this example, the date field defaults to a timestamp, so the PST is ignored. Notice that even though it is already the next day in Pacific time, the result falls on the same date in New York (two years later):

```
SET TIME ZONE 'America/New_York';
SELECT ADD_MONTHS('2008-02-29 23:30 PST', 24);
 add_months
------------
 2010-02-28
(1 row)
```

This example specifies a timestamp with time zone, so the PST is taken into account:

```
SET TIME ZONE 'America/New_York';
SELECT ADD_MONTHS('2008-02-29 23:30 PST'::TIMESTAMPTZ, 24);
 add_months
------------
 2010-03-01
(1 row)
```

AGE_IN_MONTHS

Returns an INTEGER value representing the difference in months between two TIMESTAMP, DATE or TIMESTAMPTZ values.

Behavior Type

Stable if second argument is omitted or if either argument is TIMESTAMPTZ. Immutable otherwise.

Syntax

```
AGE_IN_MONTHS ( expression1 [ , expression2 ] )
```

Parameters

expression1	specifies the beginning of the period.
expression2	specifies the end of the period. The default is the ***CURRENT_DATE*** (page 200).

Notes

The inputs can be TIMESTAMP, TIMESTAMPTZ, or DATE.

Examples

The following example returns the age in months of a person born on March 2, 1972 on the date June 21, 1990, with a time elapse of 18 years, 3 months, and 19 days:

```
SELECT AGE_IN_MONTHS(TIMESTAMP '1990-06-21', TIMESTAMP '1972-03-02');
 AGE_IN_MONTHS
---------------
```

```
           219
(1 row)
```

The next example shows the age in months of the same person (born March 2, 1972) as of March 16, 2010:

```
SELECT AGE_IN_MONTHS(TIMESTAMP 'March 16, 2010', TIMESTAMP '1972-03-02');
  AGE_IN_MONTHS
---------------
            456
(1 row)
```

This example returns the age in months of a person born on November 21, 1939:

```
SELECT AGE_IN_MONTHS(TIMESTAMP '1939-11-21');
 AGE_IN_MONTHS
---------------
           844
(1 row)
```

In the above form, the result changes as time goes by.

See Also

AGE_IN_YEARS (page 198)

INTERVAL (page 81)

AGE_IN_YEARS

Returns an INTEGER value representing the difference in years between two TIMESTAMP, DATE or TIMESTAMPTZ values.

Behavior Type

Stable if second argument is omitted or if either argument is TIMESTAMPTZ. Immutable otherwise.

Syntax

```
AGE_IN_YEARS ( expression1 [ , expression2 ] )
```

Parameters

expression1	specifies the beginning of the period.
expression2	specifies the end of the period. The default is the ***CURRENT_DATE*** (page 200).

Notes

- The AGE_IN_YEARS() function was previously called AGE. AGE() is not supported.
- Inputs can be TIMESTAMP, TIMESTAMPTZ, or DATE.

Examples

The following example returns the age in years of a person born on March 2, 1972 on the date June 21, 1990, with a time elapse of 18 years, 3 months, and 19 days:

```
SELECT AGE_IN_YEARS(TIMESTAMP '1990-06-21', TIMESTAMP '1972-03-02');
 AGE_IN_YEARS
--------------
           18
(1 row)
```

The next example shows the age in years of the same person (born March 2, 1972) as of February 24, 2009:

```
SELECT AGE_IN_YEARS(TIMESTAMP '2009-02-24', TIMESTAMP '1972-03-02');
 AGE_IN_YEARS
--------------
           36
(1 row)
```

This example returns the age in years of a person born on November 21, 1939:

```
SELECT AGE_IN_YEARS(TIMESTAMP '1939-11-21');
 AGE_IN_YEARS
--------------
           70
(1 row)
```

See Also

AGE_IN_MONTHS (page 197)

INTERVAL (page 81)

CLOCK_TIMESTAMP

Returns a value of type TIMESTAMP WITH TIMEZONE representing the current system-clock time.

Behavior Type

Volatile

Syntax

```
CLOCK_TIMESTAMP()
```

Notes

This function uses the date and time supplied by the operating system on the server to which you are connected, which should be the same across all servers. The value changes each time you call it.

Examples

The following command returns the current time on your system:

```
SELECT CLOCK_TIMESTAMP() "Current Time";
         Current Time
------------------------------
 2010-09-23 11:41:23.33772-04
(1 row)
```

Each time you call the function, you get a different result. The difference in this example is in microseconds:

```
SELECT CLOCK_TIMESTAMP() "Time 1", CLOCK_TIMESTAMP() "Time 2";
            Time 1              |             Time 2
--------------------------------+-------------------------------
 2010-09-23 11:41:55.369201-04 | 2010-09-23 11:41:55.369202-04
(1 row)
```

See Also

STATEMENT_TIMESTAMP (page 239)

TRANSACTION_TIMESTAMP (page 251)

CURRENT_DATE

Returns the date (date-type value) on which the current transaction started.

Behavior Type

Stable

Syntax

```
CURRENT_DATE
```

Notes

The CURRENT_DATE function does not require parentheses.

Examples

```
SELECT CURRENT_DATE;
  ?column?
------------
 2010-09-23
(1 row)
```

CURRENT_TIME

Returns a value of type TIME WITH TIMEZONE representing the time of day.

Behavior Type

Stable

Syntax

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

Parameters

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 timestamp.
- The CURRENT_TIME function does not require parentheses.

Examples

```
SELECT CURRENT_TIME "Current Time";
    Current Time
--------------------
 12:45:12.186089-05
(1 row)
```

CURRENT_TIMESTAMP

Returns a value of type TIMESTAMP WITH TIME ZONE representing the start of the current transaction.

Behavior Type

Stable

Syntax

```
CURRENT_TIMESTAMP [ ( precision ) ]
```

Parameters

precision	(INTEGER) causes the result to be rounded to the specified number of fractional digits in the seconds field. Range of INTEGER is 0-6.

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 timestamp.

Examples

```
SELECT CURRENT_TIMESTAMP;
           ?column?
-------------------------------
 2010-09-23 11:37:22.354823-04
```

```
(1 row)
SELECT CURRENT_TIMESTAMP(2);
         ?column?
---------------------------
 2010-09-23 11:37:22.35-04
(1 row)
```

DATE_PART

Is modeled on the traditional Ingres equivalent to the SQL-standard function EXTRACT. Internally DATE_PART is used by the EXTRACT function.

Behavior Type

Stable when source is of type TIMESTAMPTZ, Immutable otherwise.

Syntax

```
DATE_PART ( field , source )
```

Parameters

field	Is a single-quoted string value that specifies the field to extract. You must enter the constant field values (i.e. CENTURY, DAY, etc). when specifying the field. **Note:** The *field* parameter values are the same for the ***EXTRACT*** (page 218) function.
source	Is a ***date/time*** (page 78) expression

Field Values

CENTURY	The century number.
	`SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');`
	Result: 20
	`SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 21
	The first century starts at 0001-01-01 00:00:00 AD. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 to 1.

DAY	The day (of the month) field (1 - 31).
	`SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 16
	`SELECT EXTRACT(DAY FROM DATE '2001-02-16');`
	Result: 16

DECADE	The year field divided by 10.
	`SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 200
	`SELECT EXTRACT(DECADE FROM DATE '2001-02-16');`
	Result: 200

DOQ	The day within the current quarter. `SELECT EXTRACT(DOQ FROM CURRENT_DATE);` *Result*: 89 The result is calculated as follows: Current date = June 28, current quarter = 2 (April, May, June). 30 (April) + 31 (May) + 28 (June current day) = 89. DOQ recognizes leap year days.

DOW	The day of the week (0 - 6; Sunday is 0).
	`SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 5
	`SELECT EXTRACT(DOW FROM DATE '2001-02-16');`
	Result: 5
	Note that EXTRACT's day of the week numbering is different from that of the TO_CHAR function.

DOY	The day of the year (1 - 365/366)
	`SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 47
	`SELECT EXTRACT(DOY FROM DATE '2001-02-16');`
	Result: 5

EPOCH	For DATE and TIMESTAMP values, the number of seconds since 1970-01-01 00:00:00-00 (can be negative); for INTERVAL values, the total number of seconds in the interval.
	`SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-08');`
	Result: 982384720
	`SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');`
	Result: 442800
	Here is how you can convert an epoch value back to a timestamp:
	`SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720 * INTERVAL '1 second';`

HOUR	The hour field (0 - 23).
	`SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 20
	`SELECT EXTRACT(HOUR FROM TIME '13:45:59');`
	Result: 13

ISODOW	The ISO day of the week (1 - 7; Monday is 1). `SELECT EXTRACT(ISODOW FROM DATE '2010-09-27');` *Result*: 1
ISOWEEK	The ISO week, which consists of 7 days starting on Monday and ending on Sunday. The first week of the year is the week that contains January 4.

ISOYEAR	The ISO year, which is 52 or 53 weeks (Monday - Sunday).
	`SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');`
	Result: 2005
	`SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');`
	Result: 2006
	`SELECT EXTRACT(ISOYEAR FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 2001

MICROSECONDS	The seconds field, including fractional parts, multiplied by 1,000,000. This includes full seconds. `SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');` *Result*: 28500000

MILLENNIUM	The millennium number. `SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');` *Result*: 3 Years in the 1900s are in the second millennium. The third millennium starts January 1, 2001.
MILLISECONDS	The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds. `SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');` *Result*: 28500

MINUTE	The minutes field (0 - 59).
	`SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 38
	`SELECT EXTRACT(MINUTE FROM TIME '13:45:59');`
	Result: 45

MONTH	For `timestamp` values, the number of the month within the year (1 - 12) ; for `interval` values the number of months, modulo 12 (0 - 11).
	`SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 2
	`SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');`
	Result: 3
	`SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');`
	Result: 1

QUARTER	The quarter of the year (1 - 4) that the day is in (for `timestamp` values only). `SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');` *Result*: 1

SECOND	The seconds field, including fractional parts (0 - 59) (60 if leap seconds are implemented by the operating system).
	`SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 40
	`SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');`
	Result: 28.5

TIME ZONE	The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC.
TIMEZONE_HOUR	The hour component of the time zone offset.
TIMEZONE_MINUTE	The minute component of the time zone offset.

WEEK	The number of the week of the calendar year that the day is in.
	`SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 7
	`SELECT EXTRACT(WEEK FROM DATE '2001-02-16');`
	Result: 7

YEAR	The year field. Keep in mind there is no `0 AD`, so subtract `BC` years from `AD` years with care. `SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');` *Result:* 2001

Examples

The following example extracts the day value from the input parameters:

```
SELECT DATE_PART('day', TIMESTAMP '2009-02-24 20:38:40') "Day";
 Day
-----
```

```
  24
(1 row)
```

The following example extracts the month value from the input parameters:

```
SELECT DATE_PART('month', TIMESTAMP '2009-02-24 20:38:40') "Month";
 Month
-------
     2
(1 row)
```

The following example extracts the year value from the input parameters:

```
SELECT DATE_PART('year', TIMESTAMP '2009-02-24 20:38:40') "Year";
 Year
------
 2009
(1 row)
```

The following example extracts the hours from the input parameters:

```
SELECT DATE_PART('hour', TIMESTAMP '2009-02-24 20:38:40') "Hour";
 Hour
------
   20
(1 row)
```

The following example extracts the minutes from the input parameters:

```
SELECT DATE_PART('minutes', TIMESTAMP '2009-02-24 20:38:40') "Minutes";
 Minutes
---------
      38
(1 row)
```

The following example extracts the seconds from the input parameters:

```
SELECT DATE_PART('seconds', TIMESTAMP '2009-02-24 20:38:40') "Seconds";
 Seconds
---------
      40
(1 row)
```

The following example extracts the day of quarter (DOQ) from the input parameters:

```
SELECT DATE_PART('DOQ', TIMESTAMP '2009-02-24 20:38:40') "DOQ";
 DOQ
-----
  55
(1 row)

SELECT DATE_PART('day', INTERVAL '29 days 23 hours');
 date_part
-----------
        29
(1 row)
```

Notice what happens to the above query if you add an hour:

```
SELECT DATE_PART('day', INTERVAL '29 days 24 hours');
```

```
 date_part
-----------
        30
(1 row)
```

The following example returns 0 because an interval in hours is up to 24 only:

```
SELECT DATE_PART('hour', INTERVAL '24 hours 45 minutes');
 date_part
-----------
         0
(1 row)
```

See Also

EXTRACT (page 218)

DATE

Converts a TIMESTAMP, TIMESTAMPTZ, DATE, or VARCHAR to a DATE. You can also use this function to convert an INTEGER to a DATE. In this case, the resulting date reflects the *int* number of days after 0001 AD. (Day 1 is January 1, 0001.)

Syntax

```
DATE ( d | n )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the TIMESTAMP, TIMESTAMPTZ, VARCHAR, or DATE input value.
n	Is the integer you want to convert to a DATE.

Example

```
=> SELECT DATE (1);
    DATE
------------
 0001-01-01
(1 row)


=> SELECT DATE (734260);
    DATE
------------
 2011-05-03
(1 row)
```

```
=> SELECT DATE ('TODAY');
    DATE
------------
 2011-05-31
(1 row)
```

DATE_TRUNC

Truncates date and time values as indicated. The return value is of type TIME or TIMETZ with all fields that are less significant than the selected one set to zero (or one, for day and month).

Behavior Type

Stable.

Syntax

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

Parameters

field	Is a string constant that selects the precision to which truncate the input value.
source	Is a value expression of type `TIME` or `TIMETZ`.

Field Values

CENTURY	The century number. The first century starts at 0001-01-01 00:00:00 AD. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 to 1.
DAY	The day (of the month) field (1 - 31).
DECADE	The year field divided by 10.
HOUR	The hour field (0 - 23).
MICROSECONDS	The seconds field, including fractional parts, multiplied by 1,000,000. This includes full seconds.
MILLENNIUM	The millennium number. Years in the 1900s are in the second millennium. The third millennium starts January 1, 2001.
MILLISECONDS	The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.
MINUTE	The minutes field (0 - 59).
MONTH	For `timestamp` values, the number of the month within the year (1 - 12) ; for `interval` values the number of months, modulo 12 (0 - 11).

SECOND	The seconds field, including fractional parts (0 - 59) (60 if leap seconds are implemented by the operating system).
WEEK	The number of the week of the year that the day is in. By definition, the ISO-8601 week starts on Monday, and the first week of a year contains January 4 of that year. In other words, the first Thursday of a year is in week 1 of that year. Because of this, it is possible for early January dates to be part of the 52nd or 53rd week of the previous year. For example, `2005-01-01` is part of the 53rd week of year 2004, and `2006-01-01` is part of the 52nd week of year 2005.
YEAR	The year field. Keep in mind there is no `0 AD`, so subtract `BC` years from `AD` years with care.

Examples

The following example sets the field value as hour and returns the hour, truncating the minutes and seconds:

```
VMart=> select date_trunc('hour', timestamp '2012-02-24 13:38:40') as hour;
        hour
---------------------
 2012-02-24 13:00:00
(1 row)
```

The following example returns the year from the input `timestamptz '2012-02-24 13:38:40'`. The function also defaults the month and day to January 1, truncates the hour:minute:second of the timestamp, and appends the time zone (`-05`):

```
VMart=> select date_trunc('year', timestamptz '2012-02-24 13:38:40') as year;
          year
------------------------
 2012-01-01 00:00:00-05
(1 row)
```

The following example returns the year and month and defaults day of month to 1, truncating the rest of the string:

```
VMart=> select date_trunc('month', timestamp '2012-02-24 13:38:40') as year;
        year
---------------------
 2012-02-01 00:00:00
(1 row)
```

DATEDIFF

Returns the difference between two date or time values, based on the specified start and end arguments.

Behavior Type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Syntax 1

```
DATEDIFF ( datepart , startdate , enddate );
```

Syntax 2

```
DATEDIFF ( datepart , starttime , endtime );
```

Parameters

datepart	Returns the number of specified datepart boundaries between the specified startdate and enddate. Can be an unquoted identifier, a quoted string, or an expression in parentheses, which evaluates to the datepart as a character string. The following table lists the valid *datepart* arguments. datepart — abbreviation year — yy, yyyy quarter — qq, q month — mm, m day — dd, d, dy, dayofyear, y week — wk, ww hour — hh minute — mi, n second — ss, s millisecond — ms microsecond — mcs, us
startdate	Is the start date for the calculation and is an expression that returns a ***TIMESTAMP*** (page 97), ***DATE*** (page 80), or TIMESTAMPTZ value. The *startdate* value is not included in the count.
enddate	Is the end date for the calculation and is an expression that returns a ***TIMESTAMP*** (page 97), ***DATE*** (page 80), or TIMESTAMPTZ value. The *enddate* value is included in the count.

starttime	Is the start time for the calculation and is an expression that returns an ***INTERVAL*** (page 81) or ***TIME*** (page 95) data type. ▪ The *starttime* value is not included in the count. ▪ Year, quarter, or month *dateparts* are not allowed.
endtime	Is the end time for the calculation and is an expression that returns an ***INTERVAL*** (page 81) or ***TIME*** (page 95) data type. ▪ The *endtime* value is included in the count. ▪ Year, quarter, or month *dateparts* are not allowed.

Notes

- DATEDIFF() is an immutable function with a default type of TIMESTAMP. It also takes DATE. If TIMESTAMPTZ is specified, the function is stable.
- HP Vertica accepts statements written in any of the following forms:

  ```
  DATEDIFF(year, s, e);
  DATEDIFF('year', s, e);
  ```

  If you use an expression, the expression must be enclosed in parentheses:

  ```
  DATEDIFF((expression), s, e);
  ```

- Starting arguments are not included in the count, but end arguments are included.

The datepart boundaries

DATEDIFF calculates results according to ticks—or boundaries—within the date range or time range. Results are calculated based on the specified *datepart*. Let's examine the following statement and its results:

```
SELECT DATEDIFF('year', TO_DATE('01-01-2005','MM-DD-YYYY'),
TO_DATE('12-31-2008','MM-DD-YYYY'));
 datediff
----------
        3
(1 row)
```

In the above example, we specified a *datepart* of year, a *startdate* of January 1, 2005 and an *enddate* of December 31, 2008. DATEDIFF returns 3 by counting the year intervals as follows:

```
 [1] January 1, 2006 + [2] January 1, 2007 + [3] January 1, 2008 = 3
```

The function returns 3, and not 4, because *startdate* (January 1, 2005) is not counted in the calculation. DATEDIFF also ignores the months between January 1, 2008 and December 31, 2008 because the *datepart* specified is year and only the start of each year is counted.

Sometimes the *enddate* occurs earlier in the ending year than the *startdate* in the starting year. For example, assume a *datepart* of year, a *startdate* of August 15, 2005, and an *enddate* of January 1, 2009. In this scenario, less than three years have elapsed, but DATEDIFF counts the same way it did in the previous example, returning 3 because it returns the number of January 1s between the limits:

```
 [1] January 1, 2006 + [2] January 1, 2007 + [3] January 1, 2008 = 3
```

In the following query, HP Vertica recognizes the full year 2005 as the starting year and 2009 as the ending year.

```
SELECT DATEDIFF('year', TO_DATE('08-15-2005','MM-DD-YYYY'),
TO_DATE('01-01-2009','MM-DD-YYYY'));
```

The count occurs as follows:

```
 [1] January 1, 2006 + [2] January 1, 2007 + [3] January 1, 2008 + [4] January 1,
2009 = 4
```

Even though August 15 has not yet occurred in the *enddate*, the function counts the entire *enddate* year as one tick or boundary because of the year *datepart*.

Examples

Year: In this example, the *startdate* and *enddate* are adjacent. The difference between the dates is one time boundary (second) of its datepart, so the result set is 1.

```
SELECT DATEDIFF('year', TIMESTAMP '2008-12-31 23:59:59',
  '2009-01-01 00:00:00');
 datediff
----------
        1
(1 row)
```

Quarters start on January, April, July, and October.

In the following example, the result is 0 because the difference from January to February in the same calendar year does not span a quarter:

```
SELECT DATEDIFF('qq', TO_DATE('01-01-1995','MM-DD-YYYY'),
  TO_DATE('02-02-1995','MM-DD-YYYY'));
 datediff
----------
         0
(1 row)
```

The next example, however, returns 8 quarters because the difference spans two full years. The extra month is ignored:

```
SELECT DATEDIFF('quarter', TO_DATE('01-01-1993','MM-DD-YYYY'),
  TO_DATE('02-02-1995','MM-DD-YYYY'));
 datediff
----------
         8
(1 row)
```

Months are based on real calendar months.

The following statement returns 1 because there is month difference between January and February in the same calendar year:

```
SELECT DATEDIFF('mm', TO_DATE('01-01-2005','MM-DD-YYYY'),
  TO_DATE('02-02-2005','MM-DD-YYYY'));
 datediff
----------
         1
(1 row)
```

The next example returns a negative value of 1:

```
SELECT DATEDIFF('month', TO_DATE('02-02-1995','MM-DD-YYYY'),
```

```
  TO_DATE('01-01-1995','MM-DD-YYYY'));
 datediff
----------
       -1
(1 row)
```

And this third example returns 23 because there are 23 months difference between

```
SELECT DATEDIFF('m', TO_DATE('02-02-1993','MM-DD-YYYY'),
  TO_DATE('01-01-1995','MM-DD-YYYY'));
 datediff
----------
       23
(1 row)
```

Weeks start on Sunday at midnight.

The first example returns 0 because, even though the week starts on a Sunday, it is not a full calendar week:

```
SELECT DATEDIFF('ww', TO_DATE('02-22-2009','MM-DD-YYYY'),
  TO_DATE('02-28-2009','MM-DD-YYYY'));
 datediff
----------
        0
(1 row)
```

The following example returns 1 (week); January 1, 2000 fell on a Saturday.

```
SELECT DATEDIFF('week', TO_DATE('01-01-2000','MM-DD-YYYY'),
  TO_DATE('01-02-2000','MM-DD-YYYY'));
 datediff
----------
        1
(1 row)
```

In the next example, DATEDIFF() counts the weeks between January 1, 1995 and February 2, 1995 and returns 4 (weeks):

```
SELECT DATEDIFF('wk', TO_DATE('01-01-1995','MM-DD-YYYY'),
  TO_DATE('02-02-1995','MM-DD-YYYY'));
 datediff
----------
        4
(1 row)
```

The next example returns a difference of 100 weeks:

```
SELECT DATEDIFF('ww', TO_DATE('02-02-2006','MM-DD-YYYY'),
  TO_DATE('01-01-2008','MM-DD-YYYY'));
 datediff
----------
      100
(1 row)
```

Days are based on real calendar days.

The first example returns 31, the full number of days in the month of July 2008.

```
SELECT DATEDIFF('day', 'July 1, 2008', 'Aug 1, 2008'::date);
 datediff
----------
       31
(1 row)
```

Just over two years of days:

```
SELECT DATEDIFF('d', TO_TIMESTAMP('01-01-1993','MM-DD-YYYY'),
  TO_TIMESTAMP('02-02-1995','MM-DD-YYYY'));
 datediff
----------
      762
(1 row)
```

Hours, minutes, and **seconds** are based on clock time.

The first example counts backwards from March 2 to February 14 and returns -384 hours:

```
SELECT DATEDIFF('hour', TO_DATE('03-02-2009','MM-DD-YYYY'),
  TO_DATE('02-14-2009','MM-DD-YYYY'));
 datediff
----------
     -384
(1 row)
```

Another hours example:

```
SELECT DATEDIFF('hh', TO_TIMESTAMP('01-01-1993','MM-DD-YYYY'),
  TO_TIMESTAMP('02-02-1995','MM-DD-YYYY'));
 datediff
----------
    18288
(1 row)
```

This example counts the minutes backwards:

```
SELECT DATEDIFF('mi', TO_TIMESTAMP('01-01-1993 03:00:45','MM-DD-YYYY HH:MI:SS'),
  TO_TIMESTAMP('01-01-1993 01:30:21',' MM-DD-YYYY  HH:MI:SS'));
 datediff
----------
      -90
(1 row)
```

And this example counts the minutes forward:

```
SELECT DATEDIFF('minute', TO_DATE('01-01-1993','MM-DD-YYYY'),
  TO_DATE('02-02-1995','MM-DD-YYYY'));
 datediff
----------
  1097280
(1 row)
```

In the following example, the query counts the difference in seconds, beginning at a start time of 4:44 and ending at 5:55 with an interval of 2 days:

```
SELECT DATEDIFF('ss', TIME '04:44:42.315786',
  INTERVAL '2 05:55:52.963558');
 datediff
```

```
----------
   177070
(1 row)
```

See Also

Date/Time Expressions (page 55)

DAY

Extracts the day of the month from a TIMESTAMP, TIMESTAMPTZ, INTEGER, VARCHAR or INTERVAL input value. The return value is of type INTEGER.

Syntax

```
DAY (d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the TIMESTAMP, TIMESTAMPTZ, INTERVAL, VARCHAR, or INTEGER input value.

Examples

```
=> SELECT DAY (6);
 DAY
-----
   6
(1 row)

=> SELECT DAY(TIMESTAMP 'sep 22, 2011 12:34');
 DAY
-----
  22
(1 row)

=> SELECT DAY('sep 22, 2011 12:34');
 DAY
-----
  22
(1 row)

=> SELECT DAY(INTERVAL '35 12:34');
 DAY
-----
  35
(1 row)
```

DAYOFMONTH

Returns an integer representing the day of the month based on a VARCHAR, DATE, TIMESTAMP, OR TIMESTAMPTZ input value.

Syntax

```
DAYOFMONTH ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, DATE, TIMESTAMP, or TIMESTAMPTZ input value.

Example

```
=> SELECT DAYOFMONTH (TIMESTAMP 'sep 22, 2011 12:34');
 DAYOFMONTH
------------
         22
(1 row)
```

DAYOFWEEK

Returns an INTEGER representing the day of the week based on a TIMESTAMP, TIMESTAMPTZ, VARCHAR, or DATE input value. Valid return values are:

Integer	Week Day
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

Syntax

```
DAYOFWEEK ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the TIMESTAMP, TIMESTAMPTZ, VARCHAR, or DATE input value.

Example

```
=> SELECT DAYOFWEEK (TIMESTAMP 'sep 17, 2011 12:34');
 DAYOFWEEK
-----------
         7
(1 row)
```

DAYOFWEEK_ISO

Returns an INTEGER representing the ISO 8061 day of the week based on a VARCHAR, DATE, TIMESTAMP or TIMESTAMPTZ input value. Valid return values are:

Integer	Week Day
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

Syntax

```
DAYOFWEEK_ISO ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, DATE, TIMESTAMP, or TIMESTAMPTZ input value.

Example

```
=> SELECT DAYOFWEEK_ISO(TIMESTAMP 'Sep 22, 2011 12:34');
 DAYOFWEEK_ISO
---------------
             4
(1 row)
```

The following example shows how to combine the DAYOFWEEK_ISO, WEEK_ISO, and YEAR_ISO functions to find the ISO day of the week, week, and year:

```
=> SELECT DAYOFWEEK_ISO('Jan 1, 2000'), WEEK_ISO('Jan 1,
2000'),YEAR_ISO('Jan1,2000');
 DAYOFWEEK_ISO | WEEK_ISO | YEAR_ISO
---------------+----------+----------
             6 |       52 |     1999
(1 row)
```

See Also

WEEK_ISO (page 253)

DAYOFWEEK_ISO (page 216)

http://en.wikipedia.org/wiki/ISO_8601 (***http://en.wikipedia.org/wiki/ISO_8601***)

DAYOFYEAR

Returns an INTEGER representing the day of the year based on a TIMESTAMP, TIMESTAMPTZ , VARCHAR, or DATE input value. (January 1 is day 1.)

Syntax

```
DAYOFYEAR ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the TIMESTAMP, TIMESTAMPTZ, VARCHAR, OR DATE input value.

Example

```
=> SELECT DAYOFYEAR (TIMESTAMP 'SEPT 22,2011 12:34');
 DAYOFYEAR
-----------
       265
(1 row)
```

DAYS

Converts a DATE, VARCHAR, TIMESTAMP, or TIMESTAMPTZ to an INTEGER, reflecting the number of days after 0001 AD.

Syntax

```
DAYS( DATE d )
```

Behavior type

```
Immutable
```

Parameters

DATE *d*	Is the VARCHAR, DATE, TIMESTAMP, or TIMESTAMPTZ input value.

Example

```
=> SELECT DAYS (DATE '2011-01-22');
  DAYS
--------
 734159
(1 row)

=> SELECT DAYS ('1999-12-31');
  DAYS
--------
 730119
(1 row)
```

EXTRACT

Retrieves subfields such as year or hour from date/time values and returns values of type ***NUMERIC*** (page 107). EXTRACT is primarily intended for computational processing, rather than for formatting date/time values for display.

Internally EXTRACT uses the DATE_PART function.

Behavior Type

Stable when source is of type TIMESTAMPTZ, Immutable otherwise.

Syntax

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

Parameters

field	Is an identifier or string that selects what field to extract from the source value. You must enter the constant field values (i.e. CENTURY, DAY, etc). when specifying the field. **Note:** The field parameter is the same for the DATE_PART() (page 202) function.
source	Is an expression of type DATE, TIMESTAMP, TIME, or INTERVAL. **Note:** Expressions of type DATE are cast to TIMESTAMP.

Field Values

CENTURY	The century number.
	`SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');`
	Result: 20
	`SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 21
	The first century starts at 0001-01-01 00:00:00 AD. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 to 1.

DAY	The day (of the month) field (1 - 31).
	`SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 16
	`SELECT EXTRACT(DAY FROM DATE '2001-02-16');`
	Result: 16

DECADE	The year field divided by 10.
	`SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 200
	`SELECT EXTRACT(DECADE FROM DATE '2001-02-16');`
	Result: 200

DOQ	The day within the current quarter. `SELECT EXTRACT(DOQ FROM CURRENT_DATE);` *Result*: 89 The result is calculated as follows: Current date = June 28, current quarter = 2 (April, May, June). 30 (April) + 31 (May) + 28 (June current day) = 89. DOQ recognizes leap year days.

DOW	The day of the week (0 - 6; Sunday is 0).
	`SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 5
	`SELECT EXTRACT(DOW FROM DATE '2001-02-16');`
	Result: 5
	Note that EXTRACT's day of the week numbering is different from that of the TO_CHAR function.

DOY	The day of the year (1 - 365/366)
	`SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 47
	`SELECT EXTRACT(DOY FROM DATE '2001-02-16');`
	Result: 5

EPOCH	For DATE and TIMESTAMP values, the number of seconds since 1970-01-01 00:00:00-00 (can be negative); for INTERVAL values, the total number of seconds in the interval.
	`SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-08');`
	Result: 982384720
	`SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');`
	Result: 442800
	Here is how you can convert an epoch value back to a timestamp:
	`SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720 * INTERVAL '1 second';`

HOUR	The hour field (0 - 23).
	`SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 20
	`SELECT EXTRACT(HOUR FROM TIME '13:45:59');`
	Result: 13

ISODOW	The ISO day of the week (1 - 7; Monday is 1). `SELECT EXTRACT(ISODOW FROM DATE '2010-09-27');` *Result*: 1
ISOWEEK	The ISO week, which consists of 7 days starting on Monday and ending on Sunday. The first week of the year is the week that contains January 4.

ISOYEAR	The ISO year, which is 52 or 53 weeks (Monday - Sunday).
	`SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');`
	Result: 2005
	`SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');`
	Result: 2006
	`SELECT EXTRACT(ISOYEAR FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 2001

MICROSECONDS	The seconds field, including fractional parts, multiplied by 1,000,000. This includes full seconds. `SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');` *Result*: 28500000
MILLENNIUM	The millennium number. `SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');` *Result*: 3 Years in the 1900s are in the second millennium. The third millennium starts January 1, 2001.
MILLISECONDS	The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds. `SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');` *Result*: 28500

MINUTE	The minutes field (0 - 59).
	`SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 38
	`SELECT EXTRACT(MINUTE FROM TIME '13:45:59');`
	Result: 45

MONTH	For timestamp values, the number of the month within the year (1 - 12) ; for interval values the number of months, modulo 12 (0 - 11).
	`SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 2
	`SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');`
	Result: 3
	`SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');`
	Result: 1

QUARTER	The quarter of the year (1 - 4) that the day is in (for `timestamp` values only). `SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');` *Result*: 1

SECOND	The seconds field, including fractional parts (0 - 59) (60 if leap seconds are implemented by the operating system).
	`SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 40
	`SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');`
	Result: 28.5

TIME ZONE	The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC.
TIMEZONE_HOUR	The hour component of the time zone offset.
TIMEZONE_MINUTE	The minute component of the time zone offset.

WEEK	The number of the week of the calendar year that the day is in.
	`SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');`
	Result: 7
	`SELECT EXTRACT(WEEK FROM DATE '2001-02-16');`
	Result: 7

YEAR	The year field. Keep in mind there is no `0 AD`, so subtract `BC` years from `AD` years with care. `SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');` *Result:* 2001

Examples

```
=> SELECT EXTRACT (DAY FROM DATE '2008-12-25');
date_part
-----------
        25
(1 row)
=> SELECT EXTRACT (MONTH FROM DATE '2008-12-25');
date_part
-----------
        12
(1 row
SELECT EXTRACT(DOQ FROM CURRENT_DATE);
 date_part
-----------
        89
(1 row)
```

Remember that internally EXTRACT() uses the DATE_PART() function:

```
=> SELECT EXTRACT(EPOCH FROM AGE_IN_YEARS(TIMESTAMP '2009-02-24',
   TIMESTAMP '1972-03-02') :: INTERVAL year);

 date_part
-----------
 1136073600
(1 row)
```

In the above example, AGE_IN_YEARS is 36. The UNIX epoch uses 365.25 days per year:

```
=> SELECT 1136073600.0/36/(24*60*60);
 ?column?
----------
   365.25
(1 row)
```

You can extract the timezone hour from TIMETZ:

```
=> SELECT EXTRACT(timezone_hour FROM TIMETZ '10:30+13:30');
 date_part
-----------
        13
(1 row)
```

See Also

DATE_PART (page 202)

GETDATE

Returns the current system date and time as a TIMESTAMP value.

Behavior Type

Stable

Syntax

```
GETDATE();
```

Notes

- GETDATE is a stable function that requires parentheses but accepts no arguments.
- This function uses the date and time supplied by the operating system on the server to which you are connected, which is the same across all servers.
- GETDATE internally converts ***STATEMENT_TIMESTAMP()*** (page 239) from TIMESTAMPTZ to TIMESTAMP.
- This function is identical to ***SYSDATE()*** (page 239).

Example

```
=> SELECT GETDATE();
          GETDATE
```

```
---------------------------
 2011-03-07 13:21:29.497742
(1 row)
```

See Also

Date/Time Expressions (page 55)

GETUTCDATE

Returns the current system date and time as a TIMESTAMP value relative to UTC.

Behavior Type

Stable

Syntax

```
GETUTCDATE();
```

Notes

- GETUTCDATE is a stable function that requires parentheses but accepts no arguments.
- This function uses the date and time supplied by the operating system on the server to which you are connected, which is the same across all servers.
- GETUTCDATE is internally converted to STATEMENT_TIMESTAMP() at TIME ZONE 'UTC'.

Example

```
=> SELECT GETUTCDATE();
         GETUTCDATE
---------------------------
 2011-03-07 20:20:26.193052
(1 row)
```

See Also

Date/Time Expressions (page 55)

HOUR

Extracts the hour from a DATE, TIMESTAMP, TIMESTAMPTZ, VARCHAR, or INTERVAL value. The return value is of type INTEGER. (Hour 0 is midnight to 1 a.m.)

Syntax

```
HOUR ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the incoming DATE, TIMESTAMP, TIMESTAMPTZ, VARCHAR, or INTERVAL value.

Example

```
=> SELECT HOUR (TIMESTAMP 'sep 22, 2011 12:34');
 HOUR
------
   12
(1 row)

=> SELECT HOUR (INTERVAL '35 12:34');
 HOUR
------
   12
(1 row)

=> SELECT HOUR ('12:34');
 HOUR
------
   12
(1 row)
```

ISFINITE

Tests for the special TIMESTAMP constant INFINITY and returns a value of type BOOLEAN.

Behavior Type

Immutable

Syntax

```
ISFINITE ( timestamp )
```

Parameters

timestamp	Is an expression of type TIMESTAMP

Examples

```
SELECT ISFINITE(TIMESTAMP '2009-02-16 21:28:30');
 isfinite
----------
 t
(1 row)
SELECT ISFINITE(TIMESTAMP 'INFINITY');
 isfinite
----------
 f
(1 row)
```

JULIAN_DAY

Returns an INTEGER representing the Julian day based on an input TIMESTAMP, TIMESTAMPTZ, VARCHAR, or DATE value.

Syntax

```
JULIAN_DAY ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the TIMESTAMP, TIMESTAMPTZ, VARCHAR, or DATE input value.

Example

```
=> SELECT JULIAN_DAY(TIMESTAMP 'sep 22, 2011 12:34');
 JULIAN_DAY
------------
    2455827
(1 row)
```

LAST_DAY

Returns the last day of the month based on a TIMESTAMP. The TIMESTAMP can be supplied as a DATE or a TIMESTAMPTZ data type.

Behavior Type

Immutable, unless called with TIMESTAMPTZ, in which case it is Stable.

Syntax

```
LAST_DAY ( date );
```

Examples

The following example returns the last day of the month, February, as 29 because 2008 was a leap year:

```
SELECT LAST_DAY('2008-02-28 23:30 PST') "Last";
    Last
------------
 2008-02-29
(1 row)
```

The following example returns the last day of the month in March, after converting the string value to the specified DATE type:

```
SELECT LAST_DAY('2003/03/15') "Last";
    Last
------------
 2003-03-31
(1 row)
```

The following example returns the last day of February in the specified year (not a leap year):

```
SELECT LAST_DAY('2003/02/03') "Last";
    Last
------------
 2003-02-28
(1 row)
```

LOCALTIME

Returns a value of type TIME representing the time of day.

Behavior Type

Stable

Syntax

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

Parameters

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 timestamp.

Examples

```
SELECT LOCALTIME;
      time
-----------------
 16:16:06.790771
(1 row)
```

LOCALTIMESTAMP

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

Behavior Type

Stable

Syntax

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

Parameters

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 timestamp.

Examples

```
SELECT LOCALTIMESTAMP;
        timestamp
--------------------------
 2009-02-24 14:47:48.5951
(1 row)
```

MICROSECOND

Returns an INTEGER representing the microsecond portion of an input DATE, VARCHAR, TIMESTAMP, TIMESTAMPTZ, or INTERVAL value.

Syntax

```
MICROSECOND ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the DATE, VARCHAR, TIMESTAMP, TIMESTAMPTZ, or INTERVAL input value.

Example

```
=> SELECT MICROSECOND (TIMESTAMP 'Sep 22, 2011 12:34:01.123456');
 MICROSECOND
-------------
      123456
(1 row)
```

MIDNIGHT_SECONDS

Returns an INTEGER representing the number of seconds between midnight and the input value. The input value can be of type VARCHAR, TIME, TIMESTAMP, or TIMESTAMPTZ.

Syntax

```
MIDNIGHT_SECONDS ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, TIME, TIMESTAMP, or TIMESTAMPTZ input value.

Example

```
=> SELECT MIDNIGHT_SECONDS('12:34:00.987654');
 MIDNIGHT_SECONDS
------------------
            45240
(1 row)


=> SELECT MIDNIGHT_SECONDS(TIME '12:34:00.987654');
 MIDNIGHT_SECONDS
------------------
            45240
(1 row)

=> SELECT MIDNIGHT_SECONDS (TIMESTAMP 'sep 22, 2011 12:34');
 MIDNIGHT_SECONDS
------------------
            45240
(1 row)
```

MINUTE

Returns an INTEGER representing the minute value of the input value. The input value can be of type VARCHAR, DATE, TIMESTAMP, TIMESTAMPTZ, or INTERVAL.

Syntax

```
MINUTE ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, DATE, TIMESTAMP, TIMESTAMPTZ, or INTERVAL input value.

Example

```
=> SELECT MINUTE('12:34:03.456789');
 MINUTE
--------
     34
(1 row)

=>SELECT MINUTE (TIMESTAMP 'sep 22, 2011 12:34');
 MINUTE
--------
     34
(1 row)

=> SELECT MINUTE(INTERVAL '35 12:34:03.456789');
 MINUTE
--------
     34
(1 row)
```

MONTH

Returns an INTEGER representing the month portion of the input value. The input value can be of type VARCHAR, DATE, TIMESTAMP, TIMESTAMPTZ, or INTERVAL.

Syntax

```
MONTH( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the incoming VARCHAR, DATE, TIMESTAMP, TIMESTAMPTZ, or INTERVAL value.

Examples

```
=> SELECT MONTH('6-9');
 MONTH
-------
     9
```

```
(1 row)


=> SELECT MONTH (TIMESTAMP 'sep 22, 2011 12:34');
 MONTH
-------
     9
(1 row)

=> SELECT MONTH(INTERVAL '2-35' year to month);
 MONTH
-------
    11
(1 row)
```

MONTHS_BETWEEN

Returns the number of months between *date1* and *date2* as a FLOAT8. where the input arguments can be of TIMESTAMP, DATE, or TIMESTAMPTZ type.

Behavior Type

Immutable for TIMESTAMP and Date, Stable for TIMESTAMPTZ

Syntax

```
MONTHS_BETWEEN ( date1 , date2 );
```

Parameters

date1, date2	If *date1* is later than *date2,* then the result is positive. If *date1* is earlier than *date2,* then the result is negative. If *date1* and *date2* are either the same days of the month or both are the last days of their respective month, then the result is always an integer. Otherwise MONTHS_BETWEEN returns a FLOAT8 result based on a 31-day month, which considers the difference between *date1* and *date2.*

Examples

Note the following result is an integral number of days because the dates are on the same day of the month:

```
SELECT MONTHS_BETWEEN('2009-03-07 16:00'::TIMESTAMP, '2009-04-07
15:00'::TIMESTAMP);
 months_between
----------------
             -1
(1 row)
```

The result from the following example returns an integral number of days because the days fall on the last day of their respective months:

```
SELECT MONTHS_BETWEEN('29Feb2000', '30Sep2000') "Months";
     Months
----------------
             -7
(1 row)
```

In this example, and in the example that immediately follows it, `MONTHS_BETWEEN()` returns the number of months between *date1* and *date2* as a fraction because the days do not fall on the same day or on the last day of their respective months:

```
SELECT MONTHS_BETWEEN(TO_DATE('02-02-1995','MM-DD-YYYY'),
                      TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months";
      Months
------------------
 1.03225806451613
(1 row)
SELECT MONTHS_BETWEEN(TO_DATE ('2003/01/01', 'yyyy/mm/dd'),
                      TO_DATE ('2003/03/14', 'yyyy/mm/dd') ) "Months";
      Months
-------------------
 -2.41935483870968
(1 row)
```

The following two examples use the same *date1* and *date2* strings, but they are cast to a different data types (TIMESTAMP and TIMESTAMPTZ). The result set is the same for both statements:

```
SELECT MONTHS_BETWEEN('2008-04-01'::timestamp, '2008-02-29'::timestamp);
  months_between
------------------
 1.09677419354839
(1 row)
SELECT MONTHS_BETWEEN('2008-04-01'::timestamptz, '2008-02-29'::timestamptz);
  months_between
------------------
 1.09677419354839
(1 row)
```

The following two examples show alternate inputs:

```
SELECT MONTHS_BETWEEN('2008-04-01'::date, '2008-02-29'::timestamp);
  months_between
------------------
 1.09677419354839
(1 row)
SELECT MONTHS_BETWEEN('2008-02-29'::timestamptz, '2008-04-01'::date);
  months_between
-------------------
 -1.09677419354839
(1 row)
```

NEW_TIME

Converts a TIMESTAMP value between time zones. Intervals are not permitted.

Behavior Type

Immutable

Syntax

```
NEW_TIME( 'timestamp' , 'timezone1' , 'timezone2')
```

Returns

TIMESTAMP

Parameters

timestamp	The TIMESTAMP (or a TIMESTAMPTZ, DATE, or character string which can be converted to a TIMESTAMP) representing a TIMESTAMP in *timezone1* that returns the equivalent timestamp in *timezone2*.
timezone1	VARCHAR string of the form required by the ***TIMESTAMP AT TIMEZONE*** (page 102) '*zone*' clause. *timezone1* indicates the time zone from which you want to convert *timestamp*. It must be a valid timezone, as listed in the field for *timezone2* below.
timezone2	VARCHAR string of the form required by the ***TIMESTAMP AT TIMEZONE*** (page 102) '*zone*' clause. *timezone2* indicates the time zone into which you want to convert *timestamp*.

Notes

The timezone arguments are character strings of the form required by the ***TIMESTAMP AT TIMEZONE*** (page 102) '*zone*' clause; for example:

AST, ADT	Atlantic Standard Time or Daylight Time
BST, BDT	Bering Standard Time or Daylight Time
CST, CDT	Central Standard Time or Daylight Time
EST, EDT	Eastern Standard Time or Daylight Time
GMT	Greenwich Mean Time
HST	Alaska-Hawaii Standard Time
MST, MDT	Mountain Standard Time or Daylight Time
NST	Newfoundland Standard Time
PST, PDT	Pacific Standard Time or Daylight Time

Examples

The following command converts the specified time from Eastern Standard Time to Pacific Standard Time:

```
=> SELECT NEW_TIME('05-24-12 13:48:00', 'EST', 'PST');
      NEW_TIME
---------------------
 2012-05-24 10:48:00
(1 row)
```

This command converts the time on January 1 from Eastern Standard Time to Pacific Standard Time. Notice how the time rolls back to the previous year:

```
=> SELECT NEW_TIME('01-01-12 01:00:00', 'EST', 'PST');
      NEW_TIME
---------------------
 2011-12-31 22:00:00
(1 row)
```

Query the current system time:

```
=> SELECT NOW();
              now
-------------------------------
 2012-05-24 08:28:10.155887-04
(1 row)

=> SELECT NEW_TIME('NOW', 'EDT', 'CDT');
          NEW_TIME
----------------------------
 2012-05-24 07:28:10.155887
(1 row)
```

The following example returns the year 45 before the Common Era in Greenwich Mean Time and converts it to Newfoundland Standard Time:

```
=>  SELECT NEW_TIME('April 1, 45 BC', 'GMT', 'NST');
        NEW_TIME
------------------------
 0045-03-31 20:30:00 BC
(1 row)


=> SELECT NEW_TIME('April 1 2011', 'EDT', 'PDT');
      NEW_TIME
---------------------
 2011-03-31 21:00:00
(1 row)


=> SELECT NEW_TIME('May 24, 2012 10:00', 'Pacific/Kiritamati', 'EDT');
      NEW_TIME
---------------------
 2011-05-23 16:00:00
```

```
(1 row)
```

NEXT_DAY

Returns the date of the first instance of a particular day of the week that follows the specified date.

Behavior Type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Syntax

```
NEXT_DAY( 'date', 'DOW')
```

Parameters

date	Can be VARCHAR, TIMESTAMP, TIMESTAMPTZ, or DATE. Only standard English day-names and day-name abbreviations are accepted.
DOW	Day of week can be a CHAR/VARCHAR string or a character constant. DOW is not case sensitive and trailing spaces are ignored.

Examples

The following example returns the date of the next Friday following the specified date. All are variations on the same query, and all return the same result:

```
=> SELECT NEXT_DAY('28-MAR-2011','FRIDAY') "NEXT DAY" FROM DUAL;
  NEXT DAY
------------
 2011-04-01
(1 row)

=> SELECT NEXT_DAY('March 28 2011','FRI') "NEXT DAY" FROM DUAL;
  NEXT DAY
------------
 2011-04-01
(1 row)

=> SELECT NEXT_DAY('3-29-11','FRI') "NEXT DAY" FROM DUAL;
  NEXT DAY
------------
 2011-04-01
(1 row)
```

NOW [Date/Time]

Returns a value of type TIMESTAMP WITH TIME ZONE representing the start of the current transaction. NOW is equivalent to ***CURRENT_TIMESTAMP*** (page 201) except that it does not accept a precision parameter.

Behavior Type

Stable

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 timestamp.

Examples

```
SELECT NOW();
               NOW
-------------------------------
 2010-04-01 15:31:12.144584-04
(1 row)
```

See Also

CURRENT_TIMESTAMP (page 201)

OVERLAPS

Returns true when two time periods overlap, false when they do not overlap.

Behavior Type

Stable when TIMESTAMP and TIMESTAMPTZ are both used, or when TIMESTAMPTZ is used with INTERVAL, Immutable otherwise.

Syntax

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

Parameters

start	Is a DATE, TIME, or TIME STAMP value that specifies the beginning of a time period.
end	Is a DATE, TIME, or TIME STAMP value that specifies the end of a time period.

interval	Is a value that specifies the length of the time period.

Examples

The first command returns true for an overlap in date range of 2007-02-16 – 2007-12-21 with 2007-10-30 – 2008-10-30.

```
SELECT (DATE '2007-02-16', DATE '2007-12-21')
  OVERLAPS (DATE '2007-10-30', DATE '2008-10-30');
 overlaps
----------
 t
(1 row)
```

The next command returns false for an overlap in date range of 2007-02-16 – 2007-12-21 with 2008-10-30 – 2008-10-30.

```
SELECT (DATE '2007-02-16', DATE '2007-12-21')
  OVERLAPS (DATE '2008-10-30', DATE '2008-10-30');
 overlaps
----------
 f
(1 row)
```

The next command returns false for an overlap in date range of 2007-02-16, 22 hours ago with 2007-10-30, 22 hours ago.

```
SELECT (DATE '2007-02-16', INTERVAL '1 12:59:10')
  OVERLAPS (DATE '2007-10-30', INTERVAL '1 12:59:10');
 overlaps
----------
 f
(1 row)
```

QUARTER

Returns an INTEGER representing calendar quarter into which the input value falls. The input value can be of type VARCHAR, DATE, TIMESTAMP or TIMESTAMPTZ.

Syntax

```
QUARTER( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the DATE, VARCHAR, TIMESTAMP, or TIMESTAMPTZ input value.

Example

```
=> SELECT QUARTER (TIMESTAMP 'sep 22, 2011 12:34');
```

```
 QUARTER
---------
       3
(1 row)
```

ROUND [Date/Time]

Rounds a TIMESTAMP, TIMESTAMPTZ, or DATE. The return value is of type TIMESTAMP.

Behavior Type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Syntax

```
ROUND([TIMESTAMP | DATE] , format )
```

Parameters

TIMESTAMP | DATE	Is the TIMESTAMP or DATE input value.
format	Is a string constant that selects the precision to which truncate the input value. Valid values for format are:

Precision	Valid values
Century	CC, SCC
Year	SYYY, YYYY, YEAR, YYY, YY,Y
ISO Year	IYYY, IYY, IY, I
Quarter	Q
Month	MONTH, MON, MM, RM
Same day of the week as the first day of the year	WW
Same day of the week as the first day of the ISO year	IW
Same day of the week as the first day of the month	W
Day	DDD, DD, J
Starting day of the week	DAY, DY, D
Hour	HH, HH12, HH24
Minute	MI
Second	SS

Examples

```
=> SELECT ROUND(TIMESTAMP 'sep 22, 2011 12:34:00', 'dy');
        ROUND
---------------------
 2011-09-18 00:00:00
(1 row)
```

SECOND

Returns an INTEGER representing the second portion of the input value. The input value can be of type VARCHAR, TIMESTAMP, TIMESTAMPTZ, or INTERVAL.

Syntax

```
SECOND( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, TIMESTAMP, TIMESTAMPTZ, or INTERVAL input value.

Examples

```
=> SELECT SECOND ('23:34:03.456789');
 SECOND
--------
      3
(1 row)

=> SELECT SECOND (TIMESTAMP 'sep 22, 2011 12:34');
 SECOND
--------
      0
(1 row)

=> SELECT SECOND (INTERVAL '35 12:34:03.456789');
 SECOND
--------
      3
(1 row)
```

STATEMENT_TIMESTAMP

Is similar to ***TRANSACTION_TIMESTAMP*** (page 251). It returns a value of type TIMESTAMP WITH TIME ZONE representing the start of the current statement.

Behavior Type

Stable

Syntax

```
STATEMENT_TIMESTAMP()
```

Notes

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

Examples

```
SELECT STATEMENT_TIMESTAMP();
      STATEMENT_TIMESTAMP
-------------------------------
 2010-04-01 15:40:42.223736-04
(1 row)
```

See Also

CLOCK_TIMESTAMP (page 199)

TRANSACTION_TIMESTAMP (page 251)

SYSDATE

Returns the current system date and time as a TIMESTAMP value.

Behavior Type

Stable

Syntax

```
SYSDATE();
```

Notes

- SYSDATE is a stable function (called once per statement) that requires no arguments. Parentheses are optional.
- This function uses the date and time supplied by the operating system on the server to which you are connected, which must be the same across all servers.
- In implementation, SYSDATE converts ***STATEMENT_TIMESTAMP*** (page 239) from TIMESTAMPTZ to TIMESTAMP.

- This function is identical to ***GETDATE()*** (page 222).

Examples

```
=> SELECT SYSDATE();
           sysdate
----------------------------
 2011-03-07 13:22:28.295802
(1 row)
```

See Also

Date/Time Expressions (page 55)

TIME_SLICE

Aggregates data by different fixed-time intervals and returns a rounded-up input TIMESTAMP value to a value that corresponds with the start or end of the time slice interval.

Given an input TIMESTAMP value, such as '2000-10-28 00:00:01', the start time of a 3-second time slice interval is '2000-10-28 00:00:00', and the end time of the same time slice is '2000-10-28 00:00:03'.

Behavior Type

Immutable

Syntax

```
TIME_SLICE(expression, slice_length,
  [ time_unit = 'SECOND' ],
  [ start_or_end = 'START' ] )
```

Parameters

expression	Is evaluated on each row. Can be either a column of type TIMESTAMP or a (string) constant that can be parsed into a TIMESTAMP value, such as '2004-10-19 10:23:54'.
slice_length	Is the length of the slice specified in integers. Input must be a positive integer.
time_unit	Is the time unit of the slice with a default of SECOND. Domain of possible values: { HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND }.
start_or_end	Indicates whether the returned value corresponds to the start or end time of the time slice interval. The default is START. Domain of possible values: { START, END }.

Notes

- The returned value's data type is TIMESTAMP.

- The corresponding SQL data type for TIMESTAMP is TIMESTAMP WITHOUT TIME ZONE. HP Vertica supports TIMESTAMP for TIME_SLICE instead of DATE and TIME data types.
- TIME_SLICE exhibits the following behavior around nulls:
  - The system returns an error when any one of *slice_length*, *time_unit*, or *start_or_end* parameters is null.
  - When *slice_length*, *time_unit*, and *start_or_end* contain legal values, and *expression* is null, the system returns a NULL value, instead of an error.

Usage

The following command returns the (default) start time of a 3-second time slice:

```
SELECT TIME_SLICE('2009-09-19 00:00:01', 3);
     time_slice
---------------------
 2009-09-19 00:00:00
(1 row)
```

The following command returns the end time of a 3-second time slice:

```
SELECT TIME_SLICE('2009-09-19 00:00:01', 3, 'SECOND', 'END');
     time_slice
---------------------
 2009-09-19 00:00:03
(1 row)
```

This command returns results in milliseconds, using a 3-second time slice:

```
   SELECT TIME_SLICE('2009-09-19 00:00:01', 3, 'ms');
          time_slice
   -------------------------
    2009-09-19 00:00:00.999
(1 row)
```

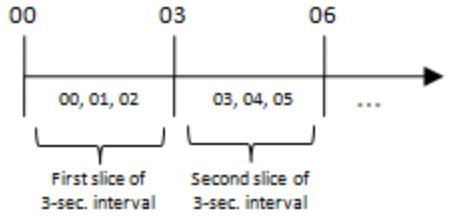
This command returns results in microseconds, using a 9-second time slice:

```
SELECT TIME_SLICE('2009-09-19 00:00:01', 3, 'us');
         time_slice
----------------------------
 2009-09-19 00:00:00.999999
(1 row)
```

The next example uses a 3-second interval with an input value of '00:00:01'. To focus specifically on seconds, the example omits date, though all values are implied as being part of the timestamp with a given input of `'00:00:01'`:

- '00:00:**00**' is the start of the 3-second time slice
- '00:00:**03**' is the end of the 3-second time slice.

- '00:00:03' is also the start of the *second* 3-second time slice. In time slice boundaries, the end value of a time slice does not belong to that time slice; it starts the next one.

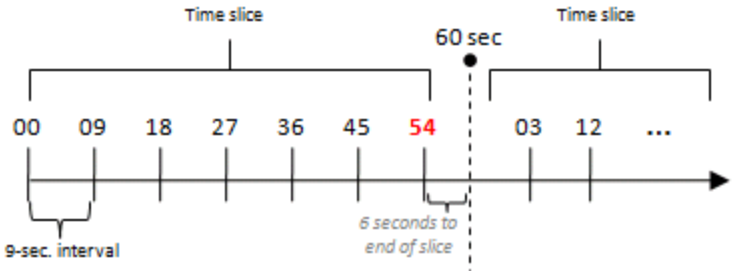


When the time slice interval is not a factor of 60 seconds, such as a given slice length of 9 in the following example, the slice does not always start or end on 00 seconds:

```
SELECT TIME_SLICE('2009-02-14 20:13:01', 9);
     time_slice
---------------------
 2009-02-14 20:12:54
(1 row)
```

This is expected behavior, as the following properties are true for all time slices:

- Equal in length
- Consecutive (no gaps between them)
- Non-overlapping



To force the above example ('2009-02-14 20:13:01') to start at '2009-02-14 20:13:00', adjust the output timestamp values so that the remainder of 54 counts up to 60:

```
SELECT TIME_SLICE('2009-02-14 20:13:01', 9 )+'6 seconds'::INTERVAL AS time;
        time
---------------------
 2009-02-14 20:13:00
(1 row)
```

Alternatively, you could use a different slice length, which is divisible by 60, such as 5:

```
SELECT TIME_SLICE('2009-02-14 20:13:01', 5);
     time_slice
---------------------
 2009-02-14 20:13:00
(1 row)
```

A TIMESTAMPTZ value is implicitly cast to TIMESTAMP. For example, the following two statements have the same effect.

```
SELECT TIME_SLICE('2009-09-23 11:12:01'::timestamptz, 3);
     TIME_SLICE
---------------------
 2009-09-23 11:12:00
(1 row)
SELECT TIME_SLICE('2009-09-23 11:12:01'::timestamptz::timestamp, 3);
     TIME_SLICE
---------------------
 2009-09-23 11:12:00
(1 row)
```

Examples

You can use the SQL analytic functions FIRST_VALUE and LAST_VALUE to find the first/last price within each time slice group (set of rows belonging to the same time slice). This structure could be useful if you want to sample input data by choosing one row from each time slice group.

```
SELECT date_key, transaction_time, sales_dollar_amount,
TIME_SLICE(DATE '2000-01-01' + date_key + transaction_time, 3),
FIRST_VALUE(sales_dollar_amount)
OVER (PARTITION BY TIME_SLICE(DATE '2000-01-01' + date_key + transaction_time, 3)
     ORDER BY DATE '2000-01-01' + date_key + transaction_time) AS first_value
FROM store.store_sales_fact
LIMIT 20;
 date_key | transaction_time | sales_dollar_amount |     time_slice      | first_value
----------+------------------+---------------------+---------------------+-------------
        1 | 00:41:16         |                 164 | 2000-01-02 00:41:15 |         164
        1 | 00:41:33         |                 310 | 2000-01-02 00:41:33 |         310
        1 | 15:32:51         |                 271 | 2000-01-02 15:32:51 |         271
        1 | 15:33:15         |                 419 | 2000-01-02 15:33:15 |         419
        1 | 15:33:44         |                 193 | 2000-01-02 15:33:42 |         193
        1 | 16:36:29         |                 466 | 2000-01-02 16:36:27 |         466
        1 | 16:36:44         |                 250 | 2000-01-02 16:36:42 |         250
        2 | 03:11:28         |                  39 | 2000-01-03 03:11:27 |          39
        3 | 03:55:15         |                 375 | 2000-01-04 03:55:15 |         375
        3 | 11:58:05         |                 369 | 2000-01-04 11:58:03 |         369
        3 | 11:58:24         |                 174 | 2000-01-04 11:58:24 |         174
        3 | 11:58:52         |                 449 | 2000-01-04 11:58:51 |         449
        3 | 19:01:21         |                 201 | 2000-01-04 19:01:21 |         201
        3 | 22:15:05         |                 156 | 2000-01-04 22:15:03 |         156
        4 | 13:36:57         |                -125 | 2000-01-05 13:36:57 |        -125
        4 | 13:37:24         |                -251 | 2000-01-05 13:37:24 |        -251
        4 | 13:37:54         |                 353 | 2000-01-05 13:37:54 |         353
        4 | 13:38:04         |                 426 | 2000-01-05 13:38:03 |         426
        4 | 13:38:31         |                 209 | 2000-01-05 13:38:30 |         209
        5 | 10:21:24         |                 488 | 2000-01-06 10:21:24 |         488
(20 rows)
```

Notice how TIME_SLICE rounds the transaction time to the 3-second slice length.

The following example uses the analytic (window) OVER() clause to return the last trading price (the last row ordered by TickTime) in each 3-second time slice partition:

```
SELECT DISTINCT TIME_SLICE(TickTime, 3), LAST_VALUE(price)
OVER (PARTITION BY TIME_SLICE(TickTime, 3)
ORDER BY TickTime ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING);
```

Note: If you omit the windowing clause from an analytic clause, LAST_VALUE defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Results can seem non-intuitive, because instead of returning the value from the bottom of the current partition, the function returns the bottom of the *window*, which continues to change along with the current input row that is being processed. For more information, see Using Time Series Analytics and Using SQL Analytics in the Programmer's Guide.

In the example below, FIRST_VALUE is evaluated once for each input record and the data is sorted by ascending values. Use SELECT DISTINCT to remove the duplicates and return only one output record per TIME_SLICE:

```
SELECT DISTINCT TIME_SLICE(TickTime, 3), FIRST_VALUE(price)
OVER (PARTITION BY TIME_SLICE(TickTime, 3)
ORDER BY TickTime ASC)
FROM tick_store;
     TIME_SLICE      | ?column?
---------------------+----------
 2009-09-21 00:00:06 |    20.00
 2009-09-21 00:00:09 |    30.00
 2009-09-21 00:00:00 |    10.00
(3 rows)
```

The information output by the above query can also return MIN, MAX, and AVG of the trading prices within each time slice.

```
SELECT DISTINCT TIME_SLICE(TickTime, 3),
FIRST_VALUE(Price) OVER (PARTITION BY TIME_SLICE(TickTime, 3)
ORDER BY TickTime ASC),
  MIN(price) OVER (PARTITION BY TIME_SLICE(TickTime, 3)),
  MAX(price) OVER (PARTITION BY TIME_SLICE(TickTime, 3)),
  AVG(price) OVER (PARTITION BY TIME_SLICE(TickTime, 3))
FROM tick_store;
```

See Also

Aggregate Functions (page 118)

FIRST_VALUE (page 160), ***LAST_VALUE*** (page 166), ***TIMESERIES Clause*** (page 894), ***TS_FIRST_VALUE*** (page 421), and ***TS_LAST_VALUE*** (page 422)

Using Time Zones with HP Vertica in the Administrator's Guide

TIMEOFDAY

Returns a text string representing the time of day.

Behavior Type

Volatile

Syntax

```
TIMEOFDAY()
```

Notes

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

Examples

```
SELECT TIMEOFDAY();
              TIMEOFDAY
-------------------------------------
 Thu Apr 01 15:42:04.483766 2010 EDT
(1 row)
```

TIMESTAMPADD

Adds a specified number of intervals to a TIMESTAMP or TIMESTAMPTZ. The return value depends on the input, as follows:

- If *starttimestamp* is of TIMESTAMP, the return value is of type TIMESTAMP.
- If *starttimestamp* is of TIMESTAMPTZ, the return value is of type TIMESTAMPTZ.

Behavior Type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Syntax 1

```
TIMESTAMPADD ( datepart ,interval, starttimestamp );
```

Parameters

datepart	(VARCHAR) Returns the number of specified datepart boundaries between the specified startdate and enddate. Can be an unquoted identifier, a quoted string, or an expression in parentheses, which evaluates to the datepart as a character string. The following table lists the valid *datepart* arguments. datepart* — abbreviation --- — --- YEAR — yy, yyyy QUARTER — qq, q MONTH — mm, m DAY — dd, d, dy, dayofyear, y WEEK — wk, ww HOUR — hh MINUTE — mi, n SECOND — ss, s MILLISECOND — ms MICROSECOND — mcs, us * Each of these dateparts can be prefixed with SQL_TSI_ (i.e. SQL_TSI_YEAR, SQL_TSI_DAY, and so forth.)
starttimestamp	Is the start TIMESTAMP or TIMESTAMPTZ for the calculation.
endtimestamp	Is the end TIMESTAMP for the calculation.

Notes

- TIMESTAMPDIFF() is an immutable function with a default type of TIMESTAMP. If TIMESTAMPTZ is specified, the function is stable.
- HP Vertica accepts statements written in any of the following forms:

  ```
  TIMESTAMPDIFF(year, s, e);
  TIMESTAMPDIFF('year', s, e);
  ```

  If you use an expression, the expression must be enclosed in parentheses:

  ```
  DATEDIFF((expression), s, e);
  ```

- Starting arguments are not included in the count, but end arguments are included.

Example

```
=> SELECT TIMESTAMPADD (SQL_TSI_MONTH, 2,('jan 1, 2006'));
      timestampadd
------------------------
 2006-03-01 00:00:00-05
(1 row)
```

See Also

Date/Time Expressions (page 55)

TIMESTAMPDIFF

Returns the difference between two TIMESTAMP or TIMESTAMPTZ values, based on the specified start and end arguments.

Behavior Type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Syntax 1

```
TIMESTAMPDIFF ( datepart , starttimestamp , endtimestamp );
```

Parameters

datepart	(VARCHAR) Returns the number of specified datepart boundaries between the specified startdate and enddate. Can be an unquoted identifier, a quoted string, or an expression in parentheses, which evaluates to the datepart as a character string. The following table lists the valid *datepart* arguments. datepart — abbreviation year — yy, yyyy quarter — qq, q month — mm, m day — dd, d, dy, dayofyear, y week — wk, ww hour — hh minute — mi, n second — ss, s millisecond — ms microsecond — mcs, us
starttimestamp	Is the start TIMESTAMP for the calculation.
endtimestamp	Is the end TIMESTAMP for the calculation.

Notes

- TIMESTAMPDIFF() is an immutable function with a default type of TIMESTAMP. If TIMESTAMPTZ is specified, the function is stable.
- HP Vertica accepts statements written in any of the following forms:

  ```
  TIMESTAMPDIFF(year, s, e);
  TIMESTAMPDIFF('year', s, e);
  ```

  If you use an expression, the expression must be enclosed in parentheses:

  ```
  TIMESTAMPDIFF((expression), s, e);
  ```

- Starting arguments are not included in the count, but end arguments are included.

Example

```
=> SELECT TIMESTAMPDIFF ('YEAR',('jan 1, 2006 12:34:00'), ('jan 1, 2008
12:34:00'));
 timestampdiff
---------------
             2
(1 row)
```

See Also

Date/Time Expressions (page 55)

TIMESTAMP_ROUND

Rounds a TIMESTAMP to a specified *format*. The return value is of type TIMESTAMP.

Behavior Type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Syntax

```
TIMESTAMP_ROUND ( timestamp, format )
```

Parameters

timestamp	Is the TIMESTAMP or TIMESTAMPTZ input value.

format	Is a string constant that selects the precision to which truncate the input value. Valid values for format are:

Precision	Valid values
Century	CC, SCC
Year	SYYY, YYYY, YEAR, YYY, YY,Y
ISO Year	IYYY, IYY, IY, I
Quarter	Q
Month	MONTH, MON, MM, RM
Same day of the week as the first day of the year	WW
Same day of the week as the first day of the ISO year	IW
Same day of the week as the first day of the month	W
Day	DDD, DD, J
Starting day of the week	DAY, DY, D
Hour	HH, HH12, HH24
Minute	MI
Second	SS

Examples

```
b=> SELECT TIMESTAMP_ROUND('sep 22, 2011 12:34:00', 'dy');
   TIMESTAMP_ROUND
---------------------
 2011-09-18 00:00:00
(1 row)
```

TIMESTAMP_TRUNC

Truncates a TIMESTAMP. The return value is of type TIMESTAMP.

Behavior Type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Syntax

```
TIMESTAMP_TRUNC ( timestamp, format )
```

Parameters

timestamp	Is the TIMESTAMP or TIMESTAMPTZ input value.
format	Is a string constant that selects the precision to which truncate the input value. Valid values for format are:

Precision	Valid values
Century	CC, SCC
Year	SYYY, YYYY, YEAR, YYY, YY,Y
ISO Year	IYYY, IYY, IY, I
Quarter	Q
Month	MONTH, MON, MM, RM
Same day of the week as the first day of the year	WW
Same day of the week as the first day of the ISO year	IW
Same day of the week as the first day of the month	W
Day	DDD, DD, J
Starting day of the week	DAY, DY, D
Hour	HH, HH12, HH24
Minute	MI
Second	SS

Examples

```
=> SELECT TIMESTAMP_TRUNC('sep 22, 2011 12:34:00');
   TIMESTAMP_TRUNC
---------------------
 2011-09-22 00:00:00
(1 row)

=> SELECT TIMESTAMP_TRUNC('sep 22, 2011 12:34:00', 'dy');
   TIMESTAMP_TRUNC
---------------------
 2011-09-18 00:00:00
(1 row)
```

TRANSACTION_TIMESTAMP

Returns a value of type TIMESTAMP WITH TIME ZONE representing the start of the current transaction. TRANSACTION_TIMESTAMP is equivalent to ***CURRENT_TIMESTAMP*** (page 201) except that it does not accept a precision parameter.

Behavior Type

Stable

Syntax

```
TRANSACTION_TIMESTAMP()
```

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 timestamp.

Examples

```
SELECT TRANSACTION_TIMESTAMP();
     TRANSACTION_TIMESTAMP
-------------------------------
 2010-04-01 15:31:12.144584-04
(1 row)
```

See Also

CLOCK_TIMESTAMP (page 199) and ***STATEMENT_TIMESTAMP*** (page 239)

TRUNC [Date/Time]

Truncates a TIMESTAMP, TIMESTAMPTZ, or DATE. The return value is of type TIMESTAMP.

Behavior Type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Syntax

```
TRUNC ([TIMESTAMP | DATE] , format )
```

Parameters

TIMESTAMP | DATE	Is the TIMESTAMP or DATE input value.

format	Is a string constant that selects the precision to which truncate the input value. Valid values for format are:

Precision	Valid values
Century	CC, SCC
Year	SYYY, YYYY, YEAR, YYY, YY,Y
ISO Year	IYYY, IYY, IY, I
Quarter	Q
Month	MONTH, MON, MM, RM
Same day of the week as the first day of the year	WW
Same day of the week as the first day of the ISO year	IW
Same day of the week as the first day of the month	W
Day	DDD, DD, J
Starting day of the week	DAY, DY, D
Hour	HH, HH12, HH24
Minute	MI
Second	SS

Examples

```
=> SELECT TRUNC(TIMESTAMP 'sep 22, 2011 12:34:00', 'dy');
        TRUNC
---------------------
 2011-09-18 00:00:00
(1 row)
```

WEEK

Returns an INTEGER representing the week of the year into which the input value falls. A week starts on Sunday. January 1 is always in the first week of the year.

The input is of type VARCHAR, DATE, TIMESTAMP, or TIMESTAMPTZ.

Syntax

```
WEEK ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, DATE, TIMESTAMP, TIMESTAMPTZ input value.

Example

```
=> SELECT WEEK (TIMESTAMP 'sep 22, 2011 12:34');
 WEEK
------
   39
(1 row)
```

WEEK_ISO

Returns an INTEGER from 1 - 53 representing the week of the year into which the input value falls. The return value is based on the ISO 8061 standard. The input is of VARCHAR, DATE, TIMESTAMP, or TIMESTAMPTZ.

The ISO week consists of 7 days starting on Monday and ending on Sunday. The first week of the year is the week that contains January 4.

Syntax

```
WEEK_ISO ( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, DATE, TIMESTAMP, or TIMESTAMPTZ input value.

Example

The following examples illustrate the different results returned by WEEK_ISO. The first shows that December 28, 2011 falls within week 52 of the ISO calendar:

```
=> SELECT WEEK_ISO (TIMESTAMP 'Dec 28, 2011 10:00:00');
 WEEK_ISO
----------
       52
(1 row)
```

The second example shows WEEK_ISO results for January 1, 2012. Note that, since this date falls on a Sunday, it falls within week 52 of the ISO year:

```
=> SELECT WEEK_ISO (TIMESTAMP 'Jan 1, 2012 10:00:00');
 WEEK_ISO
```

```
----------
        52
(1 row)
```

The third example shows WEEK_ISO results for January 2, 2012, which occurs on a Monday. This is the first week of the year that contains a Thursday and contains January 4. The function returns week 1.

```
=> SELECT WEEK_ISO (TIMESTAMP 'Jan 2, 2012 10:00:00');
 WEEK_ISO
----------
         1
```

The last example shows how to combine the DAYOFWEEK_ISO, WEEK_ISO, and YEAR_ISO functions to find the ISO day of the week, week, and year:

```
=> SELECT DAYOFWEEK_ISO('Jan 1, 2000'), WEEK_ISO('Jan 1,
2000'),YEAR_ISO('Jan1,2000');
 DAYOFWEEK_ISO | WEEK_ISO | YEAR_ISO
---------------+----------+----------
             6 |       52 |     1999
(1 row)
```

See Also

YEAR_ISO (page 255)

DAYOFWEEK_ISO (page 216)

http://en.wikipedia.org/wiki/ISO_8601 (***http://en.wikipedia.org/wiki/ISO_8601***)

YEAR

Returns an INTEGER representing the year portion of the input value. The input value can be of type VARCHAR, TIMESTAMP, TIMESTAMPTZ, or INTERVAL.

Syntax

```
YEAR( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, TIMESTAMP, TIMESTAMPTZ, or INTERVAL input value.

Example

```
=> SELECT YEAR ('6-9');
 YEAR
------
```

```
    6
(1 row)


=> SELECT YEAR (TIMESTAMP 'sep 22, 2011 12:34');
 YEAR
------
 2011
(1 row)

=> SELECT YEAR (INTERVAL '2-35' year to month);
 YEAR
------
    4
(1 row)
```

YEAR_ISO

Returns an INTEGER representing the year portion of the input value. The return value is based on the ISO 8061 standard. The input value can be of type VARCHAR, DATE, TIMESTAMP, or TIMESTAMPTZ.

The first week of the ISO year is the week that contains January 4.

Syntax

```
YEAR_ISO( d )
```

Behavior type

Immutable, except for TIMESTAMPTZ arguments where it is Stable.

Parameters

d	Is the VARCHAR, DATE, TIMESTAMP, or TIMESTAMPTZ input value.

Example

```
=> SELECT YEAR_ISO (TIMESTAMP 'sep 22, 2011 12:34');
 YEAR_ISO
----------
     2011
(1 row)
```

The following example shows how to combine the DAYOFWEEK_ISO, WEEK_ISO, and YEAR_ISO functions to find the ISO day of the week, week, and year:

```
=> SELECT DAYOFWEEK_ISO('Jan 1, 2000'), WEEK_ISO('Jan 1,
2000'),YEAR_ISO('Jan1,2000');
```

```
 DAYOFWEEK_ISO | WEEK_ISO | YEAR_ISO
---------------+----------+----------
             6 |       52 |     1999
(1 row)
```

See also

WEEK_ISO (page 253)

DAYOFWEEK_ISO (page 216)

http://en.wikipedia.org/wiki/ISO_8601 (***http://en.wikipedia.org/wiki/ISO_8601***)

Formatting Functions

Formatting functions provide a powerful tool set 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.
- 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_BITSTRING

Returns a VARCHAR that represents the given VARBINARY value in bitstring format

Behavior Type

Immutable

Syntax

```
TO_BITSTRING ( expression )
```

Parameters

expression	(VARCHAR) is the string to return.

Notes

VARCHAR TO_BITSTRING(VARBINARY) converts data from binary type to character type (where the character representation is the bitstring format). This function is the inverse of BITSTRING_TO_BINARY:

```
TO_BITSTRING(BITSTRING_TO_BINARY(x)) = x)
BITSTRING_TO_BINARY(TO_BITSTRING(x)) = x)
```

Examples

```
SELECT TO_BITSTRING('ab'::BINARY(2));
   to_bitstring
------------------
 0110000101100010
(1 row)
SELECT TO_BITSTRING(HEX_TO_BINARY('0x10'));
to_bitstring
--------------
00010000
(1 row)
SELECT TO_BITSTRING(HEX_TO_BINARY('0xF0'));
to_bitstring
--------------
11110000
(1 row)
```

See Also

BITCOUNT (page 359) and ***BITSTRING_TO_BINARY*** (page 360)

TO_CHAR

Converts various date/time and numeric values into text strings.

Behavior Type

Stable

Syntax

```
TO_CHAR ( expression [, pattern ] )
```

Parameters

expression	(TIMESTAMP, TIMESTAMPTZ, TIME, TIMETZ, INTERVAL, INTEGER, DOUBLE PRECISION) specifies the value to convert.
pattern	[Optional] (CHAR or VARCHAR) specifies an output pattern string using the ***Template Patterns for Date/Time Formatting*** (page 265) and and/or ***Template Patterns for Numeric Formatting*** (page 268).

Notes

- TO_CHAR(any) casts any type, except BINARY/VARBINARY, to VARCHAR.
  The following example returns an error if you attempt to cast `TO_CHAR` to a binary data type:
  ```
  => SELECT TO_CHAR('abc'::VARBINARY);
  ERROR:  cannot cast type varbinary to varchar
  ```
- TO_CHAR accepts TIME and TIMETZ data types as inputs if you explicitly cast TIME to TIMESTAMP and TIMETZ to TIMESTAMPTZ.

```
=> SELECT TO_CHAR(TIME '14:34:06.4','HH12:MI am');
=> SELECT TO_CHAR(TIMETZ '14:34:06.4+6','HH12:MI am');
```

You can extract the timezone hour from TIMETZ:

```
SELECT EXTRACT(timezone_hour FROM TIMETZ '10:30+13:30');
 date_part
-----------
        13
(1 row)
```

- 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.
- The `TO_CHAR` function's day-of-the-week numbering (see the 'D' ***template pattern*** (page 265)) is different from that of the ***EXTRACT*** (page 218) 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
`SELECT TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS');`	`'Tuesday , 06 05:39:18'`
`SELECT TO_CHAR(CURRENT_TIMESTAMP, 'FMDay, FMDD HH12:MI:SS');`	`'Tuesday, 6 05:39:18'`
`SELECT TO_CHAR(TIMEtz '14:34:06.4+6','HH12:MI am');`	` 04:34 am`
` TO_CHAR`	
`SELECT TO_CHAR(-0.1, '99.99');`	`' -.10'`
`SELECT TO_CHAR(-0.1, 'FM9.99');`	`'-.1'`
`SELECT TO_CHAR(0.1, '0.9');`	`' 0.1'`
`SELECT TO_CHAR(12, '9990999.9');`	`' 0012.0'`
`SELECT TO_CHAR(12, 'FM9990999.9');`	`'0012.'`
`SELECT TO_CHAR(485, '999');`	`' 485'`
`SELECT TO_CHAR(-485, '999');`	`'-485'`
`SELECT TO_CHAR(485, '9 9 9');`	`' 4 8 5'`
`SELECT TO_CHAR(1485, '9,999');`	`' 1,485'`
`SELECT TO_CHAR(1485, '9G999');`	`' 1 485'`
`SELECT TO_CHAR(148.5, '999.999');`	`' 148.500'`
`SELECT TO_CHAR(148.5, 'FM999.999');`	`'148.5'`
`SELECT TO_CHAR(148.5, 'FM999.990');`	`'148.500'`
`SELECT TO_CHAR(148.5, '999D999');`	`' 148,500'`
`SELECT TO_CHAR(3148.5, '9G999D999');`	`' 3 148,500'`
`SELECT TO_CHAR(-485, '999S');`	`'485-'`

```
SELECT TO_CHAR(-485, '999MI');                          '485-'
SELECT TO_CHAR(485, '999MI');                           '485 '
SELECT TO_CHAR(485, 'FM999MI');                         '485'
SELECT TO_CHAR(485, 'PL999');                           '+485'
SELECT TO_CHAR(485, 'SG999');                           '+485'
SELECT TO_CHAR(-485, 'SG999');                          '-485'
SELECT TO_CHAR(-485, '9SG99');                          '4-85'
SELECT TO_CHAR(-485, '999PR');                          '<485>'
SELECT TO_CHAR(485, 'L999');                            'DM 485
SELECT TO_CHAR(485, 'RN');                              '        CDLXXXV'
SELECT TO_CHAR(485, 'FMRN');                            'CDLXXXV'
SELECT TO_CHAR(5.2, 'FMRN');                            'V'
SELECT TO_CHAR(482, '999th');                           ' 482nd'
SELECT TO_CHAR(485, '"Good number:"999');               'Good number: 485'
SELECT TO_CHAR(485.8, '"Pre:"999" Post:" .999');        'Pre: 485 Post: .800'
SELECT TO_CHAR(12, '99V999');                           ' 12000'
SELECT TO_CHAR(12.4, '99V999');                         ' 12400'
SELECT TO_CHAR(12.45, '99V9');                          ' 125'
SELECT TO_CHAR(-1234.567);                              -1234.567
SELECT TO_CHAR('1999-12-25'::DATE);                      1999-12-25
SELECT TO_CHAR('1999-12-25 11:31'::TIMESTAMP);          1999-12-25 11:31:00
SELECT TO_CHAR('1999-12-25 11:31 EST'::TIMESTAMPTZ);    1999-12-25 11:31:00-05
SELECT TO_CHAR('3 days 1000.333 secs'::INTERVAL);       3 days 00:16:40.333
```

TO_DATE

Converts a string value to a DATE type.

Behavior Type

Stable

Syntax

```
TO_DATE ( expression , pattern )
```

Parameters

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 265) and/or ***Template Patterns for Numeric Formatting*** (page 268).

Input Value Considerations

The TO_DATE function requires a CHAR or VARCHAR expression. For other input types, use ***TO_CHAR*** (page 257) to perform an explicit cast to a CHAR or VARCHAR before using this function.

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, TO_TIMESTAMP_TZ, 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:
  - For example `TO_TIMESTAMP('2000  JUN', 'YYYY MON')` is correct.
  - `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 a 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')` is interpreted as a four-digit year

  Instead, use a non-digit separator after the year, such as `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

```
SELECT TO_DATE('13 Feb 2000', 'DD Mon YYYY');
  to_date
------------
 2000-02-13
(1 row)
```

See Also

Template Pattern Modifiers for Date/Time Formatting (page 267)

TO_HEX

Returns a VARCHAR or VARBINARY representing the hexadecimal equivalent of a number.

Behavior Type

Immutable

Syntax

`TO_HEX ( number )`

Parameters

number	(INTEGER) is the number to convert to hexadecimal

Notes

VARCHAR TO_HEX(INTEGER) and VARCHAR TO_HEX(VARBINARY) are similar. The function converts data from binary type to character type (where the character representation is in hexadecimal format). This function is the inverse of HEX_TO_BINARY.

```
TO_HEX(HEX_TO_BINARY(x)) = x).
HEX_TO_BINARY(TO_HEX(x)) = x).
```

Examples

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

For VARBINARY inputs, the returned value is not preceded by "0x". For example:

```
SELECT TO_HEX('ab'::binary(2));
 to_hex
--------
 6162
(1 row)
```

TO_TIMESTAMP

Converts a string value or a UNIX/POSIX epoch value to a `TIMESTAMP` type. Output depends on the current session time zone.

Behavior Type

Stable

Syntax

```
TO_TIMESTAMP ( expression, pattern )
TO_TIMESTAMP ( unix-epoch )
```

Parameters

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 265) and/or ***Template Patterns for Numeric Formatting*** (page 268).
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 ***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, 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\\"'`
- TZ/tz are not supported patterns for the TO_TIMESTAMP function; for example, the following command returns an error:

  ```
  SELECT TO_TIMESTAMP('01-01-01 01:01:01+03:00','DD-MM-YY
  HH24:MI:SSTZ');
  ERROR: "TZ"/"tz" not supported
  ```
- TO_TIMESTAMP, TO_TIMESTAMP_TZ, 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:
  - For example `TO_TIMESTAMP('2000 JUN', 'YYYY MON')` is correct.
  - `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 a 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')` is interpreted as a four-digit year

  Instead, use a non-digit separator after the year, such as `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

```
=> SELECT TO_TIMESTAMP('13 Feb 2009', 'DD Mon YYY');
    TO_TIMESTAMP
---------------------
 1200-02-13 00:00:00
(1 row)

VMart=> SELECT TO_TIMESTAMP(200120400);
    TO_TIMESTAMP
```

```
---------------------
 1976-05-05 01:00:00
(1 row)
```

See Also

Template Pattern Modifiers for Date/Time Formatting (page 267)

TO_TIMESTAMP_TZ

Converts a string value or a UNIX/POSIX epoch value to a `TIMESTAMP WITH TIME ZONE` type.

Behavior Type

Stable

Syntax

```
TO_TIMESTAMP_TZ ( expression, pattern )
TO_TIMESTAMP ( unix-epoch )
```

Parameters

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 265) and/or ***Template Patterns for Numeric Formatting*** (page 268).
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 ***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, 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, TO_TIMESTAMP_TZ, 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:
  - For example `TO_TIMESTAMP('2000  JUN', 'YYYY MON')` is correct.
  - `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 a 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')` is interpreted as a four-digit year

  Instead, use a non-digit separator after the year, such as `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

```
=>  SELECT TO_TIMESTAMP_TZ('13 Feb 2009', 'DD Mon YYY');
    TO_TIMESTAMP_TZ
------------------------
 1200-02-13 00:00:00-05
(1 row)


=> SELECT TO_TIMESTAMP_TZ(200120400);
    TO_TIMESTAMP_TZ
------------------------
 1976-05-05 01:00:00-04
(1 row)
```

See Also

Template Pattern Modifiers for Date/Time Formatting (page 267)

TO_NUMBER

Converts a string value to DOUBLE PRECISION.

Behavior Type

Stable

Syntax

```
TO_NUMBER ( expression, [ pattern ] )
```

Parameters

expression	(CHAR or VARCHAR) specifies the string to convert.
pattern	(CHAR or VARCHAR) Optional parameter specifies an output pattern string using the ***Template Patterns for Date/Time Formatting*** (page 265) and/or ***Template Patterns for Numeric Formatting*** (page 268). If omitted, function returns a floating point.

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(2009, 'rn'), TO_NUMBER('mmix', 'rn');
     to_char     | to_number
-----------------+-----------
            mmix |      2009
(1 row)
```

It the `pattern` parameter is omitted, the function returns a floating point.

```
SELECT TO_NUMBER('-123.456e-01');
 to_number
-----------
  -12.3456
```

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 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.

Note: HP Vertica uses the ISO 8601:2004 style for date/time fields in HP Vertica `*.log` files. For example,
`2008-09-16 14:40:59.123 TM Moveout:0x2aaaac002180 [Txn] <INFO>`

Certain modifiers can be applied to any template pattern to alter its behavior as described in ***Template Pattern Modifiers for Date/Time Formatting*** (page 267).

Pattern	Description
`HH`	Hour of day (00-23)
`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 p.m.	Meridian indicator (lowercase)
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) for TIMESTAMP **Note:** For INTERVAL, DD is day of year (001-366) because day of month is undefined.
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
AM	Time is before 12:00
AT	Ignored
JULIAN, JD, J	Next field is Julian Day
FM prefix	Fill mode (suppress padding blanks and zeros) For example: FMMonth **Note:** The FM modifier suppresses leading zeros and trailing blanks that would otherwise be added to make the output of a pattern fixed width.
FX prefix	Fixed format global option For example: FX Month DD Day
ON	Ignored
PM	Time is on or after 12:00
T	Next field is time
TH suffix	Uppercase ordinal number suffix For example: DDTH
th suffix	Lowercase ordinal number suffix For example: DDth

TM prefix	Translation mode (print localized day and month names based on lc_messages). For example: TMMonth

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)

Usage

- A sign formatted using SG, PL, or MI is not anchored to the number; for example:
  - TO_CHAR(-12, 'S9999') produces ' -12'
  - TO_CHAR(-12, 'MI9999') produces '- 12'
- 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.
- 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.

Geospatial Package SQL Functions

The HP Vertica Geospatial package contains a suite of geospatial SQL functions you can install to report on and analyze geographic location data.

To install the Geospatial package:

Run the install.sh script that appears in the `/opt/vertica/packages/geospatial` directory.

If you choose to install the Geospatial package in a directory other than the default, be sure to set the GEOSPATIAL_HOME environment variable to reflect the correct directory.

Contents of the Geospatial package

When you installed HP Vertica, the RPM saved the Geospatial package files here:

```
/opt/vertica/packages/geospatial
```

This directory contains these files:

`install.sh`	Installs the Geospatial package.
`readme.txt`	Contains instructions for installing the package.

This directory also contains these directories:

`/src`	Contains this file: ▪ `geospatial.sql`—This file contains all the functions that are installed with the package. The file describes the calculations used for each function, and provides examples. This file also contains links to helpful sites that provide more information about standards and calculations.
`/examples`	Contains this file: ▪ `regions_demo.sql`—This file is a demo, intended to illustrate a simple use case: determine the New England state in which a given point lies.

Using Geospatial Package SQL Functions

For high-level descriptions of all of the functions included in the package, see ***Geospatial SQL Functions*** (page 270). For more detailed information about each function and for links to other useful information, see `/opt/vertica/packages/geospatial/src/geospatial.sql`.

Using built-in HP Vertica functions for Geospatial analysis

Four mathematical functions, automatically installed with HP Vertica, perform geospatial operations:

- ***DEGREES*** (page 304)
- ***DISTANCE*** (page 305)
- ***DISTANCEV*** (page 306)

- ***RADIANS*** (page 313)

These functions are not part of the Vertica Geospatial Package; they are installed with HP Vertica.

Geospatial SQL Functions

With the Geospatial Package, HP Vertica provides SQL functions that let you find geographic constants to use in your calculations and analysis. These functions appear in the file `/opt/vertica/packages/geospatial/src/geospatial.sql`.

You can use these functions as they are supplied; you can also edit the `geospatial.sql` file to change the calculations according to your needs. If you do modify the geospatial functions, be sure to save a copy of your changes in a private location so that your changes are not lost if you upgrade your HP Vertica installation. Note that an upgrade does not overwrite any functions already loaded in your database; the upgrade only overwrites only the `.sql` file containing the function definitions.

These functions measure distances in kilometers and angles in fractional degrees, unless stated otherwise.

Of the several possible definitions of latitude, the geodetic latitude is most commonly used; and this is what the HP Vertica Geospatial Package uses. Latitude goes from +90 degrees at the North Pole to –90 at the South Pole. Longitude 0 is near Greenwich, England. It increases going east to +180 degrees, and decreases going west to –180 degrees. True bearings are measured clockwise from north. For more information, see: ***http://en.wikipedia.org/wiki/Latitude*** (***http://en.wikipedia.org/wiki/Latitude***).

WGS-84 SQL Functions

The following functions return constants determined by the World Geodetic System (WGS) standard, WGS-84.

- ***WGS84_a()*** (page 289)
- ***WGS84_b()*** (page 290)
- ***WGS84_e2()*** (page 290)
- ***WGS84_f()*** (page 291)
- ***WGS84_if()*** (page 291)

Earth Radius, Radius of Curvature, and Bearing SQL Functions

These functions return the earth's radius, radius of curvature, and bearing values.

- ***RADIUS_r(****lat****)*** (page 285)
- ***WGS84_r1****()* (page 291)
- ***RADIUS_SI****()* (page 288)
- ***RADIUS_M(****lat****)*** (page 284)
- ***RADIUS_N (****lat****)*** (page 284)
- ***RADIUS_Ra (****lat****)***
- ***RADIUS_Rv (****lat****)*** (page 287)

- ***RADIUS_Rc (****lat,bearing****)*** (page 286)
- ***BEARING*** *(lat1,lon1,lat2,lon2)* (page 272)
- ***RADIUS_LON*** *(lat)* (page 283)

ECEF Conversion SQL Functions

The following functions convert values to Earth-Centered, Earth-Fixed (ECEF) values. The ECEF system represents positions on *x*, *y*, and *z* axes in meters. (0,0,0) is the center of the earth; *x* is toward latitude 0, longitude 0; *y* is toward latitude 0, longitude 90 degrees; and *z* is toward the North Pole. The height above mean sea level (*h*) is also in meters.

- ***ECEF_x (****lat,lon,h****)*** (page 276)
- ***ECEF_y (****lat,lon,h****)*** (page 276)
- ***ECEF_z (****lat,lon,h****)*** (page 277)
- ***ECEF_chord (****lat1,lon1,h1,lat2,lon2,h2****)*** (page 275)
- ***CHORD_TO_ARC (****chord****)*** (page 273)

Bounding Box SQL Functions

These functions determine whether points are within a bounding box, a rectangular area whose edges are latitude and longitude lines. Bounding box methods allow you to narrow your focus, and they work best on HP Vertica projections that are sorted by latitude, or by region (such as swtate) and then by latitude. These methods also work on projections sorted by longitude.

- ***BB_WITHIN (****lat,lon,llat,llon,ulat,rlon****)*** (page 271)
- ***LAT_WITHIN (****lat,lat0,d****)*** (page 279)
- ***LON_WITHIN (****lon,lat0,lon0,d****)*** (page 282)
- ***LL_WITHIN (****lat,lon,lat0,lon0,d****)*** (page 280)
- ***DWITHIN*** *(lat,lon****,****lat0,lon0,d****)*** (page 274)
- ***LLD_WITHIN (****lat,lon****,****lat0,lon0,d)*** (page 281)
- ***ISLEFT (****x0,y0,x1,y1,x2,y2****)*** (page 278)
- ***RAYCROSSING (****x0,y0,x1,y1,x2,y2****)*** (page 288)

Miles/Kilometer Conversion SQL Functions

These functions convert miles to kilometers and kilometers to miles:

- ***MILES2KM (****miles****)*** (page 283)
- ***KM2MILES (****km****)*** (page 279)

BB_WITHIN

Determines whether a point (*lat*, *lon)* falls within a bounding box defined by its lower-left and upper-right corners. The return value has the type BOOLEAN.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
BB_WITHIN (lat,lon,llat,llon,ulat,rlon)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating the latitude of a given point.
lon	A value of type DOUBLE PRECISION indicating the longitude of a given point.
llat	A value of type DOUBLE PRECISION indicating the latitude used to define the lower-left corner of the bounding box.
llon	A value of type DOUBLE PRECISION indicating the longitude used to define the lower-left corner of the bounding box.
ulat	A value of type DOUBLE PRECISION indicating the latitude used to define the upper-right corner of the bounding box.
rlon	A value of type DOUBLE PRECISION indicating the longitude used in defining the upper-right corner of the bounding box.

Example

The following example determines that the point (14,30) is not contained in the bounding box defined by (23.0,45) and (13,37):

```
=> SELECT BB_WITHIN(14,30,23.0,45,13,37);
 BB_WITHIN
-----------
 f
(1 row)
```

The following example determines that the point (14,30) is contained in the bounding box defined by (13.0,45) and (23,37).

```
=> SELECT BB_WITHIN(14,30,13.0,45,23,37);
 BB_WITHIN
-----------
 t
(1 row)
```

BEARING

Returns the approximate bearing from a starting point to an ending point, in degrees. It assumes a flat earth and is useful only for short distances. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
BEARING (lat1,lon1,lat2,lon2)
```

Parameters

lat1	A value of type DOUBLE PRECISION indicating latitude of the starting point.
lon1	A value of type DOUBLE PRECISION indicating longitude of the starting point.
lat2	A value of type DOUBLE PRECISION indicating latitude of the ending point.
lon2	A value of type DOUBLE PRECISION indicating longitude of the ending point.

Example

The following examples calculate the bearing, in degrees, from point (45,13) to (33,3) and from point (33,3) to (45,13):

```
=> SELECT BEARING(45,13,33,3);
      BEARING
-------------------
 -140.194428907735
(1 row)


=> SELECT BEARING(33,3,45,13);
     BEARING
------------------
 39.8055710922652
(1 row)
```

CHORD_TO_ARC

Converts a chord (the straight line between two points) in meters to a geodesic arc length, also in meters. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
CHORD_TO_ARC (chord)
```

Parameters

chord	A value of type DOUBLE PRECISION indicating chord length (in meters)

Example

The following examples convert the length of a chord to the length of its geodesic arc:

```
=> SELECT CHORD_TO_ARC(120);
   CHORD_TO_ARC
------------------
 120.000000001774
(1 row)


=> SELECT CHORD_TO_ARC(12000);
   CHORD_TO_ARC
------------------
 12000.0017738474
(1 row)


=> SELECT CHORD_TO_ARC(1200000);
   CHORD_TO_ARC
------------------
 1201780.96402514
(1 row)
```

DWITHIN

Determines whether a point (*lat*,*lon*) is within a circle of radius *d* kilometers centered at a given point (*lat0*,*lon0*). The return value has the type BOOLEAN.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
DWITHIN (lat,lon,lat0,lon0,d)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating a given latitude.
lon	A value of type DOUBLE PRECISION indicating a given longitude.
lat0	A value of type DOUBLE PRECISION indicating the latitude of the center point of a circle.

lon0	A value of type DOUBLE PRECISION indicating the longitude of the center point of a circle.
d	A value of type DOUBLE PRECISION indicating the radius of the circle (in kilometers).

Example

The following examples determine that the point (13.6,43.5) is within 3880–3890 kilometers of the radius of a circle centered at (48.5,45.5):

```
=> SELECT DWITHIN(13.6,43.5,48.5,45.5,3880);
 DWITHIN
---------
 f
(1 row)


=> SELECT DWITHIN(13.6,43.5,48.5,45.5,3890);
 DWITHIN
---------
 t
(1 row)
```

ECEF_CHORD

Calculates the distance in meters between two ECEF coordinates. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
ECEF_CHORD (lat1,lon1,h1,lat2,lon2,h2)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating the latitude of one end point of the line.
lon1	A value of type DOUBLE PRECISION indicating the longitude of one end point of the line.
h1	A value of type DOUBLE PRECISION indicating the height above sea level (in meters) of one end point of the line.
lat2	A value of type DOUBLE PRECISION indicating the latitude of one end point of the line.
lon2	A value of type DOUBLE PRECISION indicating the longitude of one end point of the line.

h2	A value of type DOUBLE PRECISION indicating the height of one end point of the line.

Example

The following example calculates the distance in meters between the ECEF coordinates (-12,10.0,14) and (12,-10,17):

```
=> SELECT ECEF_chord (-12,10.0,14,12,-10,17);
    ECEF_chord
------------------
 3411479.93992789
(1 row)
```

ECEF_x

Converts a given latitude, longitude, and height into the ECEF *x* coordinate in meters. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
ECEF_x (lat,lon,h)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude.
lon	A value of type DOUBLE PRECISION indicating longitude.
h	A value of type DOUBLE PRECISION indicating height.

Example

The following example calculates the ECEF *x* coordinate in meters for the point (-12,13.2,0):

```
=> SELECT ECEF_x(-12,13.2,0);
      ECEF_x
------------------
 6074803.56179976
(1 row)
```

ECEF_y

Converts a given latitude, longitude, and height into the ECEF *y* coordinate in meters. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
ECEF_y (lat,lon,h)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude.
lon	A value of type DOUBLE PRECISION indicating longitude.
h	A value of type DOUBLE PRECISION indicating height.

Example

The following example calculates the ECEF *y* coordinate in meters for the point (12.0,-14.2,12):

```
=> SELECT ECEF_y(12.0,-14.2,12);
      ECEF_y
-------------------
 -1530638.12327962
(1 row)
```

ECEF_z

Converts a given latitude, longitude, and height into the ECEF *z* coordinate in meters. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
ECEF_Z (lat,lon,h)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude.
lon	A value of type DOUBLE PRECISION indicating longitude.
h	A value of type DOUBLE PRECISION indicating height.

Example

The following example calculates the ECEF *z* coordinate in meters for the point (12.0,-14.2,12):

```
=> SELECT ECEF_z(12.0,-14.2,12);
      ECEF_z
------------------
```

```
 1317405.02616989
(1 row)
```

ISLEFT

Determines whether a given point is anywhere to the left of a directed line that goes though two specified points. The return value has the type FLOAT and has the following possible values:

- > 0: The point is to the left of the line.
- = 0: The point is on the line.
- < 0: The point is to the right of the line.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
ISLEFT (x0,y0,x1,y1,x2,y2)
```

Parameters

x0	A value of type DOUBLE PRECISION indicating the latitude of the first point through which the directed line passes.
y0	A value of type DOUBLE PRECISION indicating the longitude of the the first point through which the directed line passes.
x1	A value of type DOUBLE PRECISION indicating the latitude of the second point through which the directed line passes.
y1	A value of type DOUBLE PRECISION indicating the longitude of the the second point through which the directed line passes.
x2	A value of type DOUBLE PRECISION indicating the latitude of the point whose position you are evaluating.
y2	A value of type DOUBLE PRECISION indicating the longitude of a whose position you are evaluating.

Example

The following example determines that (0,0) is to the left of the line that passes through (1,1) and (2,3):

```
=> SELECT ISLEFT(1,1,2,3,0,0);
 ISLEFT
--------
      1
(1 row)
```

KM2MILES

Converts a value from kilometers to miles. The return value is of type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
KM2MILES (km)
```

Parameters

km	A value of type DOUBLE PRECISION indicating the number of kilometers you want to convert.

Example

The following example converts 1.0 kilometers to miles:

```
=> SELECT KM2MILES(1.0);
     KM2MILES
-------------------
 0.621371192237334
(1 row)
```

LAT_WITHIN

Determines whether a certain latitude *(lat)* is within *d* kilometers of another latitude point *(lat0)*, independent of longitude. The return value has the type BOOLEAN.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
LAT_WITHIN (lat,lat0,d)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating a given latitude.
lat0	A value of type DOUBLE PRECISION indicating the latitude of the point to which you are comparing the first latitude.
d	A value of type DOUBLE PRECISION indicating the number of kilometers that determines the range you are evaluating.

Example

The following examples determine that latitude 12 is between 220 and 230 kilometers of latitude 14.0:

```
=> SELECT LAT_WITHIN(12,14.0,220);
 LAT_WITHIN
------------
 f
(1 row)


=> SELECT LAT_WITHIN(12,14.0,230);
 LAT_WITHIN
------------
 t
(1 row)
```

LL_WITHIN

Determines whether a point (*lat*, *lon*) is within a bounding box whose sides are 2*d* kilometers long, centered at a given point (*lat0*, *lon0*). The return value has the type BOOLEAN.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
LL_WITHIN (lat,lon,lat0,lon0,d);
```

Parameters

lat	A value of type DOUBLE PRECISION indicating a given latitude.
lon	A value of type DOUBLE PRECISION indicating a given longitude.
lat0	A value of type DOUBLE PRECISION indicating the latitude of the center point of the bounding box.
lon0	A value of type DOUBLE PRECISION indicating the longitude of the center point of the bounding box.
d	A value of type DOUBLE PRECISION indicating the length of half the side of the box.

Example

The following examples determine that the point (16,15) is within a bounding box centered at (12,13) whose sides are between 880 and 890 kilometers long:

```
=> SELECT LL_WITHIN(16,15,12,13.0,440);
 LL_WITHIN
```

```
-----------
 f
(1 row)


=> SELECT LL_WITHIN(16,15,12,13.0,445);
 LL_WITHIN
-----------
 t
(1 row)
```

LLD_WITHIN

Determines whether a point *(lat,lon)* is within a circle of radius *d* kilometers centered at a given point *(lat0,lon0)*. LLD_WITHIN is a faster, but less accurate version of ***DWITHIN*** (page 274). The return value has the type BOOLEAN.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
LLD_WITHIN (lat,lon,lat0,lon0,d)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating a given latitude.
lon	A value of type DOUBLE PRECISION indicating a given longitude.
lat0	A value of type DOUBLE PRECISION indicating the latitude of the center point of a circle.
lon0	A value of type DOUBLE PRECISION indicating the longitude of the center point of a circle.
d	A value of type DOUBLE PRECISION indicating the radius of the circle (in kilometers).

Example

The following examples determine that the point (13.6,43.5) is within a circle centered at (48.5,45.5) whose radius is between 3800 and 3900 kilometers long:

```
=> SELECT LLD_WITHIN(13.6,43.5,48.5,45.5,3800);
 LLD_WITHIN
------------
 f
(1 row)


=> SELECT LLD_WITHIN(13.6,43.5,48.5,45.5,3900);
```

```
 LLD_WITHIN
------------
 t
(1 row)
```

LON_WITHIN

Determines whether a longitude *(lon)* is within *d* kilometers of a given point *(lat0, lon0)*. The return value has the type BOOLEAN.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
LON_WITHIN (lon,lat0,lon0,d)
```

Parameters

lon	A value of type DOUBLE PRECISION indicating a given longitude.
lat0	A value of type DOUBLE PRECISION indicating the latitude of the point to which you want to compare the *lon* value.
lon0	A value of type DOUBLE PRECISION indicating the longitude of the point to which you want to compare the *lon* value.
d	A value of type DOUBLE PRECISION indicating the distance, in kilometers, that defines your range.

Example

The following examples determine that the longitude 15 is between 1600 and 1700 kilometers from the point (16,0):

```
=> SELECT LON_WITHIN(15,16,0,1600);
 LON_WITHIN
------------
 f
(1 row)


=> SELECT LON_WITHIN(15,16,0,1700);
 LON_WITHIN
------------
 t
(1 row)
```

MILES2KM

Converts a value from miles to kilometers. The return value is of type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
MILES2KM (miles)
```

Parameters

miles	A value of type DOUBLE PRECISION indicating the number of miles you want to convert.

Example

The following example converts 1.0 miles to kilometers:

```
=> SELECT MILES2KM(1.0);
 MILES2KM
----------
 1.609344
(1 row)
```

RADIUS_LON

Returns the radius of the circle of longitude in kilometers at a given latitude. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RADIUS_LON (lat)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude at which you want to measure the radius.

Example

The following example calculates the circle of longitude in kilometers at a latitude of 45:

```
=> SELECT RADIUS_LON(45);
     RADIUS_LON
--------------------
 4517.59087884893
(1 row)
```

RADIUS_M

Returns the earth's radius of curvature in kilometers along the meridian at the given latitude. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RADIUS_M (lat)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude at which you want to measure the radius of curvature.

Example

The following example calculates the earth's radius of curvature in kilometers along the meridian at latitude –90 (the South Pole):

```
=> SELECT RADIUS_M(-90);
    RADIUS_M
-----------------
 6399.5936257585
(1 row)
```

RADIUS_N

Returns the earth's radius of curvature in kilometer normal to the meridian at a given latitude. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RADIUS_N (lat)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude at which you want to measure the radius of curvature.

Example

The following example calculates the earth's radius of curvature in kilometers normal to the meridian at latitude –90 (the South Pole):

```
=> SELECT RADIUS_N(-90);
     RADIUS_N
------------------
 6399.59362575849
(1 row)
```

RADIUS_R

Returns the WGS-84 radius of the earth (to the center of mass) in kilometers at a given latitude. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RADIUS_R (lat)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude at which you want to measure the earth's radius.

Example

The following example calculates the WGS-84 radius of the earth in kilometers at latitude –90 (the South Pole):

```
=> SELECT RADIUS_R(-90);
     RADIUS_R
------------------
 6356.75231424518
(1 row)
```

RADIUS_Ra

Returns the earth's average radius of curvature in kilometers at a given latitude. This function is the geometric mean of ***RADIUS_M*** (page 284) and ***RADIUS_N*** (page 284). (***RADIUS_Rv*** (page 287) is a faster approximation of this function.)

The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RADIUS_Ra (lat)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude at which you want to measure the radius of curvature.

Example

The following example calculates the earth's average radius of curvature in kilometers at latitude –90 (the South Pole):

```
=> SELECT RADIUS_Ra(-90);
    RADIUS_Ra
------------------
 6399.59362575849
(1 row)
```

RADIUS_Rc

Returns the earth's radius of curvature in kilometers at a given bearing measured clockwise from north. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RADIUS_Rc (lat, bearing)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude at which you want to measure the radius of curvature.
bearing	A value of type DOUBLE PRECISION indicating a given bearing.

Example

The following example measures the earth's radius of curvature in kilometers at latitude 45, with a bearing of 45 measured clockwise from north:

```
=> SELECT RADIUS_Rc(45,45);
    RADIUS_Rc
------------------
 6378.09200754445
(1 row)
```

RADIUS_Rv

Returns the earth's average radius of curvature in kilometers at a given latitude. This value is the geometric mean of ***RADIUS_M*** (page 284) and ***RADIUS_N*** (page 284). This function is a fast approximation of ***RADIUS_Ra***. The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RADIUS_Rv (lat)
```

Parameters

lat	A value of type DOUBLE PRECISION indicating latitude at which you want to measure the radius of curvature.

Example

The following example calculates the earth's average radius of curvature in kilometers at latitude –90 (the South Pole):

```
=> SELECT RADIUS_Rv(-90);
    RADIUS_Rv
------------------
 6399.59362575849
(1 row)
```

RADIUS_SI

Returns the International System of Units (SI) radius based on the nautical mile. (A nautical mile is a unit of length about one minute of arc of latitude measured along any meridian, or about one minute of arc of longitude measured at the equator.) The return value has the type NUMERIC.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RADIUS_SI ()
```

Example

The following example calculates the SI radius based on the nautical mile:

```
=> SELECT RADIUS_SI();
      RADIUS_SI
---------------------
 6366.70701949370750
(1 row)
```

RAYCROSSING

Determines whether a ray traveling to the right from point (*x2,y2*), in the direction of increasing *x*, intersects a directed line segment that starts at point (*x0,y0*) and ends at point (*x1,y1*).

This function returns:

- 0 if the ray does not intersect the directed line segment.
- 1 if the ray intersects the line and *y1* is above *y0*.
- –1 if the ray intersects the line and *y1* is below or equal to *y0*.

The return value has the type DOUBLE PRECISION.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
RAYCROSSING (x0,y0,x1,y1,x2,y2)
```

Parameters

x0	A value of type DOUBLE PRECISION indicating the latitude of the

	starting point of the line segment.
y0	A value of type DOUBLE PRECISION indicating the longitude of the starting point of the line segment
x1	A value of type DOUBLE PRECISION indicating the latitude of the ending point of the line segment.
y1	A value of type DOUBLE PRECISION indicating the longitude of the the ending point of the line segment.
x2	A value of type DOUBLE PRECISION indicating the latitude of the point from which the ray starts.
y2	A value of type DOUBLE PRECISION indicating the longitude of the point from which the ray starts.

Example

The following example checks if a line traveling to the right from the point (0,0) intersects the line from (1,1) to (2,3):

```
=> SELECT RAYCROSSING(1,1,2,3,0,0);
 RAYCROSSING
-------------
           0
(1 row)
```

The following example checks if a line traveling to the right from the point (0,2) intersects the line from (1,1) to (2,3):

```
=> SELECT RAYCROSSING(1,1,2,3,0,2);
 RAYCROSSING
-------------
           1
(1 row)
```

The following example checks if a line traveling to the right from the point (0,2) intersects the line from (1,3) to (2,1):

```
=> SELECT RAYCROSSING(1,3,2,1,0,2);
 RAYCROSSING
-------------
          -1
(1 row)
```

WGS84_a

Returns the length, in kilometers, of the earth's semi-major axis. The return value is of type NUMERIC.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
WGS84_a ()
```

Example

```
=> SELECT WGS84_a();
   wgs84_a
-------------
 6378.137000
(1 row)
```

WGS84_b

Returns the WGS-84 semi-minor axis length value in kilometers. The return value is of type NUMERIC.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
WGS84_b ()
```

Example

```
=> SELECT WGS84_b();
       WGS84_b
--------------------
 6356.75231424517950
(1 row)
```

WGS84_e2

Returns the WGS-84 eccentricity squared value. The return value is of type NUMERIC.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
WGS84_e2 ()
```

Example

```
=> SELECT WGS84_e2();
```

```
       WGS84_e2
-----------------------
 .00669437999014131700
(1 row)
```

WGS84_f

Returns the WGS-84 flattening value. The return value is of type NUMERIC.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
WGS84_f ()
```

Example

```
=> SELECT WGS84_f();
        WGS84_f
------------------------
 .0033528106647474807 2
(1 row)
```

WGS84_if

Returns the WGS-84 inverse flattening value. The return value is of type NUMERIC.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
WGS84_if ()
```

Example

```
=> SELECT WGS84_if();
   WGS84_if
---------------
 298.257223563
(1 row)
```

WGS84_r1

Returns the International Union of Geodesy and Geophysics (IUGG) mean radius of the earth, in kilometers. The return value is of type NUMERIC.

This function is available only if you install the HP Vertica Geospatial Package. See ***Geospatial Package SQL Functions*** (page 270) for information on installing the package.

Behavior Type

Immutable

Syntax

```
WGS84_r1 ()
```

Example

```
=> SELECT WGS84_r1();
      WGS84_r1
---------------------
 6371.00877141505983
(1 row)
```

IP Conversion Functions

IP functions perform conversion, calculation, and manipulation operations on IP, network, and subnet addresses.

INET_ATON

Returns an integer that represents the value of the address in host byte order, given the dotted-quad representation of a network address as a string.

Behavior Type

Immutable

Syntax

```
INET_ATON ( expression )
```

Parameters

expression	(VARCHAR) is the string to convert.

Notes

The following syntax converts an IPv4 address represented as the string A to an integer I.

INET_ATON trims any spaces from the right of A, calls the Linux function ***inet_pton http://www.opengroup.org/onlinepubs/000095399/functions/inet_ntop.html***, and converts the result from network byte order to host byte order using ***ntohl http://opengroup.org/onlinepubs/007908775/xns/ntohl.html***.

```
INET_ATON(VARCHAR A) -> INT8 I
```

If A is NULL, too long, or inet_pton returns an error, the result is NULL.

Examples

The generated number is always in host byte order. In the following example, the number is calculated as 209×256^3 + 207×256^2 + 224×256 + 40.

```
SELECT INET_ATON('209.207.224.40');
 inet_aton
------------
 3520061480
(1 row)
SELECT INET_ATON('1.2.3.4');
 inet_aton
-----------
  16909060
(1 row)
SELECT TO_HEX(INET_ATON('1.2.3.4'));
 to_hex
---------
 1020304
(1 row)
```

See Also

INET_NTOA (page 293)

INET_NTOA

Returns the dotted-quad representation of the address as a VARCHAR, given a network address as an integer in network byte order.

Behavior Type

Immutable

Syntax

```
INET_NTOA ( expression )
```

Parameters

expression	(INTEGER) is the network address to convert.

Notes

The following syntax converts an IPv4 address represented as integer I to a string A.

INET_NTOA converts I from host byte order to network byte order using ***htonl http://opengroup.org/onlinepubs/007908775/xns/htonl.html***, and calls the Linux function ***inet_ntop http://www.opengroup.org/onlinepubs/000095399/functions/inet_ntop.html***.

```
INET_NTOA(INT8 I) -> VARCHAR A
```

If I is NULL, greater than 2^32 or negative, the result is NULL.

Examples

```
SELECT INET_NTOA(16909060);
 inet_ntoa
-----------
 1.2.3.4
(1 row)
SELECT INET_NTOA(03021962);
  inet_ntoa
-------------
 0.46.28.138
(1 row)
```

See Also

INET_ATON (page 292)

V6_ATON

Converts an IPv6 address represented as a character string to a binary string.

Behavior Type

Immutable

Syntax

```
V6_ATON ( expression )
```

Parameters

expression	(VARCHAR) is the string to convert.

Notes

The following syntax converts an IPv6 address represented as the character string A to a binary string B.

V6_ATON trims any spaces from the right of A and calls the Linux function ***inet_pton http://www.opengroup.org/onlinepubs/000095399/functions/inet_ntop.html***.

```
V6_ATON(VARCHAR A) -> VARBINARY(16) B
```

If A has no colons it is prepended with '::ffff:'. If A is NULL, too long, or if inet_pton returns an error, the result is NULL.

Examples

```
SELECT V6_ATON('2001:DB8::8:800:200C:417A');
                        v6_aton
-----------------------------------------------------
  \001\015\270\000\000\000\000\000\010\010\000 \014Az
```

```
(1 row)
SELECT TO_HEX(V6_ATON('2001:DB8::8:800:200C:417A'));
              to_hex
----------------------------------
 20010db8000000000080800200c417a
(1 row)
SELECT V6_ATON('1.2.3.4');
                              v6_aton
----------------------------------------------------------------
--

\000\000\000\000\000\000\000\000\000\000\377\377\001\002\003\004
(1 row)
SELECT V6_ATON('::1.2.3.4');
                              v6_aton
----------------------------------------------------------------
--

\000\000\000\000\000\000\000\000\000\000\000\000\001\002\003\004
(1 row)
```

See Also

V6_NTOA (page 295)

V6_NTOA

Converts an IPv6 address represented as varbinary to a character string.

Behavior Type

Immutable

Syntax

```
V6_NTOA ( expression )
```

Parameters

expression	(VARBINARY) is the binary string to convert.

Notes

The following syntax converts an IPv6 address represented as VARBINARY B to a string A.

V6_NTOA right-pads B to 16 bytes with zeros, if necessary, and calls the Linux function ***inet_ntop http://www.opengroup.org/onlinepubs/000095399/functions/inet_ntop.html***.

```
V6_NTOA(VARBINARY B) -> VARCHAR A
```

If B is NULL or longer than 16 bytes, the result is NULL.

HP Vertica automatically converts the form '::ffff:1.2.3.4' to '1.2.3.4'.

Examples

```
SELECT V6_NTOA(' \001\015\270\000\000\000\000\000\010\010\000 \014Az');
          v6_ntoa
--------------------------
 2001:db8::8:800:200c:417a
(1 row)
SELECT V6_NTOA(V6_ATON('1.2.3.4'));
 v6_ntoa
---------
 1.2.3.4
(1 row)
SELECT V6_NTOA(V6_ATON('::1.2.3.4'));
  v6_ntoa
-----------
 ::1.2.3.4
(1 row)
```

See Also

N6_ATON (page 294)

V6_SUBNETA

Calculates a subnet address in CIDR (Classless Inter-Domain Routing) format from a binary or alphanumeric IPv6 address.

Behavior Type

Immutable

Syntax

```
V6_SUBNETA ( expression1, expression2 )
```

Parameters

expression1	(VARBINARY or VARCHAR) is the string to calculate.
expression2	(INTEGER) is the size of the subnet.

Notes

The following syntax calculates a subnet address in CIDR format from a binary or varchar IPv6 address.

V6_SUBNETA masks a binary IPv6 address B so that the N leftmost bits form a subnet address, while the remaining rightmost bits are cleared. It then converts to an alphanumeric IPv6 address, appending a slash and N.

```
V6_SUBNETA(BINARY B, INT8 N) -> VARCHAR C
```

The following syntax calculates a subnet address in CIDR format from an alphanumeric IPv6 address.

```
V6_SUBNETA(VARCHAR A, INT8 N) -> V6_SUBNETA(V6_ATON(A), N) -> VARCHAR C
```

Examples

```
SELECT V6_SUBNETA(V6_ATON('2001:db8::8:800:200c:417a'), 28);
  v6_subneta
---------------
 2001:db0::/28
(1 row)
```

See Also

V6_SUBNETN (page 297)

V6_SUBNETN

Calculates a subnet address in CIDR (Classless Inter-Domain Routing) format from a varbinary or alphanumeric IPv6 address.

Behavior Type

Immutable

Syntax

```
V6_SUBNETN ( expression1, expression2 )
```

Parameters

expression1	(VARBINARY or VARCHAR) is the string to calculate. **Notes:** ▪ V6_SUBNETN(<*VARBINARY*>, <*INTEGER*>) returns varbinary. OR ▪ V6_SUBNETN(<*VARCHAR*>, <*INTEGER*>) returns varbinary, after using V6_ATON to convert the <*VARCHAR*> string to <*VARBINARY*>.
expression2	(INTEGER) is the size of the subnet.

Notes

The following syntax masks a BINARY IPv6 address **B** so that the N left-most bits of **S** form a subnet address, while the remaining right-most bits are cleared.

V6_SUBNETN right-pads B to 16 bytes with zeros, if necessary and masks B, preserving its N-bit subnet prefix.

```
V6_SUBNETN(VARBINARY B, INT8 N) -> VARBINARY(16) S
```

If B is NULL or longer than 16 bytes, or if N is not between 0 and 128 inclusive, the result is NULL.

S = [B]/N in ***Classless Inter-Domain Routing http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing*** notation (CIDR notation).

The following syntax masks an alphanumeric IPv6 address **A** so that the N leftmost bits form a subnet address, while the remaining rightmost bits are cleared.

```
V6_SUBNETN(VARCHAR A, INT8 N) -> V6_SUBNETN(V6_ATON(A), N) -> VARBINARY(16) S
```

Example

This example returns VARBINARY, after using V6_ATON to convert the VARCHAR string to VARBINARY:

```
=> SELECT V6_SUBNETN(V6_ATON('2001:db8::8:800:200c:417a'), 28);
                          v6_subnetn
-----------------------------------------------------------------
  \001\015\260\000\000\000\000\000\000\000\000\000\000\000\000
```

See Also

V6_ATON (page 294)

V6_SUBNETA (page 296)

V6_TYPE

Characterizes a binary or alphanumeric IPv6 address B as an integer type.

Behavior Type

Immutable

Syntax

```
V6_TYPE ( expression )
```

Parameters

expression	(VARBINARY or VARCHAR) is the type to convert.

Notes

V6_TYPE(VARBINARY B) returns INT8 T.

```
V6_TYPE(VARCHAR A) -> V6_TYPE(V6_ATON(A)) -> INT8 T
```

The IPv6 types are defined in the Network Working Group's ***IP Version 6 Addressing Architecture memo http://www.ietf.org/rfc/rfc4291***.

```
  GLOBAL = 0       Global unicast addresses
  LINKLOCAL = 1    Link-Local unicast (and Private-Use) addresses
  LOOPBACK = 2     Loopback
  UNSPECIFIED = 3  Unspecified
  MULTICAST = 4    Multicast
```

IPv4-mapped and IPv4-compatible IPv6 addresses are also interpreted, as specified in ***IPv4 Global Unicast Address Assignments http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml***.

- For IPv4, Private-Use is grouped with Link-Local.
- If B is VARBINARY, it is right-padded to 16 bytes with zeros, if necessary.
- If B is NULL or longer than 16 bytes, the result is NULL.

Details

IPv4 (either kind):

```
0.0.0.0/8         UNSPECIFIED
10.0.0.0/8        LINKLOCAL
127.0.0.0/8       LOOPBACK
169.254.0.0/16    LINKLOCAL
172.16.0.0/12     LINKLOCAL
192.168.0.0/16    LINKLOCAL
224.0.0.0/4       MULTICAST
others            GLOBAL
```

IPv6:

```
::0/128           UNSPECIFIED
::1/128           LOOPBACK
fe80::/10         LINKLOCAL
ff00::/8          MULTICAST
others            GLOBAL
```

Examples

```
SELECT V6_TYPE(V6_ATON('192.168.2.10'));
 v6_type
---------
       1
(1 row)
SELECT V6_TYPE(V6_ATON('2001:db8::8:800:200c:417a'));
 v6_type
---------
       0
(1 row)
```

See Also

INET_ATON (page 292)

IP Version 6 Addressing Architecture http://www.ietf.org/rfc/rfc4291

IPv4 Global Unicast Address Assignments http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml

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 105) data could vary in accuracy and behavior in boundary cases depending on the host system.

See Also

Template Patterns for Numeric Formatting (page 267)

ABS

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

Behavior Type

Immutable

Syntax

```
ABS ( expression )
```

Parameters

expression	Is a value of type INTEGER or DOUBLE PRECISION

Examples

```
SELECT ABS(-28.7);
 abs
------
 28.7
(1 row)
```

ACOS

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

Behavior Type

Immutable

Syntax

```
ACOS ( expression )
```

Parameters

expression	Is a value of type DOUBLE PRECISION

Example

```
SELECT ACOS (1);
 acos
------
    0
(1 row)
```

ASIN

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

Behavior Type

Immutable

Syntax

```
ASIN ( expression )
```

Parameters

expression	Is a value of type DOUBLE PRECISION

Example

```
SELECT ASIN(1);
      asin
-----------------
 1.5707963267949
(1 row)
```

ATAN

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

Behavior Type

Immutable

Syntax

```
ATAN ( expression )
```

Parameters

expression	Is a value of type DOUBLE PRECISION

Example

```
SELECT ATAN(1);
       atan
-------------------
 0.785398163397448
(1 row)
```

ATAN2

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

Behavior Type

Immutable

Syntax

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

Parameters

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

Example

```
SELECT ATAN2(2,1);
      atan2
------------------
 1.10714871779409
(1 row)
```

CBRT

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

Behavior Type

Immutable

Syntax

```
CBRT ( expression )
```

Parameters

expression	Is a value of type DOUBLE PRECISION

Examples

```
SELECT CBRT(27.0);
```

```
 cbrt
------
    3
(1 row)
```

CEILING (CEIL)

Rounds the returned value up to the next whole number. Any expression that contains even a slight decimal is rounded up.

Behavior Type

Immutable

Syntax

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

Parameters

expression	Is a value of type INTEGER or DOUBLE PRECISION

Notes

CEILING is the opposite of ***FLOOR*** (page 307), which rounds the returned value down:

```
=> SELECT CEIL(48.01) AS ceiling, FLOOR(48.01) AS floor;
 ceiling | floor
---------+-------
      49 |    48
(1 row)
```

Examples

```
=> SELECT CEIL(-42.8);
 CEIL
------
  -42
(1 row)
SELECT CEIL(48.01);
 CEIL
------
   49
(1 row)
```

COS

Returns a DOUBLE PRECISION value representing the trigonometric cosine of the argument.

Behavior Type

Immutable

Syntax

```
COS ( expression )
```

Parameters

expression	Is a value of type DOUBLE PRECISION

Example

```
SELECT COS(-1);
       cos
------------------
 0.54030230586814
(1 row)
```

COT

Returns a DOUBLE PRECISION value representing the trigonometric cotangent of the argument.

Behavior Type

Immutable

Syntax

```
COT ( expression )
```

Parameters

expression	Is a value of type DOUBLE PRECISION

Example

```
SELECT COT(1);
        cot
-------------------
 0.642092615934331
(1 row)
```

DEGREES

Converts an expression from ***radians*** (page 313) to fractional degrees, or from degrees, minutes, and seconds to fractional degrees. The return value has the type DOUBLE PRECISION.

Behavior Type

Immutable

Syntax
```
DEGREES (radians)
```

Syntax2
```
DEGREES (degrees, minutes, seconds)
```

Parameters

radians	A unit of angular measure, 2π radians is equal to a full rotation.
degrees	A unit of angular measure, equal to 1/360 of a full rotation.
minutes	A unit of angular measurement, representing 1/60 of a degree.
seconds	A unit of angular measurement, representing 1/60 of a minute.

Examples
```
SELECT DEGREES(0.5);
     DEGREES
------------------
 28.6478897565412
(1 row)

SELECT DEGREES(1,2,3);
     DEGREES
------------------
 1.03416666666667
(1 row)
```

DISTANCE

Returns the distance (in kilometers) between two points. You specify the latitude and longitude of both the starting point and the ending point. You can also specify the radius of curvature for greater accuracy when using an ellipsoidal model.

Behavior type

Immutable

Syntax
```
DISTANCE ( lat0, lon0, lat1, lon1, radius_of_curvature )
```

Parameters

lat0	Specifies the latitude of the starting point.
lon0	Specifies the longitude of the starting point.
lat1	Specifies the latitude of the ending point
lon1	Specifies the longitude of the ending point.

radius_of_curvature	Specifies the radius of the curvature of the earth at the midpoint between the starting and ending points. This parameter allows for greater accuracy when using an ellipsoidal earth model. If you do not specify this parameter, it defaults to the WGS-84 average r1 radius, about 6371.009 km.

Examples

This example finds the distance in kilometers for 1 degree of longitude at latitude 45 degrees, assuming earth is spherical.

```
SELECT DISTANCE(45,0, 45,1);
       distance
---------------------
 78.6262959272162
(1 row)
```

DISTANCEV

Returns the distance (in kilometers) between two points using the Vincenty formula. Because the Vincenty formula includes the parameters of the WGS-84 ellipsoid model, you need not specify a radius of curvature. You specify the latitude and longitude of both the starting point and the ending point. This function is more accurate, but will be slower, than the DISTANCE function.

Behavior type

Immutable

Syntax

```
DISTANCEV (lat0, lon0, lat1, lon1);
```

Parameter

lat0	Specifies the latitude of the starting point.
lon0	Specifies the longitude of the starting point.
lat1	Specifies the latitude of the ending point
lon1	Specifies the longitude of the ending point.

Examples

This example finds the distance in kilometers for 1 degree of longitude at latitude 45 degrees, assuming earth is ellipsoidal.

```
SELECT DISTANCEV(45,0, 45,1);
    distanceV
------------------
```

```
 78.8463347095916
(1 row)
```

EXP

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

Behavior Type

Immutable

Syntax

```
EXP ( exponent )
```

Parameters

exponent	Is an expression of type INTEGER or DOUBLE PRECISION

Example

```
SELECT EXP(1.0);
       exp
------------------
 2.71828182845905
(1 row)
```

FLOOR

Rounds the returned value down to the next whole number. For example, each of these functions evaluates to 5:

```
floor(5.01)
floor(5.5)
floor(5.99)
```

Behavior Type

Immutable

Syntax

```
FLOOR ( expression )
```

Parameters

expression	Is an expression of type INTEGER or DOUBLE PRECISION.

Notes

FLOOR is the opposite of ***CEILING*** (page 303), which rounds the returned value up:

```
=> SELECT FLOOR(48.01) AS floor, CEIL(48.01) AS ceiling;
 floor | ceiling
-------+---------
    48 |      49
(1 row)
```

Examples

```
=> SELECT FLOOR((TIMESTAMP '2005-01-17 10:00' - TIMESTAMP '2005-01-01') / INTERVAL '7');
 floor
-------
     2
(1 row)
=> SELECT FLOOR(-42.8);
 floor
-------
   -43
(1 row)
=> SELECT FLOOR(42.8);
 floor
-------
    42
(1 row)
```

Although the following example looks like an INTEGER, the number on the left is 2^49 as an INTEGER, but the number on the right is a FLOAT:

```
=> SELECT 1<<49, FLOOR(1 << 49);
    ?column?     |      floor
-----------------+-----------------
 562949953421312 | 562949953421312
(1 row)
```

Compare the above example to:

```
=> SELECT 1<<50, FLOOR(1 << 50);
     ?column?     |        floor
------------------+---------------------
 1125899906842624 | 1.12589990684262e+15
(1 row)
```

HASH

Calculates a hash value over its arguments, producing a value in the range 0 <= x < 2^{63} (two to the sixty-third power or 2^63).

Behavior Type

Immutable

Syntax

```
HASH ( expression [ ,... ] )
```

Parameters

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 54).

Notes

- The HASH() function is used to provide projection segmentation over a set of nodes in a cluster and takes up to 32 arguments, usually column names, and selects a specific node for each row based on the values of the columns for that row. HASH (Col1, Col2).
- If your data is fairly regular and you want more even distribution than you get with HASH, consider using ***MODULARHASH*** (page 311)() for project segmentation.

Examples

```
SELECT HASH(product_price, product_cost)
FROM product_dimension
WHERE product_price = '11';
        hash
--------------------
 4157497907121511878
 1799398249227328285
 3250220637492749639
(3 rows)
```

See Also

MODULARHASH (page 311)

LN

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

Behavior Type

Immutable

Syntax

```
LN ( expression )
```

Parameters

expression	Is an expression of type INTEGER or DOUBLE PRECISION

Examples

```
SELECT LN(2);
        ln
-------------------
 0.693147180559945
(1 row)
```

LOG

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

Behavior Type

Immutable

Syntax

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

Parameters

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

Examples

```
SELECT LOG(2.0, 64);
 log
-----
   6
(1 row)
SELECT LOG(100);
 log
-----
   2
(1 row)
```

MOD

Returns the remainder of a division operation. MOD is also called `modulo`.

Behavior Type

Immutable

Syntax

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

Parameters

expression1	Specifies the dividend (INTEGER, NUMERIC, or FLOAT)
expression2	Specifies the divisor (type same as dividend)

Notes

When computing mod(N,M), the following rules apply:

- If either N or M is the null value, then the result is the null value.
- If M is zero, then an exception condition is raised: data exception — division by zero.
- Otherwise, the result is the unique exact numeric value R with scale 0 (zero) such that all of the following are true:
  - R has the same sign as N.
  - The absolute value of R is less than the absolute value of M.
  - N = M * K + R for some exact numeric value K with scale 0 (zero).

Examples

```
SELECT MOD(9,4);
 mod
-----
   1
(1 row)

SELECT MOD(10,3);
 mod
-----
   1
(1 row)

SELECT MOD(-10,3);
 mod
-----
  -1
(1 row)

SELECT MOD(-10,-3);
 mod
-----
  -1
(1 row)

SELECT MOD(10,-3);
 mod
-----
   1
(1 row)
```

`MOD(<float>, 0)` gives an error:

```
=> SELECT MOD(6.2,0);
ERROR:  numeric division by zero
```

MODULARHASH

Calculates a hash value over its arguments for the purpose of projection segmentation. In all other uses, returns 0.

If you can hash segment your data using a column with a regular pattern, such as a sequential unique identifier, MODULARHASH distributes the data more evenly than HASH, which distributes data using a normal statistical distribution.

Behavior Type

Immutable

Syntax

```
MODULARHASH ( expression [ ,... ] )
```

Parameters

expression	Is a ***column reference*** (see "***Column References***" on page 54) of any data type.

Notes

The MODULARHASH() function takes up to 32 arguments, usually column names, and selects a specific node for each row based on the values of the columns for that row.

Examples

```
CREATE PROJECTION fact_ts_2 (f_price, f_cid, f_tid, f_cost, f_date)
AS (SELECT price, cid, tid, cost, dwdate
    FROM fact)
    SEGMENTED BY MODULARHASH(dwdate)
    ALL NODES OFFSET 2;
```

See Also

HASH (page 308)

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.

Behavior Type

Immutable

Syntax

```
PI()
```

Examples

```
SELECT PI();
        pi
------------------
 3.14159265358979
(1 row)
```

POWER (or POW)

Returns a DOUBLE PRECISION value representing one number raised to the power of another number. You can use either POWER or POW as the function name.

Behavior Type

Immutable

Syntax

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

Parameters

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

```
SELECT POWER(9.0, 3.0);
 power
-------
   729
(1 row)
```

RADIANS

Returns a DOUBLE PRECISION value representing an angle expressed in radians. You can express the input angle in ***degrees*** (page 304), and optionally include minutes and seconds.

Behavior Type

Immutable

Syntax

```
RADIANS (degrees [, minutes, seconds])
```

Parameters

degrees	A unit of angular measurement, representing 1/360 of a full rotation.
minutes	A unit of angular measurement, representing 1/60 of a degree.

seconds	A unit of angular measurement, representing 1/60 of a minute.

Examples

```
SELECT RADIANS(45);
      RADIANS
-------------------
 0.785398163397448
(1 row)


SELECT RADIANS (1,2,3);
      RADIANS
-------------------
 0.018049613347708
(1 row)
```

RANDOM

Returns a uniformly-distributed random number x, where 0 <= x < 1.

Behavior Type

Volatile

Syntax

```
RANDOM()
```

Parameters

RANDOM has no arguments. Its result is a FLOAT8 data type (also called ***DOUBLE PRECISION*** (page 105)).

Notes

Typical pseudo-random generators accept a seed, which is set to generate a reproducible pseudo-random sequence. HP Vertica, however, distributes SQL processing over a cluster of nodes, where each node generates its own independent random sequence.

Results depending on RANDOM are not reproducible because the work might be divided differently across nodes. Therefore, HP Vertica automatically generates truly random seeds for each node each time a request is executed and does not provide a mechanism for forcing a specific seed.

Examples

In the following example, the result is a float, which is >= 0 and < 1.0:

```
SELECT RANDOM();
      random
-------------------
 0.211625560652465
```

```
(1 row)
```

RANDOMINT

Returns a uniformly-distributed integer I, where 0 <= I < N, where N <= MAX_INT8. That is, RANDOMINT(N) returns one of the N integers from 0 through N-1.

Behavior Type

Volatile

Syntax

```
RANDOMINT ( N )
```

Example

In the following example, the result is an INT8, which is >= 0 and < N. In this case, INT8 is randomly chosen from the set {0,1,2,3,4}.

```
SELECT RANDOMINT(5);
randomint
----------
         3
(1 row)
```

ROUND

Rounds a value to a specified number of decimal places, retaining the original scale and precision. Fractions greater than or equal to .5 are rounded up. Fractions less than .5 are rounded down (truncated).

Behavior Type

Immutable

Syntax

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

Parameters

expression	Is an expression of type NUMERIC.
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

NUMERIC ROUND() returns NUMERIC, retaining the original scale and precision:

```
=> SELECT ROUND(3.5);
 ROUND
-------
   4.0
(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(2.0, 1.0 ) FROM dual;
 round
-------
     2
(1 row)
SELECT ROUND(12.345, 2.0 );
 round
-------
 12.35
(1 row)

SELECT ROUND(3.444444444444444);
       ROUND
--------------------
 3.000000000000000
(1 row)
SELECT ROUND(3.14159, 3);
  ROUND
---------
 3.14200
(1 row)
SELECT ROUND(1234567, -3);
  round
---------
 1235000
(1 row)
SELECT ROUND(3.4999, -1);
 ROUND
-------
 .0000
(1 row)
SELECT employee_last_name, ROUND(annual_salary,4) FROM
employee_dimension;
 employee_last_name | ROUND
--------------------+--------
 Li                 |   1880
 Rodriguez          |   1704
 Goldberg           |   2282
 Meyer              |   1628
 Pavlov             |   3168
 McNulty            |   1516
 Dobisz             |   3006
 Pavlov             |   2142
 Goldberg           |   2268
 Pavlov             |   1918
```

```
 Robinson            |    2366
 ...
```

SIGN

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

Behavior Type

Immutable

Syntax

```
SIGN ( expression )
```

Parameters

expression	Is an expression of type DOUBLE PRECISION

Examples

```
SELECT SIGN(-8.4);
 sign
------
   -1
(1 row)
```

SIN

Returns a DOUBLE PRECISION value representing the trigonometric sine of the argument.

Behavior Type

Immutable

Syntax

```
SIN ( expression )
```

Parameters

expression	Is an expression of type DOUBLE PRECISION

Example

```
SELECT SIN(30 * 2 * 3.14159 / 360);
        sin
-------------------
 0.499999616987256
(1 row)
```

SQRT

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

Behavior Type

Immutable

Syntax

```
SQRT ( expression )
```

Parameters

expression	Is an expression of type DOUBLE PRECISION

Examples

```
SELECT SQRT(2);
      sqrt
-----------------
 1.4142135623731
(1 row)
```

TAN

Returns a DOUBLE PRECISION value representing the trigonometric tangent of the argument.

Behavior Type

Immutable

Syntax

```
TAN ( expression )
```

Parameters

expression	Is an expression of type DOUBLE PRECISION

Example

```
SELECT TAN(30);
        tan
-------------------
 -6.40533119664628
(1 row)
```

TRUNC

Returns a value representing the argument fully truncated (toward zero) or truncated to a specific number of decimal places, retaining the original scale and precision.

Behavior Type

Immutable

Syntax

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

Parameters

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

Notes

NUMERIC TRUNC() returns NUMERIC, retaining the original scale and precision:

```
=> SELECT TRUNC(3.5);
 TRUNC
-------
   3.0
(1 row)
```

Examples

```
=>SELECT TRUNC(42.8);
 TRUNC
-------
  42.0
(1 row)
=>SELECT TRUNC(42.4382, 2);
  TRUNC
---------
 42.4300
(1 row)
```

WIDTH_BUCKET

Constructs equiwidth histograms, in which the histogram range is divided into intervals (buckets) of identical sizes. In addition, values below the low bucket return 0, and values above the high bucket return bucket_count +1. Returns an integer value.

Behavior Type

Immutable

Syntax

```
WIDTH_BUCKET ( expression, hist_min, hist_max, bucket_count )
```

Parameters

expression	Is the expression for which the histogram is created. This expression must evaluate to a numeric or datetime value or to a value that can be implicitly converted to a numeric or datetime value. If *expression* evaluates to null, then the *expression* returns null.
hist_min	Is an expression that resolves to the low boundary of bucket 1. Must also evaluate to numeric or datetime values and cannot evaluate to null.
hist_max	Is an expression that resolves to the high boundary of bucket bucket_count. Must also evaluate to a numeric or datetime value and cannot evaluate to null.
bucket_count	Is an expression that resolves to a constant, indicating the number of buckets. This expression always evaluates to a positive INTEGER.

Notes

- WIDTH_BUCKET divides a data set into buckets of equal width. For example, Age = 0-20, 20-40, 40-60, 60-80. This is known as an equiwidth histogram.
- When using WIDTH_BUCKET pay attention to the minimum and maximum boundary values. Each bucket contains values equal to or greater than the base value of that bucket, so that age ranges of 0-20, 20-40, and so on, are actually 0-19.99 and 20-39.999.
- WIDTH_BUCKET accepts the following data types: (FLOAT and/or INT), (TIMESTAMP and/or DATE and/or TIMESTAMPTZ), or (INTERVAL and/or TIME).

Examples

The following example returns five possible values and has three buckets: 0 [Up to 100), 1 [100-300), 2 [300-500), 3 [500-700), and 4 [700 and up):

```
SELECT product_description, product_cost,
WIDTH_BUCKET(product_cost, 100, 700, 3);
```

The following example creates a nine-bucket histogram on the annual_income column for customers in Connecticut who are female doctors. The results return the bucket number to an “Income” column, divided into eleven buckets, including an underflow and an overflow. Note that if customers had an annual incomes greater than the maximum value, they would be assigned to an overflow bucket, 10:

```
SELECT customer_name, annual_income,
WIDTH_BUCKET (annual_income, 100000, 1000000, 9) AS "Income"
FROM public.customer_dimension WHERE customer_state='CT'
```

```
AND title='Dr.' AND customer_gender='Female' AND household_id < '1000'
ORDER BY "Income";
```

In the following result set, the reason there is a bucket 0 is because buckets are numbered from 1 to bucket_count. Anything less than the given value of `hist_min` goes in bucket 0, and anything greater than the given value of `hist_max` goes in the bucket `bucket_count+1`. In this example, bucket 9 is empty, and there is no overflow. The value 12,283 is less than 100,000, so it goes into the underflow bucket.

```
   customer_name    | annual_income | Income
--------------------+---------------+--------
 Joanna A. Nguyen   |         12283 |      0
 Amy I. Nguyen      |        109806 |      1
 Juanita L. Taylor  |        219002 |      2
 Carla E. Brown     |        240872 |      2
 Kim U. Overstreet  |        284011 |      2
 Tiffany N. Reyes   |        323213 |      3
 Rebecca V. Martin  |        324493 |      3
 Betty . Roy        |        476055 |      4
 Midori B. Young    |        462587 |      4
 Martha T. Brown    |        687810 |      6
 Julie D. Miller    |        616509 |      6
 Julie Y. Nielson   |        894910 |      8
 Sarah B. Weaver    |        896260 |      8
 Jessica C. Nielson |        861066 |      8
(14 rows)
```

See Also

NTILE (page 175)

NULL-handling Functions

NULL-handling functions take arguments of any type, and their return type is based on their argument types.

COALESCE

Returns the value of the first non-null expression in the list. If all expressions evaluate to null, then the COALESCE function returns null.

Behavior Type

Immutable

Syntax

```
COALESCE ( expression1, expression2 );
COALESCE ( expression1, expression2, ... expression-n );
```

Parameters

- `COALESCE (expression1, expression2)` is equivalent to the following CASE expression:
  ```
  CASE WHEN expression1 IS NOT NULL THEN expression1 ELSE expression2 END;
  ```
- `COALESCE (expression1, expression2, ... expression-n)`, for n >= 3, is equivalent to the following CASE expression:
  ```
  CASE WHEN expression1 IS NOT NULL THEN expression1
  ELSE COALESCE (expression2, . . . , expression-n) END;
  ```

Notes

COALESCE is an ANSI standard function (SQL-92).

Example

```
SELECT product_description, COALESCE(lowest_competitor_price,
  highest_competitor_price, average_competitor_price) AS price
FROM product_dimension;
```

```
        product_description         | price
------------------------------------+-------
 Brand #54109 kidney beans          |   264
 Brand #53364 veal                  |   139
 Brand #50720 ice cream sandwiches  |   127
 Brand #48820 coffee cake           |   174
 Brand #48151 halibut               |   353
 Brand #47165 canned olives         |   250
 Brand #39509 lamb                  |   306
 Brand #36228 tuna                  |   245
 Brand #34156 blueberry muffins     |   183
 Brand #31207 clams                 |   163
(10 rows)
```

See Also

Case Expressions (page 52)

ISNULL (page 323)

IFNULL

Returns the value of the first non-null expression in the list.

IFNULL is an alias of ***NVL*** (page 327).

Behavior Type

Immutable

Syntax

```
IFNULL ( expression1 , expression2 );
```

Parameters

- If *expression1* is null, then IFNULL returns *expression2.*
- If *expression1* is not null, then IFNULL returns *expression1.*

Notes

- ***COALESCE*** (page 321) is the more standard, more general function.
- IFNULL is equivalent to ISNULL.
- IFNULL is equivalent to COALESCE except that IFNULL is called with only two arguments.
- `ISNULL(a,b)` is different from `x IS NULL`.
- The arguments can have any data type supported by HP Vertica.
- Implementation is equivalent to the CASE expression. For example:
  ```
  CASE WHEN expression1 IS NULL THEN expression2 ELSE expression1 END;
  ```
- The following statement returns the value 140:
  ```
  SELECT IFNULL(NULL, 140) FROM employee_dimension;
  ```
- The following statement returns the value 60:
  ```
  SELECT IFNULL(60, 90) FROM employee_dimension;
  ```

Examples

```
=> SELECT IFNULL (SCORE, 0.0) FROM TESTING;
 IFNULL
--------
  100.0
   87.0
     .0
     .0
     .0
(5 rows)
```

See Also

Case Expressions (page 52)

COALESCE (page 321)

NVL (page 327)

ISNULL (page 323)

ISNULL

Returns the value of the first non-null expression in the list.

ISNULL is an alias of ***NVL*** (page 327).

Behavior Type

Immutable

Syntax

```
ISNULL ( expression1 , expression2 );
```

Parameters

- If *expression1* is null, then ISNULL returns *expression2.*
- If *expression1* is not null, then ISNULL returns *expression1.*

Notes

- ***COALESCE*** (page 321) is the more standard, more general function.
- ISNULL is equivalent to COALESCE except that ISNULL is called with only two arguments.
- `ISNULL(a,b)` is different from `x IS NULL`.
- The arguments can have any data type supported by HP Vertica.
- Implementation is equivalent to the CASE expression. For example:
  `CASE WHEN expression1 IS NULL THEN expression2 ELSE expression1 END;`
- The following statement returns the value 140:
  `SELECT ISNULL(NULL, 140) FROM employee_dimension;`
- The following statement returns the value 60:
  `SELECT ISNULL(60, 90) FROM employee_dimension;`

Examples

```
SELECT product_description, product_price, ISNULL(product_cost, 0.0) AS cost
FROM product_dimension;
```

```
      product_description       | product_price | cost
--------------------------------+---------------+------
 Brand #59957 wheat bread       |           405 |  207
 Brand #59052 blueberry muffins |           211 |  140
 Brand #59004 english muffins   |           399 |  240
 Brand #53222 wheat bread       |           323 |   94
 Brand #52951 croissants        |           367 |  121
 Brand #50658 croissants        |           100 |   94
 Brand #49398 white bread       |           318 |   25
 Brand #46099 wheat bread       |           242 |    3
 Brand #45283 wheat bread       |           111 |  105
 Brand #43503 jelly donuts      |           259 |   19
(10 rows)
```

See Also

Case Expressions (page 52)

COALESCE (page 321)

NVL (page 327)

NULLIF

Compares two expressions. If the expressions are not equal, the function returns the first expression (expression1). If the expressions are equal, the function returns null.

Behavior Type

Immutable

Syntax

```
NULLIF( expression1, expression2 )
```

Parameters

expression1	Is a value of any data type.
expression2	Must have the same data type as *expr1* or a type that can be implicitly cast to match *expression1*. The result has the same type as *expression1*.

Examples

The following series of statements illustrates one simple use of the NULLIF function.

Creates a single-column table `t and insert some values`:

```
CREATE TABLE t (x TIMESTAMPTZ);
INSERT INTO t VALUES('2009-09-04 09:14:00-04');
INSERT INTO t VALUES('2010-09-04 09:14:00-04');
```

Issue a select statement:

```
SELECT x, NULLIF(x, '2009-09-04 09:14:00 EDT') FROM t;
           x            |         nullif
------------------------+------------------------
 2009-09-04 09:14:00-04 |
 2010-09-04 09:14:00-04 | 2010-09-04 09:14:00-04

SELECT NULLIF(1, 2);
 NULLIF
--------
      1
(1 row)

SELECT NULLIF(1, 1);
 NULLIF
--------

(1 row)

SELECT NULLIF(20.45, 50.80);
 NULLIF
--------
  20.45
```

```
(1 row)
```

NULLIFZERO

Evaluates to NULL if the value in the column is 0.

Syntax

NULLIFZERO(*expression*)

Parameters

expression	(INTEGER, DOUBLE PRECISION, INTERVAL, or NUMERIC) Is the string to evaluate for 0 values.

Example

The TESTING table below shows the test scores for 5 students. Note that test scores are missing for S. Robinson and K. Johnson (NULL values appear in the Score column.)

```
=> select * from TESTING;
    Name     | Score
-------------+-------
 J. Doe      |   100
 R. Smith    |    87
 L. White    |     0
 S. Robinson |
 K. Johnson  |
(5 rows)
```

The SELECT statement below specifies that HP Vertica should return any 0 values in the Score column as Null. In the results, you can see that HP Vertica returns L. White's 0 score as Null.

```
=> SELECT Name, NULLIFZERO(Score) FROM TESTING;

    Name     | NULLIFZERO
-------------+------------
 J. Doe      |        100
 R. Smith    |         87
 L. White    |
 S. Robinson |
 K. Johnson  |
(5 rows)
```

NVL

Returns the value of the first non-null expression in the list.

Behavior Type

Immutable

Syntax

```
NVL ( expression1 , expression2 );
```

Parameters

- If *expression1* is null, then NVL returns *expression2.*
- If *expression1* is not null, then NVL returns *expression1.*

Notes

- ***COALESCE*** (page 321) is the more standard, more general function.
- NVL is equivalent to COALESCE except that NVL is called with only two arguments.
- The arguments can have any data type supported by HP Vertica.
- Implementation is equivalent to the CASE expression:
  `CASE WHEN expression1 IS NULL THEN expression2 ELSE expression1 END;`

Examples

expression1 is not null, so NVL returns expression1:

```
SELECT NVL('fast', 'database');
 nvl
------
 fast
(1 row)
```

expression1 is null, so NVL returns expression2:

```
SELECT NVL(null, 'database');
   nvl
----------
 database
(1 row)
```

expression2 is null, so NVL returns expression1:

```
SELECT NVL('fast', null);
 nvl
------
 fast
(1 row)
```

In the following example, expression1 (title) contains nulls, so NVL returns expression2 and substitutes 'Withheld' for the unknown values:

```
SELECT customer_name,
       NVL(title, 'Withheld') as title
```

```
FROM customer_dimension
ORDER BY title;
     customer_name      | title
------------------------+-------
 Alexander I. Lang      | Dr.
 Steve S. Harris        | Dr.
 Daniel R. King         | Dr.
 Luigi I. Sanchez       | Dr.
 Duncan U. Carcetti     | Dr.
 Meghan K. Li           | Dr.
 Laura B. Perkins       | Dr.
 Samantha V. Robinson   | Dr.
 Joseph P. Wilson       | Mr.
 Kevin R. Miller        | Mr.
 Lauren D. Nguyen       | Mrs.
 Emily E. Goldberg      | Mrs.
 Darlene K. Harris      | Ms.
 Meghan J. Farmer       | Ms.
 Bettercare             | Withheld
 Ameristar              | Withheld
 Initech                | Withheld
(17 rows)
```

See Also

Case Expressions (page 52)

COALESCE (page 321)

ISNULL (page 323)

NVL2 (page 328)

NVL2

Takes three arguments. If the first argument is not NULL, it returns the second argument, otherwise it returns the third argument. The data types of the second and third arguments are implicitly cast to a common type if they don't agree, similar to ***COALESCE*** (page 321).

Behavior Type

Immutable

Syntax

```
NVL2 ( expression1 , expression2 , expression3 );
```

Parameters

- If *expression1* is not null, then NVL2 returns *expression2.*
- If *expression1* is null, then NVL2 returns *expression3.*

Notes

Arguments two and three can have any data type supported by HP Vertica.

Implementation is equivalent to the CASE expression:

```
CASE WHEN expression1 IS NOT NULL THEN expression2 ELSE expression3 END;
```

Examples

In this example, expression1 is not null, so NVL2 returns expression2:

```
SELECT NVL2('very', 'fast', 'database');
 nvl2
------
 fast
(1 row)
```

In this example, expression1 is null, so NVL2 returns expression3:

```
SELECT NVL2(null, 'fast', 'database');
   nvl2
----------
 database
(1 row)
```

In the following example, expression1 (title) contains nulls, so NVL2 returns expression3 ('Withheld') and also substitutes the non-null values with the expression 'Known':

```
SELECT customer_name,
       NVL2(title, 'Known', 'Withheld') as title
FROM customer_dimension
ORDER BY title;
     customer_name      | title
------------------------+-------
 Alexander I. Lang      | Known
 Steve S. Harris        | Known
 Daniel R. King         | Known
 Luigi I. Sanchez       | Known
 Duncan U. Carcetti     | Known
 Meghan K. Li           | Known
 Laura B. Perkins       | Known
 Samantha V. Robinson   | Known
 Joseph P. Wilson       | Known
 Kevin R. Miller        | Known
 Lauren D. Nguyen       | Known
 Emily E. Goldberg      | Known
 Darlene K. Harris      | Known
 Meghan J. Farmer       | Known
 Bettercare             | Withheld
 Ameristar              | Withheld
 Initech                | Withheld
(17 rows)
```

See Also

Case Expressions (page 52)

COALESCE (page 321)

NVL (page 321)

ZEROIFNULL

Evaluates to 0 if the column is NULL.

Syntax

```
ZEROIFNULL(expression)
```

Parameters

expression	(INTEGER, DOUBLE PRECISION, INTERVAL, or NUMERIC) Is the string to evaluate for NULL values.

Example

The TESTING table below shows the test scores for 5 students. Note that L. White's score is 0, and that scores are missing for S. Robinson and K. Johnson.

```
=> select * from TESTING;
    Name     | Score
-------------+-------
 J. Doe      |   100
 R. Smith    |    87
 L. White    |     0
 S. Robinson |
 K. Johnson  |
(5 rows)
```

The SELECT statement below specifies that HP Vertica should return any Null values in the Score column as 0s. In the results, you can see that HP Vertica returns a 0 score for S. Robinson and K. Johnson.

```
=> SELECT Name, ZEROIFNULL (Score) FROM TESTING;
    Name     | ZEROIFNULL
-------------+------------
 J. Doe      |        100
 R. Smith    |         87
 L. White    |          0
 S. Robinson |          0
 K. Johnson  |          0
(5 rows)
```

Pattern Matching Functions

Used with the ***MATCH clause*** (page 887), the HP Vertica pattern matching functions return additional data about the patterns found/output. For example, you can use these functions to return values representing the name of the event or pattern that matched the input row, the sequential number of the match, or a partition-wide unique identifier for the instance of the pattern that matched.

Pattern matching is particularly useful for clickstream analysis where you might want to identify users' actions based on their Web browsing behavior (page clicks). A typical online clickstream funnel is:

Company home page -> product home page -> search -> results -> purchase online

Using the above clickstream funnel, you can search for a match on the user's sequence of web clicks and identify that the user:

- landed on the company home page
- navigated to the product page
- ran a search
- clicked a link from the search results
- made a purchase

For examples that use this clickstream model, see Event Series Pattern Matching in the Programmer's Guide.

See Also

MATCH Clause (page 887)

Event Series Pattern Matching in the Programmer's Guide

EVENT_NAME

Returns a VARCHAR value representing the name of the event that matched the row.

Syntax

```
EVENT_NAME()
```

Notes

Pattern matching functions must be used in ***MATCH clause*** (page 887) syntax; for example, if you call EVENT_NAME() on its own, HP Vertica returns the following error message:

```
=> SELECT event_name();
ERROR:  query with pattern matching function event_name must include a MATCH clause
```

Example

Note: This example uses the schema defined in Event Series Pattern Matching in the Programmer's Guide. For a more detailed example, see that topic.

The following statement analyzes users' browsing history on website2.com and identifies patterns where the user landed on website2.com from another Web site (Entry) and browsed to any number of other pages (Onsite) before making a purchase (Purchase). The query also outputs the values for EVENT_NAME(), which is the name of the event that matched the row.

```
SELECT uid,
       sid,
       ts,
       refurl,
       pageurl,
       action,
       event_name()
FROM clickstream_log
MATCH
  (PARTITION BY uid, sid ORDER BY ts
   DEFINE
     Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',
     Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',
     Purchase AS PageURL ILIKE     '%website2.com%' AND Action = 'P'
   PATTERN
     P AS (Entry Onsite* Purchase)
   RESULTS ALL ROWS);

 uid | sid |    ts    |        refurl        |       pageurl        | action | event_name
-----+-----+----------+----------------------+----------------------+--------+------------
   1 | 100 | 12:00:00 | website1.com         | website2.com/home    | V      | Entry
   1 | 100 | 12:01:00 | website2.com/home    | website2.com/floby   | V      | Onsite
   1 | 100 | 12:02:00 | website2.com/floby   | website2.com/shamwow | V      | Onsite
   1 | 100 | 12:03:00 | website2.com/shamwow | website2.com/buy     | P      | Purchase
   2 | 100 | 12:10:00 | website1.com         | website2.com/home    | V      | Entry
   2 | 100 | 12:11:00 | website2.com/home    | website2.com/forks   | V      | Onsite
   2 | 100 | 12:13:00 | website2.com/forks   | website2.com/buy     | P      | Purchase
(7 rows)
```

See Also

MATCH Clause (page 887)

MATCH_ID (page 332)

PATTERN_ID (page 334)

Event Series Pattern Matching in the Programmer's Guide

MATCH_ID

Returns a successful pattern match as an INTEGER value. The returned value is the ordinal position of a match within a partition.

Syntax

```
MATCH_ID()
```

Notes

Pattern matching functions must be used in ***MATCH clause*** (page 887) syntax; for example, if you call MATCH_ID() on its own, HP Vertica returns the following error message:

```
=> SELECT match_id();
ERROR:  query with pattern matching function match_id must include a MATCH clause
```

Example

Note: This example uses the schema defined in Event Series Pattern Matching in the Programmer's Guide. For a more detailed example, see that topic.

The following statement analyzes users' browsing history on a site called `website2.com` and identifies patterns where the user reached `website2.com` from another Web site (`Entry` in the `MATCH` clause) and browsed to any number of other pages (`Onsite`) before making a purchase (Purchase). The query also outputs values for the MATCH_ID(), which represents a sequential number of the match.

```
SELECT uid,
       sid,
       ts,
       refurl,
       pageurl,
       action,
       match_id()
FROM clickstream_log
MATCH
  (PARTITION BY uid, sid ORDER BY ts
   DEFINE
     Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',
     Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',
     Purchase AS PageURL ILIKE     '%website2.com%' AND Action = 'P'
   PATTERN
     P AS (Entry Onsite* Purchase)
   RESULTS ALL ROWS);

 uid | sid |    ts    |        refurl        |       pageurl        | action | match_id
-----+-----+----------+----------------------+----------------------+--------+----------
   2 | 100 | 12:10:00 | website1.com         | website2.com/home    | V      |        1
   2 | 100 | 12:11:00 | website2.com/home    | website2.com/forks   | V      |        2
   2 | 100 | 12:13:00 | website2.com/forks   | website2.com/buy     | P      |        3
   1 | 100 | 12:00:00 | website1.com         | website2.com/home    | V      |        1
   1 | 100 | 12:01:00 | website2.com/home    | website2.com/floby   | V      |        2
   1 | 100 | 12:02:00 | website2.com/floby   | website2.com/shamwow | V      |        3
   1 | 100 | 12:03:00 | website2.com/shamwow | website2.com/buy     | P      |        4
(7 rows)
```

See Also

MATCH Clause (page 887)

EVENT_NAME (page 331)

PATTERN_ID (page 334)

Event Series Pattern Matching in the Programmer's Guide

PATTERN_ID

Returns an integer value that is a partition-wide unique identifier for the instance of the pattern that matched.

Syntax

```
PATTERN_ID()
```

Notes

Pattern matching functions must be used in ***MATCH clause*** (page 887) syntax; for example, if call PATTERN_ID() on its own, HP Vertica returns the following error message:

```
=> SELECT pattern_id();
ERROR:  query with pattern matching function pattern_id must include a MATCH clause
```

Example

Note: This example uses the schema defined in Event Series Pattern Matching in the Programmer's Guide. For a more detailed example, see that topic.

The following statement analyzes users' browsing history on website2.com and identifies patterns where the user landed on website2.com from another Web site (Entry) and browsed to any number of other pages (Onsite) before making a purchase (Purchase). The query also outputs values for PATTERN_ID(), which represents the partition-wide identifier for the instance of the pattern that matched.

```
SELECT uid,
       sid,
       ts,
       refurl,
       pageurl,
       action,
       pattern_id()
FROM clickstream_log
MATCH
  (PARTITION BY uid, sid ORDER BY ts
   DEFINE
     Entry    AS RefURL  NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',
     Onsite   AS PageURL ILIKE     '%website2.com%' AND Action='V',
     Purchase AS PageURL ILIKE     '%website2.com%' AND Action = 'P'
   PATTERN
     P AS (Entry Onsite* Purchase)
   RESULTS ALL ROWS);

 uid | sid |    ts    |        refurl        |       pageurl        | action | pattern_id
-----+-----+----------+----------------------+----------------------+--------+------------
   2 | 100 | 12:10:00 | website1.com         | website2.com/home    | V      |          1
   2 | 100 | 12:11:00 | website2.com/home    | website2.com/forks   | V      |          1
   2 | 100 | 12:13:00 | website2.com/forks   | website2.com/buy     | P      |          1
   1 | 100 | 12:00:00 | website1.com         | website2.com/home    | V      |          1
   1 | 100 | 12:01:00 | website2.com/home    | website2.com/floby   | V      |          1
   1 | 100 | 12:02:00 | website2.com/floby   | website2.com/shamwow | V      |          1
   1 | 100 | 12:03:00 | website2.com/shamwow | website2.com/buy     | P      |          1
(7 rows)
```

See Also

MATCH Clause (page 887)

EVENT_NAME (page 331)

MATCH_ID (page 332)

Event Series Pattern Matching in the Programmer's Guide

Regular Expression Functions

A regular expression lets you perform pattern matching on strings of characters. The regular expression syntax allows you to very precisely define the pattern used to match strings, giving you much greater control than the wildcard matching used in the ***LIKE*** (page 66) predicate. HP Vertica's regular expression functions let you perform tasks such as determining if a string value matches a pattern, extracting a portion of a string that matches a pattern, or counting the number of times a string matches a pattern.

HP Vertica uses the ***Perl Compatible Regular Expression library http://www.pcre.org/*** (PCRE) to evaluate regular expressions. As its name implies, PCRE's regular expression syntax is compatible with the syntax used by the Perl 5 programming language. You can read ***PCRE's documentation on its regular expression syntax http://vcs.pcre.org/viewvc/code/trunk/doc/html/pcrepattern.html?view=co***. However, you might find the ***Perl Regular Expressions Documentation*** (***http://perldoc.perl.org/perlre.html***) to be a better introduction, especially if you are unfamiliar with regular expressions.

Note: The regular expression functions only operate on valid UTF-8 strings. If you attempt to use a regular expression function on a string that is not valid UTF-8, then the query fails with an error. To prevent an error from occurring, you can use the ***ISUTF8*** (page 335) function as a clause in the statement to ensure the strings you want to pass to the regular expression functions are actually valid UTF-8 strings, or you can use the 'b' argument to treat the strings as binary octets rather than UTF-8 encoded strings.

ISUTF8

Tests whether a string is a valid UTF-8 string. Returns true if the string conforms to UTF-8 standards, and false otherwise. This function is useful to test strings for UTF-8 compliance before passing them to one of the regular expression functions, such as ***REGEXP_LIKE*** (page 341), which expect UTF-8 characters by default.

ISUTF8 checks for invalid UTF8 byte sequences, according to UTF-8 rules:

- invalid bytes
- an unexpected continuation byte
- a start byte not followed by enough continuation bytes

- an Overload Encoding

The presence of an invalid UTF8 byte sequence results in a return value of false.

Syntax

```
ISUTF8( string );
```

Parameters

string	The string to test for UTF-8 compliance.

Examples

```
=> SELECT ISUTF8(E'\xC2\xBF'); -- UTF-8 INVERTED QUESTION MARK
 ISUTF8
--------
 t
(1 row)

=> SELECT ISUTF8(E'\xC2\xC0'); -- UNDEFINED UTF-8 CHARACTER
 ISUTF8
--------
 f
(1 row)
```

REGEXP_COUNT

Returns the number times a regular expression matches a string.

Syntax

```
REGEXP_COUNT( string, pattern [, position [, regexp_modifier ] ] )
```

Parameters

string	The string to be searched for matches.
pattern	The regular expression to search for within the string. The syntax of the regular expression is compatible with the Perl 5 regular expression syntax. See the ***Perl Regular Expressions Documentation*** (***http://perldoc.perl.org/perlre.html***) for details.
position	The number of characters from the start of the string where the function should start searching for matches. The default value, 1, means to start searching for a match at the first (leftmost) character. Setting this parameter to a value greater than 1 starts searching for a match to the pattern that many characters into the string.

regexp_modifier	A string containing one or more single-character flags that change how the regular expression is matched against the string: `b` Treat strings as binary octets rather than UTF-8 characters. `c` Forces the match to be case sensitive (the default). `i` Forces the match to be case insensitive. `m` Treats the string being matched as multiple lines. With this modifier, the start of line (`^`) and end of line (`$`) regular expression operators match line breaks (`\n`) within the string. Ordinarily, these operators only match the start and end of the string. `n` Allows the single character regular expression operator (`.`) to match a newline (`\n`). Normally, the `.` operator will match any character except a newline. `x` Allows you to document your regular expressions. It causes all unescaped space characters and comments in the regular expression to be ignored. Comments start with a hash character (#) and end with a newline. All spaces in the regular expression that you want to be matched in strings must be escaped with a backslash (\) character.

Notes

This function operates on UTF-8 strings using the default locale, even if the locale has been set to something else.

If you are porting a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while HP Vertica does not.

Examples

Count the number of occurrences of the substring "an" in the string "A man, a plan, a canal, Panama."

```
=> SELECT REGEXP_COUNT('a man, a plan, a canal: Panama', 'an');
 REGEXP_COUNT
--------------
            4
(1 row)
```

Find the number of occurrences of the substring "an" in the string "a man, a plan, a canal: Panama" starting with the fifth character.

```
=> SELECT REGEXP_COUNT('a man, a plan, a canal: Panama', 'an',5);
 REGEXP_COUNT
--------------
            3
(1 row)
```

Find the number of occurrences of a substring containing a lower-case character followed by "an." In the first example, the query does not have a modifier. In the second example, the "i" query modifier is used to force the regular expression to ignore case.

```
=> SELECT REGEXP_COUNT('a man, a plan, a canal: Panama', '[a-z]an');
 REGEXP_COUNT
--------------
            3
(1 row)

=> SELECT REGEXP_COUNT('a man, a plan, a canal: Panama', '[a-z]an', 1, 'i');
 REGEXP_COUNT
--------------
            4
```

REGEXP_INSTR

Returns the starting or ending position in a string where a regular expression matches. This function returns 0 if no match for the regular expression is found in the string.

Syntax

```
REGEXP_INSTR( string, pattern [, position [, occurrence
... [, return_position [, regexp_modifier ]
... [, captured_subexp ] ] ] ] )
```

Parameters

string	The string to search for the pattern.
pattern	The regular expression to search for within the string. The syntax of the regular expression is compatible with the Perl 5 regular expression syntax. See the ***Perl Regular Expressions Documentation*** (***http://perldoc.perl.org/perlre.html***) for details.
position	The number of characters from the start of the string where the function should start searching for matches. The default value, 1, means to start searching for a match at the first (leftmost) character. Setting this parameter to a value greater than 1 starts searching for a match to the pattern that many characters into the string.

occurrence	Controls which occurrence of a match between the string and the pattern is returned. With the default value (1), the function returns the position of the first substring that matches the pattern. You can use this parameter to find the position of additional matches between the string and the pattern. For example, set this parameter to 3 to find the position of the third substring that matched the pattern.
return_position	Sets the position within the string that is returned. When set to the default value (0), this function returns the position in the string of the first character of the substring that matched the pattern. If you set this value to 1, the function returns the position of the first character after the end of the matching substring.
regexp_modifier	A string containing one or more single-character flags that change how the regular expression is matched against the string: `b` Treat strings as binary octets rather than UTF-8 characters. `c` Forces the match to be case sensitive (the default). `i` Forces the match to be case insensitive. `m` Treats the string being matched as multiple lines. With this modifier, the start of line (`^`) and end of line (`$`) regular expression operators match line breaks (`\n`) within the string. Ordinarily, these operators only match the start and end of the string. `n` Allows the single character regular expression operator (`.`) to match a newline (`\n`). Normally, the `.` operator will match any character except a newline. `x` Allows you to document your regular expressions. It causes all unescaped space characters and comments in the regular expression to be ignored. Comments start with a hash character (#) and end with a newline. All spaces in the regular expression that you want to be matched in strings must be escaped with a backslash (\) character.

captured_subexp	The captured subexpression whose position should be returned. If omitted or set to 0, the function returns the position of the first character in the entire string that matched the regular expression. If set to 1 through 9, the function returns the subexpression captured by the corresponding set of parentheses in the regular expression. For example, setting this value to 3 returns the substring captured by the third set of parentheses in the regular expression. **Note:** The subexpressions are numbered left to right, based on the appearance of opening parenthesis, so nested regular expressions . For example, in the regular expression `\s*(\w+\s+(\w+))`, subexpression 1 is the one that captures everything but any leading whitespaces.

Notes

This function operates on UTF-8 strings using the default locale, even if the locale has been set to something else.

If you are porting a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while HP Vertica does not.

Examples

Find the first occurrence of a sequence of letters starting with the letter e and ending with the letter y in the phrase "easy come, easy go."

```
=> SELECT REGEXP_INSTR('easy come, easy go','e\w*y');
 REGEXP_INSTR
--------------
            1
(1 row)
```

Find the first occurrence of a sequence of letters starting with the letter e and ending with the letter y starting at the second character in the string "easy come, easy go."

```
=> SELECT REGEXP_INSTR('easy come, easy go','e\w*y',2);
 REGEXP_INSTR
--------------
           12
(1 row)
```

Find the second sequence of letters starting with the letter e and ending with the letter y in the string "easy come, easy go" starting at the first character.

```
=> SELECT REGEXP_INSTR('easy come, easy go','e\w*y',1,2);
 REGEXP_INSTR
--------------
           12
(1 row)
```

Find the position of the first character after the first whitespace in the string "easy come, easy go."

```
=> SELECT REGEXP_INSTR('easy come, easy go','\s',1,1,1);
 REGEXP_INSTR
--------------
            6
(1 row)
```

Find the position of the start of the third word in a string by capturing each word as a subexpression, and returning the third subexpression's start position.

```
=> SELECT REGEXP_INSTR('one two three','(\w+)\s+(\w+)\s+(\w+)', 1,1,0,'',3);
 REGEXP_INSTR
--------------
            9
(1 row)
```

REGEXP_LIKE

Returns true if the string matches the regular expression. This function is similar to the ***LIKE-predicate*** (page 66), except that it uses regular expressions rather than simple wildcard character matching.

Syntax

```
REGEXP_LIKE( string, pattern [, modifiers ] )
```

Parameters

string	The string to match against the regular expression.
pattern	A string containing the regular expression to match against the string. The syntax of the regular expression is compatible with the Perl 5 regular expression syntax. See the ***Perl Regular Expressions Documentation*** (***http://perldoc.perl.org/perlre.html***) for details.

modifiers	A string containing one or more single-character flags that change how the regular expression is matched against the string: `b` Treat strings as binary octets rather than UTF-8 characters. `c` Forces the match to be case sensitive (the default). `i` Forces the match to be case insensitive. `m` Treats the string being matched as multiple lines. With this modifier, the start of line (`^`) and end of line (`$`) regular expression operators match line breaks (`\n`) within the string. Ordinarily, these operators only match the start and end of the string. `n` Allows the single character regular expression operator (`.`) to match a newline (`\n`). Normally, the `.` operator will match any character except a newline. `x` Allows you to document your regular expressions. It causes all unescaped space characters and comments in the regular expression to be ignored. Comments start with a hash character (#) and end with a newline. All spaces in the regular expression that you want to be matched in strings must be escaped with a backslash (\) character.

Notes

This function operates on UTF-8 strings using the default locale, even if the locale has been set to something else.

If you are porting a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while HP Vertica does not.

Examples

This example creates a table containing several strings to demonstrate regular expressions.

```
=> create table t (v varchar);
CREATE TABLE
=> create projection t1 as select * from t;
CREATE PROJECTION
=> COPY t FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> aaa
>> Aaa
```

```
>> abc
>> abc1
>> 123
>> \.
=> SELECT * FROM t;
   v
-------
 aaa
 Aaa
 abc
 abc1
 123
(5 rows)
```

Select all records in the table that contain the letter "a."

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'a');
  v
------
 Aaa
 aaa
 abc
 abc1
(4 rows)
```

Select all of the rows in the table that start with the letter "a."

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'^a');
  v
------
 aaa
 abc
 abc1
(3 rows)
```

Select all rows that contain the substring "aa."

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'aa');
  v
-----
 Aaa
 aaa
(2 rows)
```

Select all rows that contain a digit.

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'\d');
  v
------
 123
 abc1
(2 rows)
```

Select all rows that contain the substring "aaa."

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'aaa');
  v
-----
 aaa
(1 row)
```

Select all rows that contain the substring "aaa" using case insensitive matching.

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'aaa', 'i');
  v
-----
 Aaa
 aaa
(2 rows)
```

Select rows that contain the substring "a b c."

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'a b c');
 v
---
(0 rows)
```

Select rows that contain the substring "a b c" ignoring space within the regular expression.

```
=> SELECT v FROM t WHERE REGEXP_LIKE(v,'a b c','x');
  v
------
 abc
 abc1
(2 rows)
```

Add multi-line rows to demonstrate using the "m" modifier.

```
=> COPY t FROM stdin RECORD TERMINATOR '!';
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Record 1 line 1
>> Record 1 line 2
>> Record 1 line 3!
>> Record 2 line 1
>> Record 2 line 2
>> Record 2 line 3!
>> \.
```

Select rows that start with the substring "Record" and end with the substring "line 2."

```
=> SELECT v from t WHERE REGEXP_LIKE(v,'^Record.*line 2$');
 v
---
(0 rows)
```

Select rows that start with the substring "Record" and end with the substring "line 2," treating multiple lines as separate strings.

```
=> SELECT v from t WHERE REGEXP_LIKE(v,'^Record.*line 2$','m');
                          v
-----------------------------------------------

Record 2 line 1
Record 2 line 2
Record 2 line 3
 Record 1 line 1
Record 1 line 2
Record 1 line 3
(2 rows)
```

REGEXP_REPLACE

Replace all occurrences of a substring that match a regular expression with another substring. It is similar to the ***REPLACE*** (page 393) function, except it uses a regular expression to select the substring to be replaced.

Syntax

```
REGEXP_REPLACE( string, target [, replacement [, position [, occurrence
... [, regexp_modifiers ] ] ] ] )
```

Parameters

string	The string whose to be searched and replaced.
target	The regular expression to search for within the string. The syntax of the regular expression is compatible with the Perl 5 regular expression syntax. See the ***Perl Regular Expressions Documentation*** (***http://perldoc.perl.org/perlre.html***) for details.
replacement	The string to replace matched substrings. If not supplied, the matched substrings are deleted. This string can contain baccalaureates for substrings captured by the regular expression. The first captured substring is inserted into the replacement string using `\1`, the second `\2`, and so on.
position	The number of characters from the start of the string where the function should start searching for matches. The default value, 1, means to start searching for a match at the first (leftmost) character. Setting this parameter to a value greater than 1 starts searching for a match to the pattern that many characters into the string.

occurrence	Controls which occurrence of a match between the string and the pattern is replaced. With the default value (0), the function replaces all matching substrings with the replacement string. For any value above zero, the function replaces just a single occurrence. For example, set this parameter to 3 to replace the third substring that matched the pattern.
regexp_modifier	A string containing one or more single-character flags that change how the regular expression is matched against the string: `b` Treat strings as binary octets rather than UTF-8 characters. `c` Forces the match to be case sensitive (the default). `i` Forces the match to be case insensitive. `m` Treats the string being matched as multiple lines. With this modifier, the start of line (`^`) and end of line (`$`) regular expression operators match line breaks (`\n`) within the string. Ordinarily, these operators only match the start and end of the string. `n` Allows the single character regular expression operator (`.`) to match a newline (`\n`). Normally, the `.` operator will match any character except a newline. `x` Allows you to document your regular expressions. It causes all unescaped space characters and comments in the regular expression to be ignored. Comments start with a hash character (#) and end with a newline. All spaces in the regular expression that you want to be matched in strings must be escaped with a backslash (\) character.

Notes

This function operates on UTF-8 strings using the default locale, even if the locale has been set to something else.

If you are porting a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while HP Vertica does not.

Another key difference between Oracle and HP Vertica is that HP Vertica can handle an unlimited number of captured subexpressions where Oracle is limited to nine. In HP Vertica, you are able to use `\10` in the replacement pattern to access the substring captured by the tenth set of parentheses in the regular expression. In Oracle, `\10` is treated as the substring captured by the first set of parentheses followed by a zero. To force the same behavior in HP Vertica, use the `\g` backreference with the number of the captured subexpression enclosed in curly braces. For example, `\g{1}0` is the substring captured by the first set of parentheses followed by a zero. You can also name your captured subexpressions, to make your regular expressions less ambiguous. See the ***PCRE http://vcs.pcre.org/viewvc/code/trunk/doc/html/pcrepattern.html?view=co*** documentation for details.

Examples

Find groups of "word characters" (letters, numbers and underscore) ending with "thy" in the string "healthy, wealthy, and wise" and replace them with nothing.

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','\w+thy');
 REGEXP_REPLACE
----------------
 , , and wise
(1 row)
```

Find groups of word characters ending with "thy" and replace with the string "something."

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','\w+thy', 'something');
         REGEXP_REPLACE
--------------------------------
 something, something, and wise
(1 row)
```

Find groups of word characters ending with "thy" and replace with the string "something" starting at the third character in the string.

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','\w+thy', 'something', 3);
          REGEXP_REPLACE
---------------------------------
 hesomething, something, and wise
(1 row)
```

Replace the second group of word characters ending with "thy" with the string "something."

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','\w+thy', 'something', 1,
2);
        REGEXP_REPLACE
------------------------------
 healthy, something, and wise
(1 row)
```

Find groups of word characters ending with "thy" capturing the letters before the "thy", and replace with the captured letters plus the letters "ish."

```
=> SELECT REGEXP_REPLACE('healthy, wealthy, and wise','(\w+)thy', '\1ish');
```

```
       REGEXP_REPLACE
---------------------------
 healish, wealish, and wise
(1 row)
```

Create a table to demonstrate replacing strings in a query.

```
=> CREATE TABLE customers (name varchar(50), phone varchar(11));
CREATE TABLE
=> CREATE PROJECTION customers1 AS SELECT * FROM customers;
CREATE PROJECTION
=> COPY customers FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> Able, Adam|17815551234
>> Baker,Bob|18005551111
>> Chu,Cindy|16175559876
>> Dodd,Dinara|15083452121
>> \.
```

Query the customers, using REGEXP_REPLACE to format the phone numbers.

```
=> SELECT name, REGEXP_REPLACE(phone, '(\d)(\d{3})(\d{3})(\d{4})', '\1-(\2)
\3-\4') as phone FROM customers;
    name     |      phone
-------------+------------------
 Able, Adam  | 1-(781) 555-1234
 Baker,Bob   | 1-(800) 555-1111
 Chu,Cindy   | 1-(617) 555-9876
 Dodd,Dinara | 1-(508) 345-2121
(4 rows)
```

REGEXP_SUBSTR

Returns the substring that matches a regular expression within a string. If no matches are found, this function returns NULL. This is different than an empty string, which can be returned by this function if the regular expression matches a zero-length string.

Syntax

```
REGEXP_SUBSTR( string, pattern [, position [,  occurrence
... [, regexp_modifier ] [, captured_subexp ] ] ] )
```

Parameters

string	The string to search for the pattern.

pattern	The regular expression to find the substring to be extracted. The syntax of the regular expression is compatible with the Perl 5 regular expression syntax. See the ***Perl Regular Expressions Documentation*** (***http://perldoc.perl.org/perlre.html***) for details.
position	The character in the string where the search for a match should start. The default value, 1, starts the search at the beginning of the string. If you supply a value larger than 1 for this parameter, the function will start searching that many characters into the string.
occurrence	Controls which matching substring is returned by the function. When given the default value (1), the function will return the first matching substring it finds in the string. By setting this value to a number greater than 1, this function will return subsequent matching substrings. For example, setting this parameter to 3 will return the third substring that matches the regular expression within the string.
regexp_modifier	A string containing one or more single-character flags that change how the regular expression is matched against the string: `b` Treat strings as binary octets rather than UTF-8 characters. `c` Forces the match to be case sensitive (the default). `i` Forces the match to be case insensitive. `m` Treats the string being matched as multiple lines. With this modifier, the start of line (`^`) and end of line (`$`) regular expression operators match line breaks (`\n`) within the string. Ordinarily, these operators only match the start and end of the string. `n` Allows the single character regular expression operator (`.`) to match a newline (`\n`). Normally, the `.` operator will match any character except a newline. `x` Allows you to document your regular expressions. It causes all unescaped space characters and comments in the regular expression to be ignored. Comments start with a hash character (#) and end with a newline. All spaces in the regular expression that you want to be matched in strings must be escaped with a backslash (\) character.

captured_subexp	The captured subexpression whose contents should be returned. If omitted or set to 0, the function returns the entire string that matched the regular expression. If set to 1 through 9, the function returns the subexpression captured by the corresponding set of parentheses in the regular expression. For example, setting this value to 3 returns the substring captured by the third set of parentheses in the regular expression. **Note:** The subexpressions are numbered left to right, based on the appearance of opening parenthesis, so nested regular expressions . For example, in the regular expression `\s*(\w+\s+(\w+))`, subexpression 1 is the one that captures everything but any leading whitespaces.

Notes

This function operates on UTF-8 strings using the default locale, even if the locale has been set to something else.

If you are porting a regular expression query from an Oracle database, remember that Oracle considers a zero-length string to be equivalent to NULL, while HP Vertica does not.

Examples

Select the first substring of letters that end with "thy."

```
=> SELECT REGEXP_SUBSTR('healthy, wealthy, and wise','\w+thy');
 REGEXP_SUBSTR
---------------
 healthy
(1 row)
```

Select the first substring of letters that ends with "thy" starting at the second character in the string.

```
=> SELECT REGEXP_SUBSTR('healthy, wealthy, and wise','\w+thy',2);
 REGEXP_SUBSTR
---------------
 ealthy
(1 row)
```

Select the second substring of letters that ends with "thy."

```
=> SELECT REGEXP_SUBSTR('healthy, wealthy, and wise','\w+thy',1,2);
 REGEXP_SUBSTR
---------------
 wealthy
(1 row)
```

Return the contents of the third captured subexpression, which captures the third word in the string.

```
=> SELECT REGEXP_SUBSTR('one two three', '(\w+)\s+(\w+)\s+(\w+)', 1, 1, '', 3);
```

```
 REGEXP_SUBSTR
---------------
 three
(1 row)
```

Sequence Functions

The sequence functions provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

NEXTVAL

Returns the next value in a sequence. Calling NEXTVAL after creating a sequence initializes the sequence with its default value, incrementing a positive value for ascending sequences, and decrementing a negative value for descending sequences. Thereafter, calling NEXTVAL increments the sequence value. NEXTVAL is used in INSERT, COPY, and SELECT statements to create unique values.

Behavior Type

Volatile

Syntax

```
[[db-name.]schema.]sequence_name.NEXTVAL
NEXTVAL('[[db-name.]schema.]sequence_name')
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
sequence_name	Identifies the sequence for which to determine the next value.

Permissions

- SELECT privilege on sequence
- USAGE privilege on sequence schema

Examples

The following example creates an ascending sequence called `my_seq`, starting at 101:

```
CREATE SEQUENCE my_seq START 101;
```

The following command generates the first number in the sequence:

```
SELECT NEXTVAL('my_seq');
 nextval
---------
     101
(1 row)
```

The following command generates the next number in the sequence:

```
SELECT NEXTVAL('my_seq');
 nextval
---------
     102
(1 row)
```

The following command illustrates how `NEXTVAL` is evaluated on a per-row basis, so in this example, both calls to `NEXTVAL` yield the same result:

```
SELECT NEXTVAL('my_seq'), NEXTVAL('my_seq');
nextval  | nextval
---------+---------
     103 |     103
(1 row)
```

The following example illustrates how the `NEXTVAL` is always evaluated first (and here, increments the `my_seq` sequence from its previous value), even when `CURRVAL` precedes `NEXTVAL`:

```
SELECT CURRVAL('my_seq'), NEXTVAL('my_seq');
 currval | nextval
---------+---------
     104 |     104
(1 row)
```

The following example shows how to use `NEXTVAL` in a table `SELECT` statement. Notice that the `nextval` column is incremented by 1 again:

```
SELECT NEXTVAL('my_seq'), product_description FROM product_dimension LIMIT 10;
 nextval |     product_description
---------+-----------------------------
     105 | Brand #2 bagels
     106 | Brand #1 butter
     107 | Brand #6 chicken noodle soup
     108 | Brand #5 golf clubs
     109 | Brand #4 brandy
     110 | Brand #3 lamb
     111 | Brand #11 vanilla ice cream
     112 | Brand #10 ground beef
```

```
     113 | Brand #9 camera case
     114 | Brand #8 halibut
(10 rows)
```

See Also

ALTER SEQUENCE (page 669)

CREATE SEQUENCE (page 765)

CURRVAL (page 353)

DROP SEQUENCE (page 822)

Using Sequences and Sequence Privileges in the Administrator's Guide

CURRVAL

For a sequence generator, returns the LAST value across all nodes returned by a previous invocation of ***NEXTVAL*** (page 351) in the same session. If there were no calls to NEXTVAL after the sequence was created, an error is returned.

Behavior Type

Volatile

Syntax

```
[[db-name.]schema.]sequence_name.CURRVAL
CURRVAL('[[db-name.]schema.]sequence_name')
```

Parameters

`[[db-name.]schema.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
sequence_name	Identifies the sequence for which to return the current value.

Permissions

- SELECT privilege on sequence
- USAGE privilege on sequence schema

Examples

The following example creates an ascending sequence called sequential, starting at 101:

```
CREATE SEQUENCE seq2 START 101;
```

You cannot call CURRVAL until after you have initiated the sequence with NEXTVAL or the system returns an error:

```
SELECT CURRVAL('seq2');
ERROR:  Sequence seq2 has not been accessed in the session
```

Use the NEXTVAL function to generate the first number for this sequence:

```
SELECT NEXTVAL('seq2');
 nextval
---------
     101
(1 row)
```

Now you can use CURRVAL to return the current number from this sequence:

```
SELECT CURRVAL('seq2');
 currval
---------
     101
(1 row)
```

The following command shows how to use CURRVAL in a SELECT statement:

```
CREATE TABLE customer3 (
  lname VARCHAR(25),
  fname VARCHAR(25),
  membership_card INTEGER,
  ID INTEGER
);
INSERT INTO customer3 VALUES ('Brown' ,'Sabra', 072753, CURRVAL('my_seq'));
SELECT CURRVAL('seq2'), lname FROM customer3;
 CURRVAL | lname
---------+-------
     101 | Brown
(1 row)
```

The following example illustrates how the NEXTVAL is always evaluated first (and here, increments the my_seq sequence from its previous value), even when CURRVAL precedes NEXTVAL:

```
   SELECT CURRVAL('my_seq'), NEXTVAL('my_seq');
 currval | nextval
---------+---------
     102 |     102
(1 row)
```

See Also

ALTER SEQUENCE (page 669)

CREATE SEQUENCE (page 765)

DROP SEQUENCE (page 822)

NEXTVAL (page 351)

Using Sequences and Sequence Privileges in the Administrator's Guide

LAST_INSERT_ID

Returns the last value of a column whose value is automatically incremented through the AUTO_INCREMENT or IDENTITY ***column-constraint*** (page 783). If multiple sessions concurrently load the same table, the returned value is the last value generated for an AUTO_INCREMENT column by an insert in that session.

Behavior Type

Volatile

Syntax

```
LAST_INSERT_ID()
```

Privileges

- Table owner
- USAGE privileges on schema

Notes

- This function works only with AUTO_INCREMENT and IDENTITY columns. See ***column-constraints*** (page 783) for the ***CREATE TABLE*** (page 770) statement.
- LAST_INSERT_ID does not work with sequence generators created through the ***CREATE SEQUENCE*** (page 765) statement.

Examples

Create a sample table called `customer4`.

```
=> CREATE TABLE customer4(
     ID IDENTITY(2,2),
     lname VARCHAR(25),
     fname VARCHAR(25),
     membership_card INTEGER
   );
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Gupta', 'Saleem', 475987);
```

Notice that the IDENTITY column has a seed of 2, which specifies the value for the first row loaded into the table, and an increment of 2, which specifies the value that is added to the IDENTITY value of the previous row.

Query the table you just created:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
(1 row)
```

Insert some additional values:

```
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Lee', 'Chen', 598742);
```

Call the LAST_INSERT_ID function:

```
=> SELECT LAST_INSERT_ID();
last_insert_id
----------------
             4
(1 row)
```

Query the table again:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
  4 | Lee   | Chen   |          598742
(2 rows)
```

Add another row:

```
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Davis', 'Bill', 469543);
```

Call the LAST_INSERT_ID function:

```
=> SELECT LAST_INSERT_ID();
 LAST_INSERT_ID
----------------
             6
(1 row)
```

Query the table again:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
  4 | Lee   | Chen   |          598742
  6 | Davis | Bill   |          469543
(3 rows)
```

See Also

ALTER SEQUENCE (page 669)

CREATE SEQUENCE (page 765)

DROP SEQUENCE (page 822)

V_CATALOG.SEQUENCES (page 969)

Using Sequences and Sequence Privileges in the Administrator's Guide

String Functions

String functions perform conversion, extraction, or manipulation operations on strings, or return information about strings.

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types CHAR, VARCHAR, BINARY, and VARBINARY.

Unless otherwise noted, all of the functions listed in this section work on all four data types. As opposed to some other SQL implementations, HP Vertica keeps CHAR strings unpadded internally, padding them only on final output. So converting a CHAR(3) 'ab' to VARCHAR(5) results in a VARCHAR of length 2, not one with length 3 including a trailing space.

Some of the functions described here also work on data of non-string types by converting that data to a string representation first. Some functions work only on character strings, while others work only on binary strings. Many work for both. BINARY and VARBINARY functions ignore multibyte UTF-8 character boundaries.

Non-binary character string functions handle normalized multibyte UTF-8 characters, as specified by the Unicode Consortium. Unless otherwise specified, those character string functions for which it matters can optionally specify whether VARCHAR arguments should be interpreted as octet (byte) sequences, or as (locale-aware) sequences of UTF-8 characters. This is accomplished by adding "USING OCTETS" or "USING CHARACTERS" (default) as a parameter to the function.

Some character string functions are stable because in general UTF-8 case-conversion, searching and sorting can be locale dependent. Thus, LOWER is stable, while LOWERB is immutable. The USING OCTETS clause converts these functions into their "B" forms, so they become immutable. If the locale is set to collation=binary, which is the default, all string functions — except CHAR_LENGTH/CHARACTER_LENGTH, LENGTH, SUBSTR, and OVERLAY — are converted to their "B" forms and so are immutable.

BINARY implicitly converts to VARBINARY, so functions that take VARBINARY arguments work with BINARY.

ASCII

Converts the first octet of a VARCHAR to an INTEGER.

Behavior Type

Immutable

Syntax

```
ASCII ( expression )
```

Parameters

expression	(VARCHAR) is the string to convert.

Notes

- ASCII is the opposite of the ***CHR*** (page 362) function.
- ASCII operates on UTF-8 characters, not only on single-byte ASCII characters. It continues to get the same results for the ASCII subset of UTF-8.

Examples

Expression	Result
`SELECT ASCII('A');`	`65`
`SELECT ASCII('ab');`	`97`
`SELECT ASCII(null);`	
`SELECT ASCII('');`	

BIT_LENGTH

Returns the length of the string expression in bits (bytes * 8) as an INTEGER.

Behavior Type

Immutable

Syntax

```
BIT_LENGTH ( expression )
```

Parameters

expression	(CHAR or VARCHAR or BINARY or VARBINARY) is the string to convert.

Notes

BIT_LENGTH applies to the contents of VARCHAR and VARBINARY fields.

Examples

Expression	Result
`SELECT BIT_LENGTH('abc'::varbinary);`	24
`SELECT BIT_LENGTH('abc'::binary);`	8
`SELECT BIT_LENGTH(''::varbinary);`	0
`SELECT BIT_LENGTH(''::binary);`	8
`SELECT BIT_LENGTH(null::varbinary);`	
`SELECT BIT_LENGTH(null::binary);`	
`SELECT BIT_LENGTH(VARCHAR 'abc');`	24
`SELECT BIT_LENGTH(CHAR 'abc');`	24
`SELECT BIT_LENGTH(CHAR(6) 'abc');`	48
`SELECT BIT_LENGTH(VARCHAR(6) 'abc');`	24
`SELECT BIT_LENGTH(BINARY(6) 'abc');`	48
`SELECT BIT_LENGTH(BINARY 'abc');`	24
`SELECT BIT_LENGTH(VARBINARY 'abc');`	24
`SELECT BIT_LENGTH(VARBINARY(6) 'abc');`	24

See Also

CHARACTER_LENGTH (page 361), ***LENGTH*** (page 381), ***OCTET_LENGTH*** (page 385)

BITCOUNT

Returns the number of one-bits (sometimes referred to as set-bits) in the given VARBINARY value. This is also referred to as the population count.

Behavior Type

Immutable

Syntax

```
BITCOUNT ( expression )
```

Parameters

expression	(BINARY or VARBINARY) is the string to return.

Examples

```
SELECT BITCOUNT(HEX_TO_BINARY('0x10'));
 bitcount
----------
        1
(1 row)
SELECT BITCOUNT(HEX_TO_BINARY('0xF0'));
 bitcount
----------
        4
(1 row)
```

```
SELECT BITCOUNT(HEX_TO_BINARY('0xAB'))
 bitcount
----------
        5
(1 row)
```

BITSTRING_TO_BINARY

Translates the given VARCHAR bitstring representation into a VARBINARY value.

Behavior Type

Immutable

Syntax

```
BITSTRING_TO_BINARY ( expression )
```

Parameters

expression	(VARCHAR) is the string to return.

Notes

VARBINARY BITSTRING_TO_BINARY(VARCHAR) converts data from character type (in bitstring format) to binary type. This function is the inverse of TO_BITSTRING.

```
BITSTRING_TO_BINARY(TO_BITSTRING(x)) = x
TO_BITSTRING(BITSTRING_TO_BINARY(x)) = x
```

Examples

If there are an odd number of characters in the hex value, then the first character is treated as the low nibble of the first (furthest to the left) byte.

```
SELECT BITSTRING_TO_BINARY('0110000101100010');
 bitstring_to_binary
---------------------
 ab
(1 row)
```

If an invalid bitstring is supplied, the system returns an error:

```
SELECT BITSTRING_TO_BINARY('010102010');
ERROR:  invalid bitstring "010102010"
```

BTRIM

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

Behavior Type

Immutable

Syntax

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

Parameters

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

```
SELECT BTRIM('xyxtrimyyx', 'xy');
 btrim
-------
 trim
(1 row)
```

See Also

LTRIM (page 384), ***RTRIM*** (page 396), ***TRIM*** (page 406)

CHARACTER_LENGTH

The CHARACTER_LENGTH() function:

- Returns the string length in UTF-8 characters for CHAR and VARCHAR columns
- Returns the string length in bytes (octets) for BINARY and VARBINARY columns
- Strips the padding from CHAR expressions but not from VARCHAR expressions
- Is identical to ***LENGTH()*** (page 381) for CHAR and VARCHAR. For binary types, CHARACTER_LENGTH() is identical to ***OCTET_LENGTH()*** (page 385).

Behavior Type

Immutable if `USING OCTETS`, stable otherwise.

Syntax

```
[ CHAR_LENGTH | CHARACTER_LENGTH ] ( expression ,
... [ USING { CHARACTERS | OCTETS } ] )
```

Parameters

expression	(CHAR or VARCHAR) is the string to measure
`USING CHARACTERS | OCTETS`	Determines whether the character length is expressed in characters (the default) or octets.

Examples

```
SELECT CHAR_LENGTH('1234  '::CHAR(10), USING OCTETS);
 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)
```

See Also

BIT_LENGTH (page 358)

CHR

Converts the first octet of an INTEGER to a VARCHAR.

Behavior Type

Immutable

Syntax

```
CHR ( expression )
```

Parameters

expression	(INTEGER) is the string to convert and is masked to a single octet.

Notes

- CHR is the opposite of the ***ASCII*** (page 357) function.
- CHR operates on UTF-8 characters, not only on single-byte ASCII characters. It continues to get the same results for the ASCII subset of UTF-8.

Examples

Expression	Result
`SELECT CHR(65);`	`A`
`SELECT CHR(65+32);`	`a`

```
SELECT CHR(null);
```

CONCAT

Used to concatenate two or more VARBINARY strings. The return value is of type VARBINARY.

Syntax

```
CONCAT ('a','b')
```

Behavior type

Immutable

Parameters

a	Is the first VARBINARY string.
b	Is the second VARBINARY string.

Examples

```
=> SELECT CONCAT ('A','B');
 CONCAT
--------
 AB
(1 row)
```

DECODE

Compares *expression* to each search value one by one. If *expression* is equal to a search, the function returns the corresponding result. If no match is found, the function returns default. If default is omitted, the function returns null.

Behavior Type

Immutable

Syntax

```
DECODE ( expression, search, result [ , search, result ]
...[, default ] );
```

Parameters

expression	Is the value to compare.
search	Is the value compared against *expression.*
result	Is the value returned, if *expression* is equal to search.
default	Is optional. If no matches are found, DECODE returns default. If

	default is omitted, then DECODE returns NULL (if no matches are found).

Notes

DECODE is similar to the IF-THEN-ELSE and ***CASE*** (page 52) expression:

```
CASE expression
WHEN search THEN result
[WHEN search THEN result]
[ELSE default];
```

The arguments can have any data type supported by HP Vertica. The result types of individual results are promoted to the least common type that can be used to represent all of them. This leads to a character string type, an exact numeric type, an approximate numeric type, or a DATETIME type, where all the various result arguments must be of the same type grouping.

Examples

The following example converts numeric values in the weight column from the product_dimension table to descriptive values in the output.

```
SELECT product_description, DECODE(weight,
    2, 'Light',
   50, 'Medium',
   71, 'Heavy',
   99, 'Call for help',
       'N/A')
FROM product_dimension
WHERE category_description = 'Food'
AND department_description = 'Canned Goods'
AND sku_number BETWEEN 'SKU-#49750' AND 'SKU-#49999'
LIMIT 15;

        product_description        |     case
-----------------------------------+--------------
 Brand #499 canned corn            | N/A
 Brand #49900 fruit cocktail       | Medium
 Brand #49837 canned tomatoes      | Heavy
 Brand #49782 canned peaches       | N/A
 Brand #49805 chicken noodle soup  | N/A
 Brand #49944 canned chicken broth | N/A
 Brand #49819 canned chili         | N/A
 Brand #49848 baked beans          | N/A
 Brand #49989 minestrone soup      | N/A
 Brand #49778 canned peaches       | N/A
 Brand #49770 canned peaches       | N/A
 Brand #4977 fruit cocktail        | N/A
 Brand #49933 canned olives        | N/A
 Brand #49750 canned olives        | Call for help
 Brand #49777 canned tomatoes      | N/A
(15 rows)
```

GREATEST

Returns the largest value in a list of expressions.

Behavior Type

Stable

Syntax

```
GREATEST ( expression1, expression2, ... expression-n );
```

Parameters

expression1, *expression2*, and *expression-n* are the expressions to be evaluated.

Notes

- Works for all data types, and implicitly casts similar types. See Examples.
- A NULL value in any one of the expressions returns NULL.
- Depends on the collation setting of the locale.

Examples

This example returns 9 as the greatest in the list of expressions:

```
SELECT GREATEST(7, 5, 9);
 greatest
----------
        9
(1 row)
```

Note that putting quotes around the integer expressions returns the same result as the first example:

```
SELECT GREATEST('7', '5', '9');
 greatest
----------
        9
(1 row)
```

The next example returns FLOAT 1.5 as the greatest because the integer is implicitly cast to float:

```
SELECT GREATEST(1, 1.5);
 greatest
----------
      1.5
(1 row)
```

The following example returns 'vertica' as the greatest:

```
SELECT GREATEST('vertica', 'analytic', 'database');
 greatest
----------
 vertica
(1 row)
```

Notice this next command returns NULL:

```
SELECT GREATEST('vertica', 'analytic', 'database', null);
 greatest
----------

(1 row)
```

And one more:

```
SELECT GREATEST('sit', 'site', 'sight');
 greatest
----------
 site
(1 row)
```

See Also

LEAST (page 377)

GREATESTB

Returns its greatest argument, using binary ordering, not UTF-8 character ordering.

Behavior Type

Immutable

Syntax

```
GREATESTB ( expression1, expression2, ... expression-n );
```

Parameters

expression1, *expression2*, and *expression-n* are the expressions to be evaluated.

Notes

- Works for all data types, and implicitly casts similar types. See Examples.
- A NULL value in any one of the expressions returns NULL.
- Depends on the collation setting of the locale.

Examples

The following command selects straße as the greatest in the series of inputs:

```
SELECT GREATESTB('straße', 'strasse');
 GREATESTB
-----------
 straße
(1 row)
```

This example returns 9 as the greatest in the list of expressions:

```
SELECT GREATESTB(7, 5, 9);
 GREATESTB
```

```
-----------
          9
(1 row)
```

Note that putting quotes around the integer expressions returns the same result as the first example:

```
 GREATESTB
-----------
 9
(1 row)
```

The next example returns FLOAT 1.5 as the greatest because the integer is implicitly cast to float:

```
SELECT GREATESTB(1, 1.5);
 GREATESTB
-----------
       1.5
(1 row)
```

The following example returns 'vertica' as the greatest:

```
SELECT GREATESTB('vertica', 'analytic', 'database');
 GREATESTB
-----------
 vertica
(1 row)
```

Notice this next command returns NULL:

```
SELECT GREATESTB('vertica', 'analytic', 'database', null);
 GREATESTB
-----------

(1 row)
```

And one more:

```
SELECT GREATESTB('sit', 'site', 'sight');
 GREATESTB
-----------
 site
(1 row)
```

See Also

LEASTB (page 379)

HEX_TO_BINARY

Translates the given VARCHAR hexadecimal representation into a VARBINARY value.

Behavior Type

Immutable

Syntax

```
HEX_TO_BINARY ( [ 0x ] expression )
```

Parameters

expression	(BINARY or VARBINARY) is the string to translate.
0x	Is optional prefix

Notes

VARBINARY HEX_TO_BINARY(VARCHAR) converts data from character type in hexadecimal format to binary type. This function is the inverse of ***TO_HEX*** (page 260).

```
HEX_TO_BINARY(TO_HEX(x)) = x)
TO_HEX(HEX_TO_BINARY(x)) = x)
```

If there are an odd number of characters in the hexadecimal value, the first character is treated as the low nibble of the first (furthest to the left) byte.

Examples

If the given string begins with "0x" the prefix is ignored. For example:

```
SELECT HEX_TO_BINARY('0x6162') AS hex1, HEX_TO_BINARY('6162') AS hex2;
 hex1 | hex2
------+------
 ab   | ab
(1 row)
```

If an invalid hex value is given, HP Vertica returns an “invalid binary representation" error; for example:

```
SELECT HEX_TO_BINARY('0xffgf');
ERROR:  invalid hex string "0xffgf"
```

See Also

TO_HEX (page 260)

HEX_TO_INTEGER

Translates the given VARCHAR hexadecimal representation into an INTEGER value.

HP Vertica completes this conversion as follows:

- Adds the 0x prefix if it is not specified in the input
- Casts the VARCHAR string to a NUMERIC
- Casts the NUMERIC to an INTEGER

Behavior Type

Immutable

Syntax

```
HEX_TO_INTEGER ( [ 0x ] expression )
```

Parameters

expression	VARCHAR is the string to translate.
0x	Is the optional prefix.

Examples

You can enter the string with or without the Ox prefix. For example:

```
VMart=>  SELECT HEX_TO_INTEGER ('0aedc')AS hex1,HEX_TO_INTEGER ('aedc') AS hex2;
 hex1  | hex2
-------+-------
 44764 | 44764
(1 row)
```

If you pass the function an invalid hex value, HP Vertica returns an `invalid input syntax` error; for example:

```
VMart=> SELECT HEX_TO_INTEGER ('0xffgf');
ERROR 3691:  Invalid input syntax for numeric: "0xffgf"
```

See Also

TO_HEX (page 260)

HEX_TO_BINARY (page 367)

INET_ATON

Returns an integer that represents the value of the address in host byte order, given the dotted-quad representation of a network address as a string.

Behavior Type

Immutable

Syntax

```
INET_ATON ( expression )
```

Parameters

expression	(VARCHAR) is the string to convert.

Notes

The following syntax converts an IPv4 address represented as the string A to an integer I.

INET_ATON trims any spaces from the right of A, calls the Linux function ***inet_pton http://www.opengroup.org/onlinepubs/000095399/functions/inet_ntop.html***, and converts the result from network byte order to host byte order using ***ntohl http://opengroup.org/onlinepubs/007908775/xns/ntohl.html***.

```
INET_ATON(VARCHAR A) -> INT8 I
```

If A is NULL, too long, or inet_pton returns an error, the result is NULL.

Examples

The generated number is always in host byte order. In the following example, the number is calculated as 209×256^3 + 207×256^2 + 224×256 + 40.

```
SELECT INET_ATON('209.207.224.40');
 inet_aton
------------
 3520061480
(1 row)
SELECT INET_ATON('1.2.3.4');
 inet_aton
-----------
  16909060
(1 row)
SELECT TO_HEX(INET_ATON('1.2.3.4'));
 to_hex
---------
 1020304
(1 row)
```

See Also

INET_NTOA (page 293)

INET_NTOA

Returns the dotted-quad representation of the address as a VARCHAR, given a network address as an integer in network byte order.

Behavior Type

Immutable

Syntax

```
INET_NTOA ( expression )
```

Parameters

expression	(INTEGER) is the network address to convert.

Notes

The following syntax converts an IPv4 address represented as integer I to a string A.

INET_NTOA converts I from host byte order to network byte order using ***htonl http://opengroup.org/onlinepubs/007908775/xns/htonl.html***, and calls the Linux function ***inet_ntop http://www.opengroup.org/onlinepubs/000095399/functions/inet_ntop.html***.

```
INET_NTOA(INT8 I) -> VARCHAR A
```

If I is NULL, greater than 2^32 or negative, the result is NULL.

Examples

```
SELECT INET_NTOA(16909060);
 inet_ntoa
-----------
 1.2.3.4
(1 row)
SELECT INET_NTOA(03021962);
  inet_ntoa
-------------
 0.46.28.138
(1 row)
```

See Also

INET_ATON (page 292)

INITCAP

Capitalizes first letter of each alphanumeric word and puts the rest in lowercase. Starting in Release 5.1, this function treats the `string` argument as a UTF-8 encoded string, rather than depending on the collation setting of the locale (for example, collation=binary) to identify the encoding. Prior to Release 5.1, the behavior type of this function was stable.

Behavior Type

Immutable

Syntax

```
INITCAP ( expression )
```

Parameters

expression	(VARCHAR) is the string to format.

Notes

- Depends on collation setting of the locale.
- INITCAP is restricted to 32750 octet inputs, since it is possible for the UTF-8 representation of result to double in size.

Examples

Expression	Result
`SELECT INITCAP('high speed database');`	`High Speed Database`
`SELECT INITCAP('LINUX TUTORIAL');`	`Linux Tutorial`
`SELECT INITCAP('abc DEF 123aVC 124Btd,lAsT');`	`Abc Def 123Avc 124Btd,Last`
`SELECT INITCAP('');`	
`SELECT INITCAP(null);`	

INITCAPB

Capitalizes first letter of each alphanumeric word and puts the rest in lowercase. Multibyte characters are not converted and are skipped.

Behavior Type

Immutable

Syntax

```
INITCAPB ( expression )
```

Parameters

expression	(VARCHAR) is the string to format.

Notes

Depends on collation setting of the locale.

Examples

Expression	Result
`SELECT INITCAPB('étudiant');`	`éTudiant`
`SELECT INITCAPB('high speed database');`	`High Speed Database`
`SELECT INITCAPB('LINUX TUTORIAL');`	`Linux Tutorial`
`SELECT INITCAPB('abc DEF 123aVC 124Btd,lAsT');`	`Abc Def 123Avc 124Btd,Last`
`SELECT INITCAPB('');`	
`SELECT INITCAPB(null);`	

INSERT

Inserts a character string into a specified location in another character string.

Syntax

```
INSERT( 'string1', n, m, 'string2');
```

Behavior type

Immutable

Parameters

string1	(VARCHAR) Is the string in which to insert the new string.
n	A character of type INTEGER that represents the starting point for the insertion within *string1*. You specify the number of characters from the first character in string1 as the starting point for the insertion. For example, to insert characters before "c", in the string "abcdef," enter 3.
m	A character of type INTEGER that represents the the number of characters in *string1* (if any) that should be replaced by the insertion. For example,if you want the insertion to replace the letters "cd" in the string "abcdef, " enter 2.
string2	(VARCHAR) Is the string to be inserted.

Example

The following example changes the string Warehouse to Storehouse using the INSERT function:

```
=> SELECT INSERT ('Warehouse',1,3,'Stor');
   INSERT
------------
 Storehouse
(1 row)
```

INSTR

Searches *string* for *substring* and returns an integer indicating the position of the character in *string* that is the first character of this *occurrence*. The return value is based on the character position of the identified character. Starting in Release 5.1, this function treats the `string` argument as a UTF-8 encoded string, rather than depending on the collation setting of the locale (for example, collation=binary) to identify the encoding. Prior to Release 5.1, the behavior type of this function was stable.

Behavior Type

Immutable

Syntax

```
INSTR ( string , substring [, position [, occurrence ] ] )
```

Parameters

string	(CHAR or VARCHAR, or BINARY or VARBINARY) Is the text expression to search.
substring	(CHAR or VARCHAR, or BINARY or VARBINARY) Is the string to search for.
position	Is a nonzero integer indicating the character of string where HP Vertica begins the search. If position is negative, then HP Vertica counts backward from the end of string and then searches backward from the resulting position. The first character of string occupies the default position 1, and position cannot be 0.
occurrence	Is an integer indicating which occurrence of string HP Vertica searches. The value of occurrence must be positive (greater than 0), and the default is 1.

Notes

Both *position* and *occurrence* must be of types that can resolve to an integer. The default values of both parameters are 1, meaning HP Vertica begins searching at the first character of string for the first occurrence of substring. The return value is relative to the beginning of string, regardless of the value of position, and is expressed in characters.

If the search is unsuccessful (that is, if substring does not appear *occurrence* times after the *position* character of *string,* then the return value is 0.

Examples

The first example searches forward in string ‘abc’ for substring ‘b’. The search returns the position in ‘abc’ where ‘b’ occurs, or position 2. Because no position parameters are given, the default search starts at ‘a’, position 1.

```
SELECT INSTR('abc', 'b');
INSTR
-------
     2
(1 row)
```

The following three examples use character position to search backward to find the position of a substring.

Note: Although it might seem intuitive that the function returns a negative integer, the position of *n* occurrence is read left to right in the sting, even though the search happens in reverse (from the end — or right side — of the string).

In the first example, the function counts backward one character from the end of the string, starting with character ‘c’. The function then searches backward for the first occurrence of ‘a’, which it finds it in the first position in the search string.

```
SELECT INSTR('abc', 'a', -1);
 INSTR
-------
     1
```

```
(1 row)
```

In the second example, the function counts backward one byte from the end of the string, starting with character 'c'. The function then searches backward for the first occurrence of 'a', which it finds it in the first position in the search string.

```
SELECT INSTR(VARBINARY 'abc', VARBINARY 'a', -1);
 INSTR
-------
     1
(1 row)
```

In the third example, the function counts backward one character from the end of the string, starting with character 'b', and searches backward for substring 'bc', which it finds in the second position of the search string.

```
SELECT INSTR('abcb', 'bc', -1);
 INSTR
-------
     2
(1 row)
```

In the fourth example, the function counts backward one character from the end of the string, starting with character 'b', and searches backward for substring 'bcef', which it does not find. The result is 0.

```
SELECT INSTR('abcb', 'bcef', -1);
INSTR
-------
     0
(1 row)
```

In the fifth example, the function counts backward one byte from the end of the string, starting with character 'b', and searches backward for substring 'bcef', which it does not find. The result is 0.

SELECT INSTR(VARBINARY 'abcb', VARBINARY 'bcef', -1);

```
 INSTR
-------
     0
(1 row)
```

Multibyte characters are treated as a single character:

```
dbadmin=> SELECT INSTR('aébc', 'b');
 INSTR
-------
     3
(1 row)
```

Use INSTRB to treat multibyte characters as binary:

```
dbadmin=> SELECT INSTRB('aébc', 'b');
 INSTRB
--------
      4
```

```
(1 row)
```

INSTRB

Searches *string* for *substring* and returns an integer indicating the octet position within string that is the first *occurrence*. The return value is based on the octet position of the identified byte.

Behavior Type

Immutable

Syntax

```
INSTRB ( string , substring [, position [, occurrence ] ] )
```

Parameters

string	Is the text expression to search.
substring	Is the string to search for.
position	Is a nonzero integer indicating the character of string where HP Vertica begins the search. If position is negative, then HP Vertica counts backward from the end of string and then searches backward from the resulting position. The first byte of string occupies the default position 1, and position cannot be 0.
occurrence	Is an integer indicating which occurrence of string HP Vertica searches. The value of occurrence must be positive (greater than 0), and the default is 1.

Notes

Both *position* and *occurrence* must be of types that can resolve to an integer. The default values of both parameters are 1, meaning HP Vertica begins searching at the first byte of string for the first occurrence of substring. The return value is relative to the beginning of string, regardless of the value of position, and is expressed in octets.

If the search is unsuccessful (that is, if substring does not appear *occurrence* times after the *position* character of *string,* then the return value is 0.

Examples

```
SELECT INSTRB('straße', 'ß');
 INSTRB
--------
      5
(1 row)
```

See Also

INSTR (page 373)

ISUTF8

Tests whether a string is a valid UTF-8 string. Returns true if the string conforms to UTF-8 standards, and false otherwise. This function is useful to test strings for UTF-8 compliance before passing them to one of the regular expression functions, such as ***REGEXP_LIKE*** (page 341), which expect UTF-8 characters by default.

ISUTF8 checks for invalid UTF8 byte sequences, according to UTF-8 rules:

- invalid bytes
- an unexpected continuation byte
- a start byte not followed by enough continuation bytes
- an Overload Encoding

The presence of an invalid UTF8 byte sequence results in a return value of false.

Syntax

```
ISUTF8( string );
```

Parameters

string	The string to test for UTF-8 compliance.

Examples

```
=> SELECT ISUTF8(E'\xC2\xBF'); -- UTF-8 INVERTED QUESTION MARK
 ISUTF8
--------
 t
(1 row)

=> SELECT ISUTF8(E'\xC2\xC0'); -- UNDEFINED UTF-8 CHARACTER
 ISUTF8
--------
 f
(1 row)
```

LEAST

Returns the smallest value in a list of expressions.

Behavior Type

Stable

Syntax

```
LEAST ( expression1, expression2, ... expression-n );
```

Parameters

expression1, *expression2*, and *expression-n* are the expressions to be evaluated.

Notes

- Works for all data types, and implicitly casts similar types. See Examples below.
- A NULL value in any one of the expressions returns NULL.

Examples

This example returns 5 as the least:

```
SELECT LEAST(7, 5, 9);
 least
-------
     5
(1 row)
```

Note that putting quotes around the integer expressions returns the same result as the first example:

```
SELECT LEAST('7', '5', '9');
 least
-------
 5
(1 row)
```

In the above example, the values are being compared as strings, so '10' would be less than '2'.

The next example returns 1.5, as INTEGER 2 is implicitly cast to FLOAT:

```
SELECT LEAST(2, 1.5);
 least
-------
   1.5
(1 row)
```

The following example returns 'analytic' as the least:

```
SELECT LEAST('vertica', 'analytic', 'database');
  least
----------
 analytic
(1 row)
```

Notice this next command returns NULL:

```
SELECT LEAST('vertica', 'analytic', 'database', null);
 least
-------

(1 row)
```

And one more:

```
SELECT LEAST('sit', 'site', 'sight');
 least
-------
```

```
 sight
(1 row)
```

See Also

GREATEST (page 365)

LEASTB

Returns the function's least argument, using binary ordering, not UTF-8 character ordering.

Behavior Type

Immutable

Syntax

```
LEASTB ( expression1, expression2, ... expression-n );
```

Parameters

expression1, *expression2*, and *expression-n* are the expressions to be evaluated.

Notes

- Works for all data types, and implicitly casts similar types. See Examples below.
- A NULL value in any one of the expressions returns NULL.

Examples

The following command selects strasse as the least in the series of inputs:

```
SELECT LEASTB('straße', 'strasse');
 LEASTB
---------
 strasse
(1 row)
```

This example returns 5 as the least:

```
SELECT LEASTB(7, 5, 9);
 LEASTB
--------
      5
(1 row)
```

Note that putting quotes around the integer expressions returns the same result as the first example:

```
SELECT LEASTB('7', '5', '9');
 LEASTB
--------
 5
(1 row)
```

In the above example, the values are being compared as strings, so '10' would be less than '2'.

The next example returns 1.5, as INTEGER 2 is implicitly cast to FLOAT:

```
SELECT LEASTB(2, 1.5);
 LEASTB
--------
    1.5
(1 row)
```

The following example returns 'analytic' as the least in the series of inputs:

```
SELECT LEASTB('vertica', 'analytic', 'database');
  LEASTB
----------
 analytic
(1 row)
```

Notice this next command returns NULL:

```
SELECT LEASTB('vertica', 'analytic', 'database', null);
 LEASTB
--------

(1 row)
```

See Also

GREATESTB (page 366)

LEFT

Returns the specified characters from the left side of a string.

Behavior Type

Immutable

Syntax

```
LEFT ( string , length )
```

Parameters

string	(CHAR or VARCHAR) is the string to return.
length	Is an INTEGER value that specifies the count of characters to return.

Examples

```
SELECT LEFT('vertica', 3);
 left
------
 ver
(1 row)
```

```
SELECT LEFT('straße', 5);
 LEFT
-------
 straß
(1 row)
```

See Also

SUBSTR (page 400)

LENGTH

The LENGTH() function:

- Returns the string length in UTF-8 characters for CHAR and VARCHAR columns
- Returns the string length in bytes (octets) for BINARY and VARBINARY columns
- Strips the padding from CHAR expressions but not from VARCHAR expressions
- Is is identical to ***CHARACTER_LENGTH*** (page 361) for CHAR and VARCHAR. For binary types, LENGTH() is identical to ***OCTET_LENGTH*** (page 385).

Behavior Type

Immutable

Syntax

```
LENGTH ( expression )
```

Parameters

expression	(CHAR or VARCHAR or BINARY or VARBINARY) is the string to measure

Examples

Expression	Result
`SELECT LENGTH('1234  '::CHAR(10));`	`4`
`SELECT LENGTH('1234  '::VARCHAR(10));`	`6`
`SELECT LENGTH('1234  '::BINARY(10));`	`10`
`SELECT LENGTH('1234  '::VARBINARY(10));`	`6`
`SELECT LENGTH(NULL::CHAR(10)) IS NULL;`	`t`

See Also

BIT_LENGTH (page 358)

LOWER

Returns a VARCHAR value containing the argument converted to lowercase letters. Starting in Release 5.1, this function treats the `string` argument as a UTF-8 encoded string, rather than depending on the collation setting of the locale (for example, collation=binary) to identify the encoding. Prior to Release 5.1, the behavior type of this function was stable.

Behavior Type

Immutable

Syntax

```
LOWER ( expression )
```

Parameters

expression	(CHAR or VARCHAR) is the string to convert

Notes

LOWER is restricted to 32750 octet inputs, since it is possible for the UTF-8 representation of result to double in size.

Examples

```
SELECT LOWER('AbCdEfG');
  lower
----------
 abcdefg
(1 row)
SELECT LOWER('The Cat In The Hat');
       lower
--------------------
 the cat in the hat
(1 row)
dbadmin=> SELECT LOWER('ÉTUDIANT');
  LOWER
----------
 étudiant
(1 row)
```

LOWERB

Returns a character string with each ASCII character converted to lowercase. Multibyte characters are not converted and are skipped.

Behavior Type

Immutable

Syntax

```
LOWERB ( expression )
```

Parameters

expression	(CHAR or VARCHAR) is the string to convert

Examples

In the following example, the multibyte UTF-8 character É is not converted to lowercase:

```
SELECT LOWERB('ÉTUDIANT');
  LOWERB
----------
 Étudiant
(1 row)

dbadmin=> SELECT LOWER('ÉTUDIANT');
  LOWER
----------
 étudiant
(1 row)
SELECT LOWERB('AbCdEfG');
 LOWERB
---------
 abcdefg
(1 row)
SELECT LOWERB('The Vertica Database');
        LOWERB
----------------------
 the vertica database
(1 row)
```

LPAD

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

Behavior Type

Immutable

Syntax

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

Parameters

expression	(CHAR OR VARCHAR) specifies the string to fill

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

Examples

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

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

```
SELECT LPAD('establishment', 10, 'abc');
    lpad
------------
 establishm
(1 row)
```

LTRIM

Returns a VARCHAR value representing a string with leading blanks removed from the left side (beginning).

Behavior Type

Immutable

Syntax

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

Parameters

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)
```

See Also

BTRIM (page 360), ***RTRIM*** (page 396), ***TRIM*** (page 406)

MD5

Calculates the MD5 hash of string, returning the result as a VARCHAR string in hexadecimal.

Behavior Type

Immutable

Syntax

```
MD5 ( string )
```

Parameters

string	Is the argument string.

Examples

```
SELECT MD5('123');
               md5
----------------------------------
 202cb962ac59075b964b07152d234b70
(1 row)

SELECT MD5('Vertica'::bytea);
               md5
----------------------------------
 fc45b815747d8236f9f6fdb9c2c3f676
(1 row)
```

OCTET_LENGTH

Takes one argument as an input and returns the string length in octets for all string types.

Behavior Type

Immutable

Syntax

```
OCTET_LENGTH ( expression )
```

Parameters

expression	(CHAR or VARCHAR or BINARY or VARBINARY) is the string to measure.

Notes

- If the data type of *expression* is a CHAR, VARCHAR or VARBINARY, the result is the same as the actual length of *expression* in octets. For CHAR, the length does not include any trailing spaces.

- If the data type of *expression* is BINARY, the result is the same as the fixed-length of *expression*.
- If the value of *expression* is NULL, the result is NULL.

Examples

Expression	Result
`SELECT OCTET_LENGTH(CHAR(10) '1234  ');`	4
`SELECT OCTET_LENGTH(CHAR(10) '1234');`	4
`SELECT OCTET_LENGTH(CHAR(10) '  1234');`	6
`SELECT OCTET_LENGTH(VARCHAR(10) '1234  ');`	6
`SELECT OCTET_LENGTH(VARCHAR(10) '1234 ');`	5
`SELECT OCTET_LENGTH(VARCHAR(10) '1234');`	4
`SELECT OCTET_LENGTH(VARCHAR(10) '  1234');`	7
`SELECT OCTET_LENGTH('abc'::VARBINARY);`	3
`SELECT OCTET_LENGTH(VARBINARY 'abc');`	3
`SELECT OCTET_LENGTH(VARBINARY 'abc  ');`	5
`SELECT OCTET_LENGTH(BINARY(6) 'abc');`	6
`SELECT OCTET_LENGTH(VARBINARY '');`	0
`SELECT OCTET_LENGTH(''::BINARY);`	1
`SELECT OCTET_LENGTH(null::VARBINARY);`	
`SELECT OCTET_LENGTH(null::BINARY);`	

See Also

BIT_LENGTH (page 358), ***CHARACTER_LENGTH*** (page 361), ***LENGTH*** (page 381)

OVERLAY

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

Behavior Type

Immutable if using OCTETS, Stable otherwise

Syntax

```
OVERLAY ( expression1 PLACING expression2 FROM position
... [ FOR extent ]
... [ USING { CHARACTERS | OCTETS } ] )
```

Parameters

expression1	(CHAR or VARCHAR) is the string to process
expression2	(CHAR or VARCHAR) is the substring to overlay
position	(INTEGER) is the character or octet position (counting from one)

	at which to begin the overlay
extent	(INTEGER) specifies the number of characters or octets to replace with the overlay
USING CHARACTERS | OCTETS	Determines whether OVERLAY uses characters (the default) or octets

Examples

```
SELECT OVERLAY('123456789' PLACING 'xxx' FROM 2);
  overlay
-----------
 1xxx56789
(1 row)
SELECT OVERLAY('123456789' PLACING 'XXX' FROM 2 USING OCTETS);
 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)
```

OVERLAYB

Returns an octet value representing a string having had a substring replaced by another string.

Behavior Type

Immutable

Syntax

OVERLAYB (*expression1*, *expression2*, *position* [, *extent*])

Parameters

expression1	(CHAR or VARCHAR) is the string to process
expression2	(CHAR or VARCHAR) is the substring to overlay
position	(INTEGER) is the octet position (counting from one) at which to begin the

	overlay
extent	(INTEGER) specifies the number of octets to replace with the overlay

Notes

This function treats the multibyte character string as a string of octets (bytes) and use octet numbers as incoming and outgoing position specifiers and lengths. The strings themselves are type VARCHAR, but they treated as if each byte was a separate character.

Examples

```
SELECT OVERLAYB('123456789', 'ééé', 2);
 OVERLAYB
----------
 1ééé89
(1 row)
SELECT OVERLAYB('123456789', 'ßßß', 2);
 OVERLAYB
----------
 1ßßß89
(1 row)
SELECT OVERLAYB('123456789', 'xxx', 2);
  OVERLAYB
-----------
 1xxx56789
(1 row)
SELECT OVERLAYB('123456789', 'xxx', 2, 4);
 OVERLAYB
----------
 1xxx6789
(1 row)
SELECT OVERLAYB('123456789', 'xxx', 2, 5);
 OVERLAYB
---------
 1xxx789
(1 row)
SELECT OVERLAYB('123456789', 'xxx', 2, 6);
 OVERLAYB
---------
 1xxx89
(1 row)
```

POSITION

Returns an INTEGER value representing the character location of a specified substring with a string (counting from one). Starting in Release 5.1, this function treats the `string` argument as a UTF-8 encoded string, rather than depending on the collation setting of the locale (for example, collation=binary) to identify the encoding. Prior to Release 5.1, the behavior type of this function was stable.

Behavior Type

Immutable

Syntax 1

```
POSITION ( substring IN string [ USING { CHARACTERS | OCTETS } ] )
```

Parameters

substring	(CHAR or VARCHAR) is the substring to locate
string	(CHAR or VARCHAR) is the string in which to locate the substring
USING CHARACTERS | OCTETS	Determines whether the position is reported by using characters (the default) or octets.

Syntax 2

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

Parameters

substring	(VARBINARY) is the substring to locate
string	(VARBINARY) is the string in which to locate the substring

Notes

- When the string and substring are CHAR or VARCHAR, the return value is based on either the character or octet position of the substring.
- When the string and substring are VARBINARY, the return value is always based on the octet position of the substring.
- The string and substring must be consistent. Do not mix VARBINARY with CHAR or VARCHAR.

Examples

```
SELECT POSITION('é' IN 'étudiant' USING CHARACTERS);
 position
----------
        1
(1 row)
SELECT POSITION('ß' IN 'straße' USING OCTETS);
 position
----------
        5
(1 row)
SELECT POSITION('c' IN 'abcd' USING CHARACTERS);
 position
----------
        3
(1 row)
SELECT POSITION(VARBINARY '456' IN VARBINARY '123456789');
 position
```

```
----------
         4
(1 row)

SELECT POSITION('n' in 'León') as 'default',
       POSITIONB('León', 'n') as 'POSITIONB',
       POSITION('n' in 'León' USING CHARACTERS) as 'pos_chars',
       POSITION('n' in 'León' USING OCTETS) as 'pos_oct',INSTR('León','n'),
       INSTRB('León','n'),REGEXP_INSTR('León','n');

-[ RECORD 1 ]+--
default      | 4
POSITIONB    | 5
pos_chars    | 4
pos_oct      | 5
INSTR        | 4
INSTRB       | 5
REGEXP_INSTR | 4
```

POSITIONB

Returns an INTEGER value representing the octet location of a specified substring with a string (counting from one).

Behavior Type

Immutable

Syntax

```
POSITIONB ( string, substring )
```

Parameters

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

Examples

```
SELECT POSITIONB('straße', 'ße');
 POSITIONB
-----------
          5
(1 row)
SELECT POSITIONB('étudiant', 'é');
 position
----------
         1
(1 row)
```

QUOTE_IDENT

Returns the given string, suitably quoted, to be used as an ***identifier*** (page 22) in a SQL statement string. Quotes are added only if necessary; that is, if the string contains non-identifier characters, is a SQL ***keyword*** (page 19), such as `'1time'`, `'Next week'` and `'Select'`. Embedded double quotes are doubled.

Behavior Type

Immutable

Syntax

```
QUOTE_IDENT( string )
```

Parameters

string	Is the argument string.

Notes

- SQL identifiers, such as table and column names, are stored as created, and references to them are resolved using case-insensitive compares. Thus, you do not need to double-quote mixed-case identifiers.
- HP Vertica quotes all currently-reserved keywords, even those not currently being used.

Examples

Quoted identifiers are case-insensitive, and HP Vertica does not supply the quotes:

```
SELECT QUOTE_IDENT('VErtIcA');
 QUOTE_IDENT
-------------
 VErtIcA
(1 row)

SELECT QUOTE_IDENT('Vertica database');
    QUOTE_IDENT
--------------------
 "Vertica database"
(1 row)
```

Embedded double quotes are doubled:

```
SELECT QUOTE_IDENT('Vertica "!" database');
       QUOTE_IDENT
-------------------------
 "Vertica ""!"" database"
(1 row)
```

The following example uses the SQL keyword, SELECT; results are double quoted:

```
 SELECT QUOTE_IDENT('select');
 QUOTE_IDENT
```

```
-------------
 "select"
(1 row)
```

QUOTE_LITERAL

Returns the given string, suitably quoted, to be used as a string literal in a SQL statement string. Embedded single quotes and backslashes are doubled.

Behavior Type

Immutable

Syntax

```
QUOTE_LITERAL ( string )
```

Parameters

string	Is the argument string.

Notes

HP Vertica recognizes two consecutive single quotes within a string literal as one single quote character. For example, `'You''re here!'`. This is the SQL standard representation and is preferred over the form, `'You\'re here!'`, as backslashes are not parsed as before.

Examples

```
SELECT QUOTE_LITERAL('You''re here!');
  QUOTE_LITERAL
-----------------
 'You''re here!'
(1 row)

SELECT QUOTE_LITERAL('You\'re here!');
WARNING:  nonstandard use of \' in a string literal at character 22
HINT:  Use '' to write quotes in strings, or use the escape string syntax (E'\'').
```

See Also

String Literals (Character) (page 26)

REPEAT

Returns a VARCHAR or VARBINARY value that repeats the given value COUNT times, given a value and a count this function.

If the return value is truncated the given value might not be repeated count times, and the last occurrence of the given value might be truncated.

Behavior Type

Immutable

Syntax

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

Parameters

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

Notes

If the repetitions field depends on the contents of a column (is not a constant), then the repeat operator maximum length is 65000 bytes. You can add a cast of the repeat to cast the result down to a size big enough for your purposes (reflects the actual maximum size) so you can do other things with the result.

REPEAT () and || check for result strings longer than 65000. REPEAT () silently truncates to 65000 octets, and || reports an error (including the octet length).

Examples

The following example repeats 'vmart' three times:

```
SELECT REPEAT ('vmart', 3);
     repeat
-----------------
 vmartvmartvmart
(1 row)
```

If you run the following example, you get an error message:

```
SELECT '123456' || REPEAT('a', colx);
ERROR: Operator || may give a 65006-byte Varchar result; the limit is 65000 bytes.
```

If you know that `colx` can never be greater than 3, the solution is to add a cast (::VARCHAR(3)):

```
SELECT '123456' || REPEAT('a', colx)::VARCHAR(3);
```

If `colx` is greater than 3, the repeat is truncated to exactly three (3) a's.

REPLACE

Replaces all occurrences of characters in a string with another set of characters.

Behavior Type

Immutable

Syntax

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

Parameters

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)
SELECT REPLACE('This & That', '&', 'and');
    replace
---------------
 This and That
(1 row)
SELECT REPLACE('straße', 'ß', 'ss');
 REPLACE
---------
 strasse
(1 row)
```

RIGHT

Returns the specified characters from the right side of a string.

Behavior Type

Immutable

Syntax

```
RIGHT ( string , length )
```

Parameters

string	(CHAR or VARCHAR) is the string to return.
length	Is an INTEGER value that specifies the count of characters to return.

Examples

The following command returns the last three characters of the string 'vertica':

```
SELECT RIGHT('vertica', 3);
 right
```

```
-------
 ica
(1 row)
```

The following command returns the last two characters of the string 'straße':

```
SELECT RIGHT('straße', 2);
 RIGHT
-------
 ße
(1 row)
```

See Also

SUBSTR (page 400)

RPAD

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

Behavior Type

Immutable

Syntax

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

Parameters

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

Examples

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

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

```
SELECT RPAD('database', 6, 'xzy');
  rpad
--------
 databa
(1 row)
```

RTRIM

Returns a VARCHAR value representing a string with trailing blanks removed from the right side (end).

Behavior Type

Immutable

Syntax

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

Parameters

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('trimzzzyyyyyyyxxxxxxxx', 'xyz');
 ltrim
-------
 trim
(1 row)
```

See Also

BTRIM (page 360), ***LTRIM*** (page 384), ***TRIM*** (page 406)

SPACE

Inserts blank spaces into a specified location within a character string.

Syntax

```
SELECT INSERT( 'string1', || SPACE (n) || 'string2');
```

Parameters

string1	(VARCHAR) Is the string after which to insert the space.
n	A character of type INTEGER that represents the number of spaces to insert.
string2	(VARCHAR) Is the remainder of the string that appears after the inserted spaces

Example

The following example inserts 10 spaces between the strings 'x' and 'y':

```
SELECT 'x' || SPACE(10) || 'y';
   ?column?
--------------
 x          y
(1 row)
```

SPLIT_PART

Splits string on the delimiter and returns the location of the beginning of the given field (counting from one). Starting in Release 5.1, this function treats the `string` argument as a UTF-8 encoded string, rather than depending on the collation setting of the locale (for example, collation=binary) to identify the encoding. Prior to Release 5.1, the behavior type of this function was stable.

Behavior Type

Immutable

Syntax

```
SPLIT_PART ( string , delimiter , field )
```

Parameters

string	Is the argument string.
delimiter	Is the given delimiter.
field	(INTEGER) is the number of the part to return.

Note

Use this with the character form of the subfield.

Examples

The specified integer of 2 returns the second string, or `def`.

```
SELECT SPLIT_PART('abc~@~def~@~ghi', '~@~', 2);
 split_part
------------
 def
(1 row)
```

Here, we specify 3, which returns the third string, or `789`.

```
SELECT SPLIT_PART('123~|~456~|~789', '~|~', 3);
 split_part
------------
 789
(1 row)
```

Note that the tildes are for readability only. Omitting them returns the same results:

```
SELECT SPLIT_PART('123|456|789', '|', 3);
```

```
 split_part
------------
 789
(1 row)
```

See what happens if you specify an integer that exceeds the number of strings: No results.

```
SELECT SPLIT_PART('123|456|789', '|', 4);
 split_part
------------

(1 row)
```

The above result is not null, it is the empty string.

```
SELECT SPLIT_PART('123|456|789', '|', 4) IS NULL;
 ?column?
----------
 f
(1 row)
```

If SPLIT_PART had returned NULL, LENGTH would have returned null.

```
SELECT LENGTH (SPLIT_PART('123|456|789', '|', 4));
 length
--------
      0
(1 row)
```

SPLIT_PARTB

Splits string on the delimiter and returns the location of the beginning of the given field (counting from one). The VARCHAR arguments are treated as octets rather than UTF-8 characters.

Behavior Type

Immutable

Syntax

```
SPLIT_PARTB ( string , delimiter , field )
```

Parameters

string	(VARCHAR) Is the argument string.
delimiter	(VARCHAR) Is the given delimiter.
field	(INTEGER) is the number of the part to return.

Note

Use this function with the character form of the subfield.

Examples

The specified integer of 3 returns the third string, or `soupçon`.

```
SELECT SPLIT_PARTB('straße~@~café~@~soupçon', '~@~', 3);
 SPLIT_PARTB
-------------
 soupçon
(1 row)
```

Note that the tildes are for readability only. Omitting them returns the same results:

```
SELECT SPLIT_PARTB('straße @ café @ soupçon', '@', 3);
 SPLIT_PARTB
-------------
  soupçon
(1 row)
```

See what happens if you specify an integer that exceeds the number of strings: No results.

```
 SELECT SPLIT_PARTB('straße @ café @ soupçon', '@', 4);
 SPLIT_PARTB
-------------

(1 row)
```

The above result is not null, it is the empty string.

```
SELECT SPLIT_PARTB('straße @ café @ soupçon', '@', 4) IS NULL;
 ?column?
----------
 f
(1 row)
```

STRPOS

Returns an INTEGER value representing the character location of a specified substring within a string (counting from one). Starting in Release 5.1, this function treats the `string` argument as a UTF-8 encoded string, rather than depending on the collation setting of the locale (for example, collation=binary) to identify the encoding. Prior to Release 5.1, the behavior type of this function was stable.

Behavior Type

Immutable

Syntax

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

Parameters

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 388) except for the order of the arguments.

Examples

```
SELECT STRPOS('abcd','c');
 strpos
--------
      3
(1 row)
```

STRPOSB

Returns an INTEGER value representing the location of a specified substring within a string, counting from one, where each octet in the string is counted (as opposed to characters).

Behavior Type

Immutable

Syntax

```
STRPOSB ( string , substring )
```

Parameters

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

Notes

STRPOSB is identical to ***POSITIONB*** (page 390) except for the order of the arguments.

Examples

```
dbadmin=> SELECT STRPOSB('straße', 'e');
 STRPOSB
---------
       7
(1 row)

dbadmin=> SELECT STRPOSB('étudiant', 'tud');
 STRPOSB
---------
       3
(1 row)
```

SUBSTR

Returns VARCHAR or VARBINARY value representing a substring of a specified string.

Behavior Type

Immutable

Syntax

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

Parameters

string	(CHAR/VARCHAR or BINARY/VARBINARY) is the string from which to extract a substring.
position	(INTEGER or DOUBLE PRECISION) is the starting position of the substring (counting from one by characters).
extent	(INTEGER or DOUBLE PRECISION) is the length of the substring to extract (in characters). The default is the end of the string.

Notes

SUBSTR truncates DOUBLE PRECISION input values.

Examples

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

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

=> SELECT SUBSTR(TO_BITSTRING(HEX_TO_BINARY('0x10')), 2, 2);
 substr
--------
 00
(1 row)

=> SELECT SUBSTR(TO_HEX(10010), 2, 2);
 substr
--------
 71
(1 row)
```

SUBSTRB

Returns an octet value representing the substring of a specified string.

Behavior Type

Immutable

Syntax

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

Parameters

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

Notes

- This function treats the multibyte character string as a string of octets (bytes) and uses octet numbers as incoming and outgoing position specifiers and lengths. The strings themselves are type VARCHAR, but they treated as if each octet were a separate character.
- SUBSTRB truncates DOUBLE PRECISION input values.

Examples

```
=> SELECT SUBSTRB('soupçon', 5);
 SUBSTRB
---------
 çon
(1 row)

=> SELECT SUBSTRB('soupçon', 5, 2);
 SUBSTRB
---------
 ç
(1 row)
```

HP Vertica returns the following error message if you use BINARY/VARBINARY:

```
=>SELECT SUBSTRB('abc'::binary(3),1);
ERROR: function substrb(binary, int) does not exist, or permission is denied for
substrb(binary, int)
HINT: No function matches the given name and argument types. You may need to add
explicit type casts.
```

SUBSTRING

Returns a value representing a substring of the specified string at the given position, given a value, a position, and an optional length.

Behavior Type

Immutable if USING OCTETS, stable otherwise.

Syntax

```
SUBSTRING ( string , position [ , length ]
... [USING {CHARACTERS | OCTETS } ] )
SUBSTRING ( string FROM position [ FOR length ]
... [USING { CHARACTERS | OCTETS } ] )
```

Parameters

string	(CHAR/VARCHAR or BINARY/VARBINARY) is the string from which to extract a substring
position	(INTEGER or DOUBLE PRECISION) is the starting position of the substring (counting from one by either characters or octets). (The default is characters.) If position is greater than the length of the given value, an empty value is returned.
length	(INTEGER or DOUBLE PRECISION) is the length of the substring to extract in either characters or octets. (The default is characters.) The default is the end of the string.If a length is given the result is at most that many bytes. The maximum length is the length of the given value less the given position. If no length is given or if the given length is greater than the maximum length then the length is set to the maximum length.
USING CHARACTERS | OCTETS	Determines whether the value is expressed in characters (the default) or octets.

Notes

- SUBSTRING truncates DOUBLE PRECISION input values.
- Neither length nor position can be negative, and the position cannot be zero because it is one based. If these forms are violated, the system returns an error:

```
SELECT SUBSTRING('ab'::binary(2), -1, 2);
ERROR:  negative or zero substring start position not allowed
```

Examples

```
=> SELECT SUBSTRING('abc'::binary(3),1);
 substring
-----------
 abc
(1 row)

=> SELECT SUBSTRING('soupçon', 5, 2 USING CHARACTERS);
```

```
 substring
-----------
 ço
(1 row)

=> SELECT SUBSTRING('soupçon', 5, 2 USING OCTETS);
 substrb
---------
 ç
(1 row)
```

TO_BITSTRING

Returns a VARCHAR that represents the given VARBINARY value in bitstring format

Behavior Type

Immutable

Syntax

```
TO_BITSTRING ( expression )
```

Parameters

expression	(VARCHAR) is the string to return.

Notes

VARCHAR TO_BITSTRING(VARBINARY) converts data from binary type to character type (where the character representation is the bitstring format). This function is the inverse of BITSTRING_TO_BINARY:

```
TO_BITSTRING(BITSTRING_TO_BINARY(x)) = x)
BITSTRING_TO_BINARY(TO_BITSTRING(x)) = x)
```

Examples

```
SELECT TO_BITSTRING('ab'::BINARY(2));
   to_bitstring
------------------
 0110000101100010
(1 row)
SELECT TO_BITSTRING(HEX_TO_BINARY('0x10'));
to_bitstring
--------------
00010000
(1 row)
```

```
SELECT TO_BITSTRING(HEX_TO_BINARY('0xF0'));
to_bitstring
--------------
11110000
(1 row)
```

See Also

BITCOUNT (page 359) and ***BITSTRING_TO_BINARY*** (page 360)

TO_HEX

Returns a VARCHAR or VARBINARY representing the hexadecimal equivalent of a number.

Behavior Type

Immutable

Syntax

```
TO_HEX ( number )
```

Parameters

number	(INTEGER) is the number to convert to hexadecimal

Notes

VARCHAR TO_HEX(INTEGER) and VARCHAR TO_HEX(VARBINARY) are similar. The function converts data from binary type to character type (where the character representation is in hexadecimal format). This function is the inverse of HEX_TO_BINARY.

```
TO_HEX(HEX_TO_BINARY(x)) = x).
HEX_TO_BINARY(TO_HEX(x)) = x).
```

Examples

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

For VARBINARY inputs, the returned value is not preceded by "0x". For example:

```
SELECT TO_HEX('ab'::binary(2));
 to_hex
--------
 6162
(1 row)
```

TRANSLATE

Replaces individual characters in *string_to_replace* with other characters.

Behavior Type

Immutable

Syntax

```
TRANSLATE ( string_to_replace , from_string , to_string );
```

Parameters

string_to_replace	Is the string to be translated.
from_string	Contains characters that should be replaced in string_to_replace.
to_string	Any character in string_to_replace that matches a character in from_string is replaced by the corresponding character in to_string.

Example

```
SELECT TRANSLATE('straße', 'ß', 'ss');
 TRANSLATE
-----------
 strase
(1 row)
```

TRIM

Combines the BTRIM, LTRIM, and RTRIM functions into a single function.

Behavior Type

Immutable

Syntax

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

Parameters

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

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

See Also

BTRIM (page 360)

LTRIM (page 384)

RTRIM (page 396)

UPPER

Returns a VARCHAR value containing the argument converted to uppercase letters. Starting in Release 5.1, this function treats the `string` argument as a UTF-8 encoded string, rather than depending on the collation setting of the locale (for example, collation=binary) to identify the encoding. Prior to Release 5.1, the behavior type of this function was stable.

Behavior Type

Immutable

Syntax

```
UPPER ( expression )
```

Parameters

expression	(CHAR or VARCHAR) is the string to convert

Notes

UPPER is restricted to 32750 octet inputs, since it is possible for the UTF-8 representation of result to double in size.

Examples

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

dbadmin=> SELECT UPPER('étudiant');
  UPPER
----------
 ÉTUDIANT
(1 row)
```

UPPERB

Returns a character string with each ASCII character converted to uppercase. Multibyte characters are not converted and are skipped.

Behavior Type

Immutable

Syntax

```
UPPERB ( expression )
```

Parameters

expression	(CHAR or VARCHAR) is the string to convert

Examples

In the following example, the multibyte UTF-8 character é is not converted to uppercase:

```
SELECT UPPERB('étudiant');
  UPPERB
----------
```

```
 éTUDIANT
(1 row)

SELECT UPPERB('AbCdEfG');
 UPPERB
---------
 ABCDEFG
(1 row)

SELECT UPPERB('The Vertica Database');
        UPPERB
----------------------
 THE VERTICA DATABASE
(1 row)
```

V6_ATON

Converts an IPv6 address represented as a character string to a binary string.

Behavior Type

Immutable

Syntax

V6_ATON (*expression*)

Parameters

expression	(VARCHAR) is the string to convert.

Notes

The following syntax converts an IPv6 address represented as the character string A to a binary string B.

V6_ATON trims any spaces from the right of A and calls the Linux function ***inet_pton http://www.opengroup.org/onlinepubs/000095399/functions/inet_ntop.html***.

```
V6_ATON(VARCHAR A) -> VARBINARY(16) B
```

If A has no colons it is prepended with '::ffff:'. If A is NULL, too long, or if inet_pton returns an error, the result is NULL.

Examples

```
SELECT V6_ATON('2001:DB8::8:800:200C:417A');
                      v6_aton
-----------------------------------------------------
  \001\015\270\000\000\000\000\000\010\010\000 \014Az
(1 row)
SELECT TO_HEX(V6_ATON('2001:DB8::8:800:200C:417A'));
              to_hex
---------------------------------
```

```
 20010db8000000000080800200c417a
(1 row)
SELECT V6_ATON('1.2.3.4');
                              v6_aton
-----------------------------------------------------------------
--

\000\000\000\000\000\000\000\000\000\000\377\377\001\002\003\004
(1 row)
SELECT V6_ATON('::1.2.3.4');
                              v6_aton
-----------------------------------------------------------------
--

\000\000\000\000\000\000\000\000\000\000\000\000\001\002\003\004
(1 row)
```

See Also

V6_NTOA (page 295)

V6_NTOA

Converts an IPv6 address represented as varbinary to a character string.

Behavior Type

Immutable

Syntax

```
V6_NTOA ( expression )
```

Parameters

expression	(VARBINARY) is the binary string to convert.

Notes

The following syntax converts an IPv6 address represented as VARBINARY B to a string A.

V6_NTOA right-pads B to 16 bytes with zeros, if necessary, and calls the Linux function ***inet_ntop http://www.opengroup.org/onlinepubs/000095399/functions/inet_ntop.html***.

```
V6_NTOA(VARBINARY B) -> VARCHAR A
```

If B is NULL or longer than 16 bytes, the result is NULL.

HP Vertica automatically converts the form '::ffff:1.2.3.4' to '1.2.3.4'.

Examples

```
SELECT V6_NTOA(' \001\015\270\000\000\000\000\000\010\010\000 \014Az');
          v6_ntoa
```

```
---------------------------
 2001:db8::8:800:200c:417a
(1 row)
SELECT V6_NTOA(V6_ATON('1.2.3.4'));
 v6_ntoa
---------
 1.2.3.4
(1 row)
SELECT V6_NTOA(V6_ATON('::1.2.3.4'));
  v6_ntoa
-----------
 ::1.2.3.4
(1 row)
```

See Also

N6_ATON (page 294)

V6_SUBNETA

Calculates a subnet address in CIDR (Classless Inter-Domain Routing) format from a binary or alphanumeric IPv6 address.

Behavior Type

Immutable

Syntax

```
V6_SUBNETA ( expression1, expression2 )
```

Parameters

expression1	(VARBINARY or VARCHAR) is the string to calculate.
expression2	(INTEGER) is the size of the subnet.

Notes

The following syntax calculates a subnet address in CIDR format from a binary or varchar IPv6 address.

V6_SUBNETA masks a binary IPv6 address B so that the N leftmost bits form a subnet address, while the remaining rightmost bits are cleared. It then converts to an alphanumeric IPv6 address, appending a slash and N.

```
V6_SUBNETA(BINARY B, INT8 N) -> VARCHAR C
```

The following syntax calculates a subnet address in CIDR format from an alphanumeric IPv6 address.

```
V6_SUBNETA(VARCHAR A, INT8 N) -> V6_SUBNETA(V6_ATON(A), N) -> VARCHAR C
```

Examples

```
SELECT V6_SUBNETA(V6_ATON('2001:db8::8:800:200c:417a'), 28);
  v6_subneta
---------------
 2001:db0::/28
(1 row)
```

See Also

V6_SUBNETN (page 297)

V6_SUBNETN

Calculates a subnet address in CIDR (Classless Inter-Domain Routing) format from a varbinary or alphanumeric IPv6 address.

Behavior Type

Immutable

Syntax

```
V6_SUBNETN ( expression1, expression2 )
```

Parameters

expression1	(VARBINARY or VARCHAR) is the string to calculate. **Notes:** ▪ V6_SUBNETN(<*VARBINARY*>, <*INTEGER*>) returns varbinary. OR ▪ V6_SUBNETN(<*VARCHAR*>, <*INTEGER*>) returns varbinary, after using V6_ATON to convert the <*VARCHAR*> string to <*VARBINARY*>.
expression2	(INTEGER) is the size of the subnet.

Notes

The following syntax masks a BINARY IPv6 address **B** so that the N left-most bits of **S** form a subnet address, while the remaining right-most bits are cleared.

V6_SUBNETN right-pads B to 16 bytes with zeros, if necessary and masks B, preserving its N-bit subnet prefix.

```
V6_SUBNETN(VARBINARY B, INT8 N) -> VARBINARY(16) S
```

If B is NULL or longer than 16 bytes, or if N is not between 0 and 128 inclusive, the result is NULL.

S = [B]/N in ***Classless Inter-Domain Routing http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing*** notation (CIDR notation).

The following syntax masks an alphanumeric IPv6 address **A** so that the N leftmost bits form a subnet address, while the remaining rightmost bits are cleared.

```
V6_SUBNETN(VARCHAR A, INT8 N) -> V6_SUBNETN(V6_ATON(A), N) -> VARBINARY(16) S
```

Example

This example returns VARBINARY, after using V6_ATON to convert the VARCHAR string to VARBINARY:

```
=> SELECT V6_SUBNETN(V6_ATON('2001:db8::8:800:200c:417a'), 28);
                          v6_subnetn
-----------------------------------------------------------------
  \001\015\260\000\000\000\000\000\000\000\000\000\000\000\000
```

See Also

V6_ATON (page 294)

V6_SUBNETA (page 296)

V6_TYPE

Characterizes a binary or alphanumeric IPv6 address B as an integer type.

Behavior Type

Immutable

Syntax

```
V6_TYPE ( expression )
```

Parameters

expression	(VARBINARY or VARCHAR) is the type to convert.

Notes

V6_TYPE(VARBINARY B) returns INT8 T.

```
V6_TYPE(VARCHAR A) -> V6_TYPE(V6_ATON(A)) -> INT8 T
```

The IPv6 types are defined in the Network Working Group's ***IP Version 6 Addressing Architecture memo http://www.ietf.org/rfc/rfc4291***.

```
  GLOBAL = 0       Global unicast addresses
  LINKLOCAL = 1    Link-Local unicast (and Private-Use) addresses
  LOOPBACK = 2     Loopback
  UNSPECIFIED = 3  Unspecified
  MULTICAST = 4    Multicast
```

IPv4-mapped and IPv4-compatible IPv6 addresses are also interpreted, as specified in ***IPv4 Global Unicast Address Assignments http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml***.

- For IPv4, Private-Use is grouped with Link-Local.
- If B is VARBINARY, it is right-padded to 16 bytes with zeros, if necessary.
- If B is NULL or longer than 16 bytes, the result is NULL.

Details

IPv4 (either kind):

```
0.0.0.0/8         UNSPECIFIED
10.0.0.0/8        LINKLOCAL
127.0.0.0/8       LOOPBACK
169.254.0.0/16    LINKLOCAL
172.16.0.0/12     LINKLOCAL
192.168.0.0/16    LINKLOCAL
224.0.0.0/4       MULTICAST
others            GLOBAL
```

IPv6:

```
::0/128           UNSPECIFIED
::1/128           LOOPBACK
fe80::/10         LINKLOCAL
ff00::/8          MULTICAST
others            GLOBAL
```

Examples

```
SELECT V6_TYPE(V6_ATON('192.168.2.10'));
 v6_type
---------
       1
(1 row)
SELECT V6_TYPE(V6_ATON('2001:db8::8:800:200c:417a'));
 v6_type
---------
       0
(1 row)
```

See Also

INET_ATON (page 292)

IP Version 6 Addressing Architecture http://www.ietf.org/rfc/rfc4291

IPv4 Global Unicast Address Assignments http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml

System Information Functions

These functions provide system information regarding user sessions. A superuser has unrestricted access to all system information, but users can view only information about their own, current sessions.

CURRENT_DATABASE

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

Behavior Type

Immutable

Syntax

```
CURRENT_DATABASE()
```

Notes

- The parentheses following the CURRENT_DATABASE function are optional.
- This function is equivalent to DBNAME.

Examples

```
SELECT CURRENT_DATABASE();
 current_database
------------------
 VMart
(1 row)
```

The following command returns the same results without the parentheses:

```
SELECT CURRENT_DATABASE;
 current_database
------------------
 VMart
(1 row)
```

CURRENT_SCHEMA

Returns the name of the current schema.

Behavior Type

Stable

Syntax

```
CURRENT_SCHEMA()
```

Privileges

None

Notes

The CURRENT_SCHEMA function does not require parentheses.

Example

```
=> SELECT CURRENT_SCHEMA();
```

```
 current_schema
----------------
 public
(1 row)
```

The following command returns the same results without the parentheses:

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

The following command shows the current schema, listed after the ***current user*** (page 416), in the search path:

```
=> SHOW SEARCH_PATH;
    name     |                       setting
-------------+-------------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

See Also

SET SEARCH_PATH

CURRENT_USER

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

Behavior Type

Stable

Syntax

```
CURRENT_USER()
```

Notes

- The CURRENT_USER function does not require parentheses.
- This function is useful for permission checking.
- Is equivalent to ***SESSION_USER*** (page 419), ***USER*** (page 419), and ***USERNAME*** (page 420).

Examples

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

The following command returns the same results without the parentheses:

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

DBNAME (function)

Returns a VARCHAR value containing the name of the database to which you are connected. DBNAME is equivalent to ***CURRENT_DATABASE*** (page 415).

Behavior Type

Immutable

Syntax

```
DBNAME()
```

Examples

```
SELECT DBNAME();
 dbname
------------------
 VMart
(1 row)
```

HAS_TABLE_PRIVILEGE

Indicates whether a user can access a table in a particular way. The function returns a true (t) or false (f) value.

A superuser can check all other user's table privileges.

Users without superuser privileges can use HAS_TABLE_PRIVILEGE to check:

- Any tables they own.
- Tables in a schema to which they have been granted USAGE privileges, and at least one other table privilege, as described in ***GRANT (Table)*** (page 842).

Behavior Type

Stable

Syntax

```
HAS_TABLE_PRIVILEGE ( [ user, ] [[db-name.]schema-name.]table , privilege )
```

Parameters

user	Specifies the name or OID of a database user. The default is the ***CURRENT_USER*** (page 416).

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the name or OID of a table in the logical schema. If necessary, specify the database and schema, as noted above.
privilege	▪ `SELECT` Allows the user to SELECT from any column of the specified table. ▪ `INSERT` Allows the user to INSERT records into the specified table and to use the ***COPY*** (page 699) command to load the table. ▪ `UPDATE` Allows the user to UPDATE records in the specified table. ▪ `DELETE` Allows the user to delete a row from the specified table. ▪ `REFERENCES` Allows the user to create a foreign key constraint (privileges required on both the referencing and referenced tables).

Examples

```
SELECT HAS_TABLE_PRIVILEGE('store.store_dimension', 'SELECT');
 has_table_privilege
---------------------
 t
(1 row)
SELECT HAS_TABLE_PRIVILEGE('release', 'store.store_dimension',
'INSERT');
 has_table_privilege
---------------------
 t
(1 row)
SELECT HAS_TABLE_PRIVILEGE('store.store_dimension', 'UPDATE');
 has_table_privilege
---------------------
 t
(1 row)
SELECT HAS_TABLE_PRIVILEGE('store.store_dimension', 'REFERENCES');
 has_table_privilege
---------------------
 t
(1 row)
SELECT HAS_TABLE_PRIVILEGE(45035996273711159, 45035996273711160,
'select');
 has_table_privilege
---------------------
t
```

```
(1 row)
```

SESSION_USER

Returns a VARCHAR containing the name of the user who initiated the current database session.

Behavior Type

Stable

Syntax

```
SESSION_USER()
```

Notes

- The SESSION_USER function does not require parentheses.
- Is equivalent to ***CURRENT_USER*** (page 416), ***USER*** (page 419), and ***USERNAME*** (page 420).

Examples

```
SELECT SESSION_USER();
 session_user
--------------
 dbadmin
(1 row)
```

The following command returns the same results without the parentheses:

```
SELECT SESSION_USER;
 session_user
--------------
 dbadmin
(1 row)
```

USER

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

Behavior Type

Stable

Syntax

```
USER()
```

Notes

- The USER function does not require parentheses.
- Is equivalent to ***CURRENT_USER*** (page 416), ***SESSION_USER*** (page 419), and ***USERNAME*** (page 420).

Examples

```
SELECT USER();
 current_user
--------------
 dbadmin
(1 row)
```

The following command returns the same results without the parentheses:

```
SELECT USER;
 current_user
--------------
 dbadmin
(1 row)
```

USERNAME

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

Behavior Type

Stable

Syntax

```
USERNAME()
```

Notes

- This function is useful for permission checking.
- It is equivalent to ***CURRENT_USER*** (page 416), ***SESSION_USER*** (page 419) and ***USER*** (page 419).

Examples

```
SELECT USERNAME();
 username
--------------
 dbadmin
(1 row)
```

VERSION

Returns a VARCHAR containing an HP Vertica node's version information.

Behavior Type

Stable

Syntax

```
VERSION()
```

Examples

```
SELECT VERSION();
                      VERSION
-------------------------------------------------
 Vertica Analytic Database v4.0.12-20100513010203
(1 row)
```

The parentheses are required. If you omit them, the system returns an error:

```
SELECT VERSION;
ERROR:  column "version" does not exist
```

Timeseries Functions

Timeseries aggregate functions evaluate the values of a given set of variables over time and group those values into a window for analysis and aggregation.

One output row is produced per time slice—or per partition per time slice—if partition expressions are present.

See Also

TIMESERIES Clause (page 894)

CONDITIONAL_CHANGE_EVENT (page 151) and ***CONDITIONAL_TRUE_EVENT*** (page 152)

Using Time Series Analytics in the Programmer's Guide

TS_FIRST_VALUE

Processes the data that belongs to each time slice. A time series aggregate (TSA) function, TS_FIRST_VALUE returns the value at the start of the time slice, where an interpolation scheme is applied if the timeslice is missing, in which case the value is determined by the values corresponding to the previous (and next) timeslices based on the interpolation scheme of const (linear). There is one value per time slice per partition.

Behavior Type

Immutable

Syntax

```
TS_FIRST_VALUE ( expression [ IGNORE NULLS ]
... [, { 'CONST' | 'LINEAR' } ] )
```

Parameters

expression	Is the argument expression on which to aggregate and interpolate. *expression* is data type INTEGER or FLOAT.

IGNORE NULLS	The IGNORE NULLS behavior changes depending on a CONST or LINEAR interpolation scheme. See When Time Series Data Contains Nulls in the Programmer's Guide for details.
'CONST' | 'LINEAR'	(default CONST) Optionally specifies the interpolation value as either constant or linear. ▪ CONST—New value are interpolated based on previous input records. ▪ LINEAR—Values are interpolated in a linear slope based on the specified time slice.

Notes

- The function returns one output row per time slice or one output row per partition per time slice if partition expressions are specified.
- Multiple time series aggregate functions can exists in the same query. They share the same gap-filling policy as defined by the ***TIMESERIES clause*** (page 894); however, each time series aggregate function can specify its own interpolation policy. For example:

```
SELECT slice_time, symbol,
       TS_FIRST_VALUE(bid, 'const') fv_c,
       TS_FIRST_VALUE(bid, 'linear') fv_l,
       TS_LAST_VALUE(bid, 'const') lv_c
FROM TickStore
TIMESERIES slice_time AS '3 seconds' OVER(PARTITION BY symbol ORDER BY ts);
```

- You must use an ORDER BY clause with a timestamp column.

Example

For detailed examples, see Gap Filling and Interpolation in the Programmer's Guide.

See Also

TIMESERIES Clause (page 894) and ***TS_LAST_VALUE*** (page 422)

Using Time Series Analytics in the Programmer's Guide

TS_LAST_VALUE

Processes the data that belongs to each time slice. A time series aggregate (TSA) function, TS_LAST_VALUE returns the value at the end of the time slice, where an interpolation scheme is applied if the timeslice is missing, in which case the value is determined by the values corresponding to the previous (and next) timeslices based on the interpolation scheme of const (linear). There is one value per time slice per partition.

Behavior Type

Immutable

Syntax

```
TS_LAST_VALUE ( expression [ IGNORE NULLS ]
... [, { 'CONST' | 'LINEAR' } ] )
```

Parameters

expression	Is the argument expression on which to aggregate and interpolate. *expression* is data type INTEGER or FLOAT.
IGNORE NULLS	The IGNORE NULLS behavior changes depending on a CONST or LINEAR interpolation scheme. See When Time Series Data Contains Nulls in the Programmer's Guide for details.
'CONST' | 'LINEAR'	(default CONST) Optionally specifies the interpolation value as either constant or linear. ▪ CONST—New value are interpolated based on previous input records. ▪ LINEAR—Values are interpolated in a linear slope based on the specified time slice.

Notes

- The function returns one output row per time slice or one output row per partition per time slice if partition expressions are specified.
- Multiple time series aggregate functions can exists in the same query. They share the same gap-filling policy as defined by the ***TIMESERIES clause*** (page 894); however, each time series aggregate function can specify its own interpolation policy. For example:

```
SELECT slice_time, symbol,
       TS_FIRST_VALUE(bid, 'const') fv_c,
       TS_FIRST_VALUE(bid, 'linear') fv_l,
       TS_LAST_VALUE(bid, 'const') lv_c
FROM TickStore
TIMESERIES slice_time AS 3 seconds OVER(PARTITION BY symbol ORDER BY ts);
```

- You must use the ORDER BY clause with a TIMESTAMP column.

Example

For detailed examples, see Gap Filling and Interpolation in the Programmer's Guide.

See Also

TIMESERIES Clause (page 894) and ***TS_FIRST_VALUE*** (page 421)

Using Time Series Analytics in the Programmer's Guide

URI Encode/Decode Functions

The functions in this section follow the RFC 3986 standard for percent-encoding a Universal Resource Identifier (URI).

URI_PERCENT_DECODE

Decodes a percent-encoded Universal Resource Identifier (URI) according to the RFC 3986 standard.

Syntax

```
URI_PERCENT_DECODE (expression)
```

Behavior type

Immutable

Parameters

expression	(VARCHAR) is the string to convert.

Examples

The following example invokes uri_percent_decode on the Websites column of the URI table and returns a decoded URI:

```
=> SELECT URI_PERCENT_DECODE(Websites) from URI;
              URI_PERCENT_DECODE
-----------------------------------------------
 http://www.faqs.org/rfcs/rfc3986.html x xj%a%

(1 row)
```

The following example returns the original URI in the Websites column and its decoded version:

```
=> SELECT Websites, URI_PERCENT_DECODE (Websites) from URI;
                     Websites                      |
URI_PERCENT_DECODE
---------------------------------------------------+-------------------------
-------------------
 http://www.faqs.org/rfcs/rfc3986.html+x%20x%6a%a% |
http://www.faqs.org/rfcs/rfc3986.html x xj%a%

(1 row)
```

URI_PERCENT_ENCODE

Encodes a Universal Resource Identifier (URI) according to the RFC 3986 standard for percent encoding. In addition, for compatibility with older encoders this function converts '+' to space; space is converted to %20 by uri_percent_encode.

Syntax

```
URI_PERCENT_ENCODE (expression)
```

Behavior type

Immutable

Parameters

expression	(VARCHAR) is the string to convert.

Example

The following example shows how the `uri_percent_encode` function is invoked on a the Websites column of the URI table and returns an encoded URI:

```
=> SELECT URI_PERCENT_ENCODE(Websites) from URI;
            URI_PERCENT_ENCODE
------------------------------------------
 http%3A%2F%2Fexample.com%2F%3F%3D11%2F15
(1 row)
```

The following example returns the original URI in the Websites column and it's encoded form:

```
=> SELECT Websites, URI_PERCENT_ENCODE(Websites) from URI;
          Websites           |            URI_PERCENT_ENCODE
-----------------------------+------------------------------------------
 http://example.com/?=11/15 | http%3A%2F%2Fexample.com%2F%3F%3D11%2F15
(1 row)
```

HP Vertica Meta-functions

HP Vertica built-in (meta) functions access the internal state of HP Vertica and are used in SELECT queries with the function name and an argument (where required). These functions are not part of the SQL standard and take the following form:

```
SELECT <meta-function>(<args>);
```

Note: The query cannot contain other clauses, such as FROM or WHERE.

The behavior type of HP Vertica meta-functions is immutable.

Alphabetical List of HP Vertica Meta-functions

This section contains the HP Vertica meta-functions, listed alphabetically. These functions are also grouped into their appropriate category.

ADD_LOCATION

Adds a storage location to the cluster. Use this function to add a new location, optionally with a location label. You can also add a location specifically for user access, and then grant one or more users access to the location.

Syntax

```
ADD_LOCATION ( 'path' [, 'node' , 'usage', 'location_label' ] )
```

Parameters

path	[Required] Specifies where the storage location is mounted. Path must be an empty directory with write permissions for user, group, or all.
node	[Optional] Indicates the cluster node on which a storage location resides. If you omit this parameter, the function adds the location to only the initiator node. Specifying the *node* parameter as an empty string (`''`) adds a storage location to all cluster nodes in a single transaction. **NOTE:** If you specify a node, you must also add a usage parameter.
usage	[Optional] Specifies what the storage location will be used for: ▪ DATA: Stores only data files. Use this option for labeled storage locations. ▪ TEMP: Stores only temporary files, created during loads or queries. ▪ DATA,TEMP: Stores both types of files in the location. ▪ USER: Allows non-dbadmin users access to the storage location for data files (not temp files), once they are granted privileges. DO NOT create a storage location for later use in a storage policy. Storage locations with policies must be for DATA usage. Also, note that this keyword is orthogonal to DATA and TEMP, and does not specify a particular usage, other than being accessible to non-dbadmin users with assigned privileges. You cannot alter a storage location to or from USER usage. **NOTE:** You can use this parameter only in conjunction with the node option. If you omit the usage parameter, the default is DATA,TEMP.

location_label	[Optional] Specifies a location label as a string, for example, *SSD*. Labeling a storage location lets you use the location label to create storage policies and as part of a multi-tenanted storage scheme.

Privileges

Must be a superuser

Storage Location Subdirectories

You cannot create a storage location in a subdirectory of an existing location. For example, if you create a storage location at one location, you cannot add a second storage location in a subdirectory of the first:

```
dbt=> select add_location ('/myvertica/Test/KMM','','DATA','SSD');
               add_location
-----------------------------------------
 /myvertica/Test/KMM added.
(1 row)
dbt=> select add_location ('/myvertica/Test/KMM/SSD','','DATA','SSD');
ERROR 5615:  Location [/myvertica/Test/KMM/SSD] conflicts with existing location
[/myvertica/Test/KMM] on node v_node0001
ERROR 5615:  Location [/myvertica/Test/KMM/SSD] conflicts with existing location
[/myvertica/Test/KMM] on node v_node0002
ERROR 5615:  Location [/myvertica/Test/KMM/SSD] conflicts with existing location
[/myvertica/Test/KMM] on node v_node0003
```

Example

This example adds a location that stores data and temporary files on the initiator node:

```
=> SELECT ADD_LOCATION('/secondverticaStorageLocation/');
```

This example adds a location to store data on `v_vmartdb_node0004`:

```
=> SELECT ADD_LOCATION('/secondverticaStorageLocation/' , 'v_vmartdb_node0004' ,
'DATA');
```

This example adds a new `DATA` storage location with a label, *SSD*. The label identifies the location when you create storage policies. Specifying the *node* parameter as an empty string adds the storage location to all cluster nodes in a single transaction:

```
VMART=> select add_location ('home/dbadmin/SSD/schemas', '', 'DATA', 'SSD');
          add_location
--------------------------------
 home/dbadmin/SSD/schemas added.
(1 row)
```

See Also

Adding Storage Locations in the Administrator's Guide

ALTER_LOCATION_USE (page 429)

DROP_LOCATION (page 472)

RESTORE_LOCATION (page 525)

RETIRE_LOCATION (page 526)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

ADVANCE_EPOCH

Manually closes the current epoch and begins a new epoch.

Syntax

```
ADVANCE_EPOCH ( [ integer ] )
```

Parameters

integer	Specifies the number of epochs to advance.

Privileges

Must be a superuser

Note

This function is primarily maintained for backward compatibility with earlier versions of HP Vertica.

Example

The following command increments the epoch number by 1:

```
=> SELECT ADVANCE_EPOCH(1);
```

See Also

ALTER PROJECTION (page 659)

ALTER_LOCATION_USE

Alters the type of files that can be stored at the specified storage location.

Syntax

```
ALTER_LOCATION_USE ( 'path' , [ 'node' ] , 'usage' )
```

Parameters

path	Specifies where the storage location is mounted.
node	[Optional] The HP Vertica node with the storage location. Specifying the *node* parameter as an empty string (`''`) alters the location across all cluster nodes in a single transaction. If you omit this parameter, *node* defaults to the initiator.

usage	Is one of the following: ▪ DATA: The storage location stores only data files. This is the supported use for both a USER storage location, and a labeled storage location. ▪ TEMP: The location stores only temporary files that are created during loads or queries. ▪ DATA,TEMP: The location can store both types of files.

Privileges

Must be a superuser

USER Storage Location Restrictions

You cannot change a storage location from a USER usage type if you created the location that way, or to a USER type if you did not. You can change a USER storage location to specify DATA (storing TEMP files is not supported). However, doing so does not affect the primary objective of a USER storage location, to be accessible by non-dbadmin users with assigned privileges.

Monitoring Storage Locations

Disk storage information that the database uses on each node is available in the ***V_MONITOR.DISK_STORAGE*** (page 1014) system table.

Example

The following example alters the storage location across all cluster nodes to store only data:

```
=> SELECT ALTER_LOCATION_USE ('/thirdVerticaStorageLocation/' , '' , 'DATA');
```

See Also

Altering Storage Locations in the Administrator's Guide

ADD_LOCATION (page 426)

DROP_LOCATION (page 472)

RESTORE_LOCATION (page 525)

RETIRE_LOCATION (page 526)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

ALTER_LOCATION_LABEL

Alters the location label. Use this function to add, change, or remove a location label. You change a location label only if it is not currently in use as part of a storage policy.

You can use this function to remove a location label. However, you cannot remove a location label if the name being removed is used in a storage policy, *and* the location from which you are removing the label is the last available storage for its associated objects.

NOTE: If you label an existing storage location that already contains data, and then include the labeled location in one or more storage policies, existing data could be moved. If the ATM determines data stored on a labeled location does not comply with a storage policy, the ATM moves the data elsewhere.

Syntax

```
ALTER_LOCATION_LABEL ( 'path' , 'node' , 'location_label' )
```

Parameters

path	Specifies the path of the storage location.
node	The HP Vertica node for the storage location. If you enter node as an empty string (''), the function performs a cluster-wide label change to all nodes. Any node that is unavailable generates an error.
location_label	Specifies a storage label as a string, for instance *SSD*. You can change an existing label assigned to a storage location, or add a new label. Specifying an empty string (") removes an existing label.

Privileges

Must be a superuser

Example

The following example alters (or adds) the label `SSD` to the storage location at the given path on all cluster nodes:

```
VMART=> select alter_location_label('/home/dbadmin/SSD/tables','', 'SSD');
          alter_location_label
--------------------------------------
 /home/dbadmin/SSD/tables label changed.
(1 row)
```

See Also

Altering Location Labels in the Administrator's Guide

CLEAR_OBJECT_STORAGE_POLICY (page 457)

SET_OBJECT_STORAGE_POLICY (page 534)

ANALYZE_CONSTRAINTS

Analyzes and reports on constraint violations within the current schema search path, or external to that path if you specify a database name (noted in the syntax statement and parameter table).

You can check for constraint violations by passing arguments to the function as follows:

1. An empty argument (' '), which returns violations on all tables within the current schema
2. One argument, referencing a table
3. Two arguments, referencing a table name and a column or list of columns

Syntax

```
ANALYZE_CONSTRAINTS [ ( '' )
... | ( '[[db-name.]schema.]table [.column_name]' )
... | ( '[[db-name.]schema.]table' ,  'column' ) ]
```

Parameters

`('')`	Analyzes and reports on all tables within the current schema search path.
`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
`table`	Analyzes and reports on all constraints referring to the specified table.
`column`	Analyzes and reports on all constraints referring to the specified table that contains the column.

Privileges

- SELECT privilege on table
- USAGE privilege on schema

Notes

ANALYZE_CONSTRAINTS() performs a lock in the same way that `SELECT * FROM t1` holds a lock on table `t1`. See ***LOCKS*** (page 1037) for additional information.

Detecting Constraint Violations During a Load Process

HP Vertica checks for constraint violations when queries are run, not when data is loaded. To detect constraint violations as part of the load process, use a ***COPY*** (page 699) statement with the NO COMMIT option. By loading data without committing it, you can run a post-load check of your data using the ANALYZE_CONSTRAINTS function. If the function finds constraint violations, you can roll back the load because you have not committed it.

If ANALYZE_CONSTRAINTS finds violations, such as when you insert a duplicate value into a primary key, you can correct errors using the following functions. Effects last until the end of the session only:

- SELECT ***DISABLE_DUPLICATE_KEY_ERROR*** (page 466)
- SELECT ***REENABLE_DUPLICATE_KEY_ERROR*** (page 522)

Return Values

ANALYZE_CONSTRAINTS returns results in a structured set (see table below) that lists the schema name, table name, column name, constraint name, constraint type, and the column values that caused the violation.

If the result set is empty, then no constraint violations exist; for example:

```
=> SELECT ANALYZE_CONSTRAINTS ('public.product_dimension', 'product_key');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
(0 rows)
```

The following result set shows a primary key violation, along with the value that caused the violation `('10')`:

```
=> SELECT ANALYZE_CONSTRAINTS ('');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 store         t1           c1             pk_t1             PRIMARY           ('10')
(1 row)
```

The result set columns are described in further detail in the following table:

Column Name	Data Type	Description
`Schema Name`	VARCHAR	The name of the schema.
`Table Name`	VARCHAR	The name of the table, if specified.

`Column Names`	VARCHAR	Names of columns containing constraints. Multiple columns are in a comma-separated list:
		`store_key,`
		`store_key, date_key,`

`Constraint Name`	VARCHAR	The given name of the primary key, foreign key, unique, or not null constraint, if specified.
`Constraint Type`	VARCHAR	Identified by one of the following strings: 'PRIMARY KEY', 'FOREIGN KEY', 'UNIQUE', or 'NOT NULL'.

`Column Values`	VARCHAR	Value of the constraint column, in the same order in which `Column Names` contains the value of that column in the violating row.
		When interpreted as SQL, the value of this column forms a list of values of the same type as the columns in `Column Names`; for example:
		`('1'),`
		`('1', 'z')`

Understanding Function Failures

If ANALYZE_CONSTRAINTS() fails, HP Vertica returns an error identifying the failure condition. For example, if there are insufficient resources, the database cannot perform constraint checks and ANALYZE_CONSTRAINTS() fails.

If you specify the wrong table, the system returns an error message:

```
=> SELECT ANALYZE_CONSTRAINTS('abc');
   ERROR 2069: 'abc' is not a table in the current search_path
```

If you issue the function with incorrect syntax, the system returns an error message with a hint:

```
ANALYZE ALL CONSTRAINT;
```

Or

```
ANALYZE CONSTRAINT abc;
ERROR: ANALYZE CONSTRAINT is not supported.
HINT: You may consider using analyze_constraints().
```

If you run ANALYZE_CONSTRAINTS from a non-default locale, the function returns an error with a hint:

```
=> \locale LEN
INFO 2567:  Canonical locale: 'en'
Standard collation: 'LEN'
English
=> SELECT ANALYZE_CONSTRAINTS('t1');
ERROR:  ANALYZE_CONSTRAINTS is currently not supported in non-default
locales
HINT:  Set the locale in this session to en_US@collation=binary using
the
command "\locale en_US@collation=binary"
```

Examples

Given the following inputs, HP Vertica returns one row, indicating one violation, because the same primary key value (10) was inserted into table t1 twice:

```
=> CREATE TABLE t1(c1 INT);
=> ALTER TABLE t1 ADD CONSTRAINT pk_t1 PRIMARY KEY (c1);
=> CREATE PROJECTION t1_p (c1) AS SELECT * FROM t1 UNSEGMENTED ALL NODES;
=> INSERT INTO t1 values (10);
=> INSERT INTO t1 values (10); --Duplicate primary key value
=> SELECT ANALYZE_CONSTRAINTS('t1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      t1           c1             pk_t1             PRIMARY           ('10')
(1 row)
```

If the second INSERT statement above had contained any different value, the result would have been 0 rows (no violations).

In the following example, create a table that contains three integer columns, one a unique key and one a primary key:

```
=> CREATE TABLE fact_1(
     f INTEGER,
```

```
    f_UK INTEGER UNIQUE,
    f_PK INTEGER PRIMARY KEY
  );
```

Issue a command that refers to a nonexistent table and column:

```
=> SELECT ANALYZE_CONSTRAINTS('f_BB');
   ERROR: 'f_BB' is not a table name in the current search path
```

Issue a command that refers to a nonexistent column:

```
=> SELECT ANALYZE_CONSTRAINTS('fact_1','x');
   ERROR 41614:  Nonexistent columns: 'x '
```

Insert some values into table `fact_1` and commit the changes:

```
=> INSERT INTO fact_1 values (1, 1, 1);
=> COMMIT;
```

Run ANALYZE_CONSTRAINTS on table `fact_1`. No constraint violations are reported:

```
=> SELECT ANALYZE_CONSTRAINTS('fact_1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
(0 rows)
```

Insert duplicate unique and primary key values and run ANALYZE_CONSTRAINTS on table `fact_1` again. The system shows two violations: one against the primary key and one against the unique key:

```
=> INSERT INTO fact_1 VALUES (1, 1, 1);
=> COMMIT;
=> SELECT ANALYZE_CONSTRAINTS('fact_1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | fact_1     | f_pk         | -               | PRIMARY         | ('1')
 public      | fact_1     | f_uk         | -               | UNIQUE          | ('1')
(2 rows)
```

The following command looks for constraint validations on only the unique key in the table `fact_1`, qualified with its schema name:

```
=> SELECT ANALYZE_CONSTRAINTS('public.fact_1', 'f_UK');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | fact_1     | f_uk         | C_UNIQUE        | UNIQUE          | ('1')
(1 row)
```

The following example shows that you can specify the same column more than once; ANALYZE_CONSTRAINTS, however, returns the violation only once:

```
=> SELECT ANALYZE_CONSTRAINTS('fact_1', 'f_PK, F_PK');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | fact_1     | f_pk         | C_PRIMARY       | PRIMARY         | ('1')
(1 row)
```

The following example creates a new dimension table, `dim_1`, and inserts a foreign key and different (character) data types:

```
=> CREATE TABLE dim_1 (b VARCHAR(3), b_PK VARCHAR(4), b_FK INTEGER REFERENCES fact_1(f_PK));
```

Alter the table to create a multicolumn unique key and multicolumn foreign key and create superprojections:

```
=> ALTER TABLE dim_1 ADD CONSTRAINT dim_1_multiuk PRIMARY KEY (b, b_PK);
```

The following command inserts a missing foreign key (0) into table dim_1 and commits the changes:

```
=> INSERT INTO dim_1 VALUES ('r1', 'Xpk1', 0);
=> COMMIT;
```

Checking for constraints on the table dim_1 in the public schema detects a foreign key violation:

```
=> SELECT ANALYZE_CONSTRAINTS('public.dim_1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | dim_1      | b_fk         | C_FOREIGN       | FOREIGN         | ('0')
(1 row)
```

Now add a duplicate value into the unique key and commit the changes:

```
=> INSERT INTO dim_1 values ('r2', 'Xpk1', 1);
=> INSERT INTO dim_1 values ('r1', 'Xpk1', 1);
=> COMMIT;
```

Checking for constraint violations on table dim_1 detects the duplicate unique key error:

```
=> SELECT ANALYZE_CONSTRAINTS('dim_1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | dim_1      | b, b_pk      | dim_1_multiuk   | PRIMARY         | ('r1', 'Xpk1')
 public      | dim_1      | b_fk         | C_FOREIGN       | FOREIGN         | ('0')
(2 rows)
```

Create a table with multicolumn foreign key and create the superprojections:

```
=> CREATE TABLE dim_2(z_fk1 VARCHAR(3), z_fk2 VARCHAR(4));
=> ALTER TABLE dim_2 ADD CONSTRAINT dim_2_multifk FOREIGN KEY (z_fk1, z_fk2) REFERENCES dim_1(b, b_PK);
```

Insert a foreign key that matches a foreign key in table dim_1 and commit the changes:

```
=> INSERT INTO dim_2 VALUES ('r1', 'Xpk1');
=> COMMIT;
```

Checking for constraints on table dim_2 detects no violations:

```
=> SELECT ANALYZE_CONSTRAINTS('dim_2');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
(0 rows)
```

Add a value that does not match and commit the change:

```
=> INSERT INTO dim_2 values ('r1', 'NONE');
=> COMMIT;
```

Checking for constraints on table dim_2 detects a foreign key violation:

```
=> SELECT ANALYZE_CONSTRAINTS('dim_2');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | dim_2      | z_fk1, z_fk2 | dim_2_multifk   | FOREIGN         | ('r1', 'NONE')
(1 row)
```

Analyze all constraints on all tables:

```
=> SELECT ANALYZE_CONSTRAINTS('');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | dim_1      | b, b_pk      | dim_1_multiuk   | PRIMARY         | ('r1', 'Xpk1')
 public      | dim_1      | b_fk         | C_FOREIGN       | FOREIGN         | ('0')
 public      | dim_2      | z_fk1, z_fk2 | dim_2_multifk   | FOREIGN         | ('r1', 'NONE')
```

```
 public      | fact_1     | f_pk         | C_PRIMARY       | PRIMARY         | ('1')
 public      | fact_1     | f_uk         | C_UNIQUE        | UNIQUE          | ('1')
(5 rows)
```

To quickly clean up your database, issue the following command:

```
=> DROP TABLE fact_1 cascade;
=> DROP TABLE dim_1 cascade;
=> DROP TABLE dim_2 cascade;
```

To learn how to remove violating rows, see the ***DISABLE_DUPLICATE_KEY_ERROR*** (page 466) function.

See Also

Adding Constraints in the Administrator's Guide

COPY (page 699)

ALTER TABLE (page 672)

CREATE TABLE (page 770)

ANALYZE_HISTOGRAM

Collects and aggregates data samples and storage information from all nodes that store projections associated with the specified table or column.

If the function returns successfully (0), HP Vertica writes the returned statistics to the catalog. The query optimizer uses this collected data to recommend the best possible plan to execute a query. Without analyzing table statistics, the query optimizer would assume uniform distribution of data values and equal storage usage for all projections.

ANALYZE_HISTOGRAM is a DDL operation that auto-commits the current transaction, if any. The ANALYZE_HISTOGRAM function reads a variable amount of disk contents to aggregate sample data for statistical analysis. Use the function's *percent* float parameter to specify the total disk space from which HP Vertica collects sample data. The ***ANALYZE_STATISTICS*** (page 440) function returns similar data, but uses a fixed disk space amount (10 percent). Analyzing more than 10 percent disk space takes proportionally longer to process, but produces a higher level of sampling accuracy. ANALYZE_HISTOGRAM is supported on local temporary tables, but not on global temporary tables.

Syntax

```
ANALYZE_HISTOGRAM ('')
... | ( '[ [ db-name.]schema.]table [.column-name ]' [, percent ] )
```

Return value

0 - For success. If an error occurs, refer to `vertica.log` for details.

Parameters

`''`	Empty string. Collects statistics for all tables.
`[[db-name.]schema.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search

	path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the name of the table and collects statistics for all projections of that table. If you are using more than one schema, specify the schema that contains the projection, as noted in the [[*db-name*.]*schema*.] entry.
[.*column-name*]	[Optional] Specifies the name of a single column, typically a predicate column. Using this option with a table specification lets you collect statistics for only that column. **Note:** If you alter a table to add or drop a column, or add a new column to a table and populate its contents with either default or other values, HP Vertica recommends calling this function on the new table column to get the most current statistics.
percent	[Optional] Specifies what percentage of data to read from disk (not the amount of data to analyze). Specify a float from 1 – 100, such as 33.3. By default, the function reads 10% of the table data from disk. For more information, see Collecting Statistics in the Administrator's Guide.

Privileges

- Any INSERT/UPDATE/DELETE privilege on table
- USAGE privilege on schema that contains the table

Use the HP Vertica statistics functions as follows:

Use this function...	To obtain...
ANALYZE_STATISTICS (page 440)	A fixed-size statistical data sampling (10 percent per disk). This function returns results quickly, but is less accurate than using ANALYZE_HISTOGRAM to get a larger sampling of disk data.
ANALYZE_HISTOGRAM (page 437)	A specified percentage of disk data sampling (from 1 - 100). If you analyze more than 10 percent data per disk, this function is more accurate than ANALYZE_STATISTICS, but requires proportionately longer to return statistics.

Analyzing Results

To retrieve hints about under-performing queries and the associated root causes, use the ***ANALYZE_WORKLOAD*** (page 443) function. This function runs the Workload Analyzer and returns tuning recommendations, such as "`run analyze_statistics on schema.table.column`". You or your database administrator should act upon the tuning recommendations.

You can also find database tuning recommendations on the Management Console.

Canceling ANALYZE_HISTOGRAM

You can cancel this function mid-analysis by issuing CTRL-C in a vsql shell or by invoking the ***INTERRUPT_STATEMENT()*** (page 503) function.

Notes

By default, HP Vertica analyzes more than one column (subject to resource limits) in a single-query execution plan to:

- Reduce plan execution latency
- Help speed up analysis of relatively small tables that have a large number of columns

Examples

In this example, the ANALYZE_STATISTICS() function reads 10 percent of the disk data. This is the static default value for this function. The function returns 0 for success:

```
=> SELECT ANALYZE_STATISTICS('shipping_dimension.shipping_key');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

This example uses ANALYZE_HISTOGRAM () without specifying a percentage value. Since this function has a default value of 10 percent, it returns the identical data as the ANALYZE_STATISTICS() function, and returns 0 for success:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key');
 ANALYZE_HISTOGRAM
-------------------
                 0
(1 row)
```

This example uses ANALYZE_HISTOGRAM (), specifying its percent parameter as 100, indicating it will read the entire disk to gather data. After the function performs a full column scan, it returns 0 for success:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key', 100);
 ANALYZE_HISTOGRAM
-------------------
                 0
(1 row)
```

In this command, only 0.1% (1/1000) of the disk is read:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key', 0.1);
 ANALYZE_HISTOGRAM
-------------------
                 0
(1 row)
```

See Also

ANALYZE_STATISTICS (page 440)

ANALYZE_WORKLOAD (page 443)

DROP_STATISTICS (page 476)

EXPORT_STATISTICS (page 490)

IMPORT_STATISTICS (page 502)

INTERRUPT_STATEMENT (page 503)

Collecting Statistics in the Administrator's Guide

ANALYZE_STATISTICS

Collects and aggregates data samples and storage information from all nodes that store projections associated with the specified table or column.

If the function returns successfully (0), HP Vertica writes the returned statistics to the catalog. The query optimizer uses this collected data to recommend the best possible plan to execute a query. Without analyzing table statistics, the query optimizer would assume uniform distribution of data values and equal storage usage for all projections.

ANALYZE_STATISTICS is a DDL operation that auto-commits the current transaction, if any. The ANALYZE_STATISTICS function reads a fixed, 10 percent of disk contents to aggregate sample data for statistical analysis. To obtain a larger (or smaller) data sampling, use the ***ANALYZE_HISTOGRAM*** (page 437) function, which lets you specify the percent of disk to read. Analyzing more that 10 percent disk space takes proportionally longer to process, but results in a higher level of sampling accuracy. ***ANALYZE_STATISTICS*** (page 440) is supported on local temporary tables, but not on global temporary tables.

Syntax

```
ANALYZE_STATISTICS [ ('')
... | ( '[ [ db-name.]schema.]table [.column-name ]' ) ]
```

Return Value

0 - For success.

If an error occurs, refer to `vertica.log` for details.

Parameters

`''`	Empty string. Collects statistics for all tables.
`[[`*db-name*`.]`*schema*`.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the name of the table and collects statistics for all projections of that table. **Note:** If you are using more than one schema, specify the schema that contains the projection, as noted as noted in the [[*db-name*.]*schema*.] entry.
`[.`*column-name*`]`	[Optional] Specifies the name of a single column, typically a predicate column. Using this option with a table specification lets you collect statistics for only that column. **Note:** If you alter a table to add or drop a column, or add a new column to a table and populate its contents with either default or other values, HP Vertica recommends calling this function on the new table column to get the most current statistics.

Privileges

- Any INSERT/UPDATE/DELETE privilege on table
- USAGE privilege on schema that contains the table

Use the HP Vertica statistics functions as follows:

Use this function...	To obtain...
ANALYZE_STATISTICS (page 440)	A fixed-size statistical data sampling (10 percent per disk). This function returns results quickly, but is less accurate than using ANALYZE_HISTOGRAM to get a larger sampling of disk data.
ANALYZE_HISTOGRAM (page 437)	A specified percentage of disk data sampling (from 1 - 100). If you analyze more than 10 percent data per disk, this function is more accurate than ANALYZE_STATISTICS, but requires proportionately longer to return statistics.

Analyzing results

To retrieve hints about under-performing queries and the associated root causes, use the ***ANALYZE_WORKLOAD*** (page 443) function. This function runs the Workload Analyzer and returns tuning recommendations, such as "`run analyze_statistics on schema.table.column`". You or your database administrator should act upon the tuning recommendations.

You can also find database tuning recommendations on the Management Console.

Canceling this function

You can cancel statistics analysis by issuing CTRL-C in a vsql shell or by invoking the ***INTERRUPT_STATEMENT()*** (page 503) function.

Notes

- Always run ANALYZE_STATISTICS on a table or column rather than a projection.
- By default, HP Vertica analyzes more than one column (subject to resource limits) in a single-query execution plan to:
  - reduce plan execution latency
  - help speed up analysis of relatively small tables that have a large number of columns
- Pre-join projection statistics are updated on any pre-joined tables.

Examples

Computes statistics on all projections in the Vmart database and returns 0 (success):

```
=> SELECT ANALYZE_STATISTICS ('');
 analyze_statistics
--------------------
                  0
(1 row)
```

Computes statistics on a single table (`shipping_dimension`) and returns 0 (success):

```
=> SELECT ANALYZE_STATISTICS ('shipping_dimension');
 analyze_statistics
--------------------
                  0
(1 row)
```

Computes statistics on a single column (`shipping_key`) across all projections for the `shipping_dimension` table and returns 0 (success):

```
=> SELECT ANALYZE_STATISTICS('shipping_dimension.shipping_key');
 analyze_statistics
--------------------
                  0
(1 row)
```

For use cases, see Collecting Statistics in the Administrator's Guide

See Also

ANALYZE_HISTOGRAM (page 437)

ANALYZE_WORKLOAD (page 443)

DROP_STATISTICS (page 476)

EXPORT_STATISTICS (page 490)

IMPORT_STATISTICS (page 502)

INTERRUPT_STATEMENT (page 503)

ANALYZE_WORKLOAD

Runs the Workload Analyzer (WLA), a utility that analyzes system information held in ***system tables*** (page 933).

The Workload Analyzer intelligently monitors the performance of SQL queries and workload history, resources, and configurations to identify the root causes for poor query performance. Calling the ANALYZE_WORKLOAD function returns tuning recommendations for all events within the scope and time that you specify.

Tuning recommendations are based on a combination of statistics, system and data collector events, and database-table-projection design. WLA's recommendations let database administrators quickly and easily tune query performance without needing sophisticated skills.

See Understanding WLA Triggering Conditions in the Administrator's Guide for the most common triggering conditions and recommendations.

Syntax 1

```
ANALYZE_WORKLOAD ( 'scope' , 'since_time' );
```

Syntax 2

```
ANALYZE_WORKLOAD ( 'scope' , [ true ] );
```

Parameters

scope	Specifies which HP Vertica catalog objects to analyze. Can be one of: ▪ An empty string (`''`) returns recommendations for all database objects ▪ 'table_name' returns all recommendations related to the specified table ▪ 'schema_name' returns recommendations on all database objects in the specified schema

since_time	Limits the recommendations from all events that you specified in 'scope' since the specified time in this argument, up to the current system status. If you omit the since_time parameter, ANALYZE_WORKLOAD returns recommendations on events since the last recorded time that you called this function.
	Note: You must explicitly cast strings that you use for the since_time parameter to TIMESTAMP or TIMESTAMPTZ. For example:
	`SELECT ANALYZE_WORKLOAD('T1', '2010-10-04 11:18:15'::TIMESTAMPTZ);`
	`SELECT ANALYZE_WORKLOAD('T1', TIMESTAMPTZ '2010-10-04 11:18:15');`

true	[Optional] Tells HP Vertica to record this particular call of WORKLOAD_ANALYZER() in the system. The default value is false (do not record). If recorded, subsequent calls to ANALYZE_WORKLOAD analyze only the events that have occurred since this recorded time, ignoring all prior events.

Return value

ANALYZE_WORKLOAD() returns aggregated tuning recommendations, as described in the following table.

Column	Data type	Description
observation_count	INTEGER	Integer for the total number of events observed for this tuning recommendation. For example, if you see a return value of 1, WLA is making its first tuning recommendation for the event in 'scope'.
first_observation_time	TIMESTAMPTZ	Timestamp when the event first occurred. If this column returns a null value, the tuning recommendation is from the current status of the system instead of from any prior event.
last_observation_time	TIMESTAMPTZ	Timestamp when the event last occurred. If this column returns a null value, the tuning recommendation is from the current status of the system instead of from any prior event.
tuning_parameter	VARCHAR	Objects on which you should perform a tuning action. For example, a return value of: ▪ **`public.t`** informs the DBA to run Database Designer on table t in the public schema ▪ **`bsmith`** notifies a DBA to set a password for user bsmith

tuning_description	VARCHAR	Textual description of the tuning recommendation from the Workload Analyzer to perform on the tuning_parameter object. Examples of some of the returned values include, but are not limited to: ▪ Run database designer on table `schema.table` ▪ Create replicated projection for table `schema.table` ▪ Consider query-specific design on query ▪ Reset configuration parameter with `SELECT set_config_parameter('parameter', 'new_value')` ▪ Re-segment projection `projection-name` on high-cardinality column(s) ▪ Drop the projection `projection-name` ▪ Alter a table's partition expression ▪ Reorganize data in partitioned table ▪ Decrease the MoveOutInterval configuration parameter setting

tuning_command	VARCHAR	Command string if tuning action is a SQL command. For example, the following example statements recommend that the DBA:
		Update statistics on a particular schema's table.column:
		`SELECT ANALYZE_STATISTICS('public.table.column');`
		Resolve mismatched configuration parameter 'LockTimeout':
		`SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'LockTimeout';`
		Set the password for user bsmith:
		`ALTER USER (user) IDENTIFIED BY ('new_password');`

tuning_cost	VARCHAR	Cost is based on the type of tuning recommendation and is one of: ▪ LOW—minimal impact on resources from running the tuning command ▪ MEDIUM—moderate impact on resources from running the tuning command ▪ HIGH—maximum impact on resources from running the tuning command Depending on the size of your database or table, consider running high-cost operations after hours instead of during peak load times.

Privileges

Must be a superuser

Examples

See Analyzing Workloads through an API in the Administrator's Guide for examples.

See also

V_MONITOR.TUNING_RECOMMENDATIONS (page 1120) in this guide

Analyzing Workloads in the Administrator's Guide

Understanding WLA Triggering Conditions in the Administrator's Guide

AUDIT

Estimates the raw data size of a database, a schema, a projection, or a table as it is counted in an audit of the database size.

The AUDIT function estimates the size using the same data sampling method as the audit that HP Vertica performs to determine if a database is compliant with the database size allowances in its license. The results of this function are not considered when HP Vertica determines whether the size of the database complies with the HP Vertica license's data allowance. See How HP Vertica Calculates Database Size in the Administrator's Guide for details.

Note: This function can only audit the size of tables, projections, schemas, and databases which the user has permission to access. If a non-superuser attempts to audit the entire database, the audit will only estimate the size of the data that the user is allowed to read.

Syntax

```
AUDIT([name] [, granularity] [, error_tolerance [, confidence_level]])
```

Parameters

name	Specifies the schema, projection, or table to audit. Enter name as a string, in single quotes ('). If the

	name string is empty (''), the entire database is audited.
granularity	Indicates the level at which the audit reports its results. The recognized levels are: ▪ `'schema'` ▪ `'table'` ▪ `'projection'` By default, the granularity is the same level as *name*. For example, if *name* is a schema, then the size of the entire schema is reported. If you instead specify `'table'` as the granularity, AUDIT reports the size of each table in the schema. The granularity must be finer than that of object: specifying `'schema'` for an audit of a table has no effect. The results of an audit with a granularity are reported in the V_CATALOG.USER_AUDITS system table.
error_tolerance	Specifies the percentage margin of error allowed in the audit estimate. Enter the tolerance value as a decimal number, between 0 and 100. The default value is 5, for a 5% margin of error. **Note:** The lower this value is, the more resources the audit uses since it will perform more data sampling. Setting this value to 0 results in a full audit of the database, which is very resource intensive, as all of the data in the database is analyzed. Doing a full audit of the database significantly impacts performance and is not recommended on a production database.
confidence_level	Specifies the statistical confidence level percentage of the estimate. Enter the confidence value as a decimal number, between 0 and 100. The default value is 99, indicating a confidence level of 99%. **Note:** The higher the confidence value, the more resources the function uses since it will perform more data sampling. Setting this value to 1 results in a full audit of the database, which is very resource intensive, as all of the database is analyzed. Doing a full audit of the database significantly impacts performance and is not recommended on a production database.

Permissions

- SELECT privilege on table
- USAGE privilege on schema

Note: AUDIT() works only on the tables where the user calling the function has SELECT permissions.

Notes

Due to the iterative sampling used in the auditing process, making the error tolerance a small fraction of a percent (0.00001, for example) can cause the AUDIT function to run for a longer period than a full database audit.

Examples

To audit the entire database:

```
=> SELECT AUDIT('');
  AUDIT
----------
 76376696
(1 row)
```

To audit the database with a 25% error tolerance:

```
=> SELECT AUDIT('',25);
  AUDIT
----------
 75797126
(1 row)
```

To audit the database with a 25% level of tolerance and a 90% confidence level:

```
=> SELECT AUDIT('',25,90);
  AUDIT
----------
 76402672
(1 row)
```

To audit just the online_sales schema in the VMart example database:

```
VMart=> SELECT AUDIT('online_sales');
  AUDIT
----------
 35716504
(1 row)
```

To audit the online_sales schema and report the results by table:

```
=> SELECT AUDIT('online_sales','table');
                              AUDIT
-------------------------------------------------------------------
 See table sizes in v_catalog.user_audits for schema online_sales
(1 row)

=> \x
Expanded display is on.

=> SELECT * FROM user_audits WHERE object_schema = 'online_sales';
-[ RECORD 1 ]------------------------+-----------------------------
```

```
size_bytes                          | 64960
user_id                             | 45035996273704962
user_name                           | dbadmin
object_id                           | 45035996273717636
object_type                         | TABLE
object_schema                       | online_sales
object_name                         | online_page_dimension
audit_start_timestamp               | 2011-04-05 09:24:48.224081-04
audit_end_timestamp                 | 2011-04-05 09:24:48.337551-04
confidence_level_percent            | 99
error_tolerance_percent             | 5
used_sampling                       | f
confidence_interval_lower_bound_bytes | 64960
confidence_interval_upper_bound_bytes | 64960
sample_count                        | 0
cell_count                          | 0
-[ RECORD 2 ]-----------------------+-----------------------------
size_bytes                          | 20197
user_id                             | 45035996273704962
user_name                           | dbadmin
object_id                           | 45035996273717640
object_type                         | TABLE
object_schema                       | online_sales
object_name                         | call_center_dimension
audit_start_timestamp               | 2011-04-05 09:24:48.340206-04
audit_end_timestamp                 | 2011-04-05 09:24:48.365915-04
confidence_level_percent            | 99
error_tolerance_percent             | 5
used_sampling                       | f
confidence_interval_lower_bound_bytes | 20197
confidence_interval_upper_bound_bytes | 20197
sample_count                        | 0
cell_count                          | 0
-[ RECORD 3 ]-----------------------+-----------------------------
size_bytes                          | 35614800
user_id                             | 45035996273704962
user_name                           | dbadmin
object_id                           | 45035996273717644
object_type                         | TABLE
object_schema                       | online_sales
object_name                         | online_sales_fact
audit_start_timestamp               | 2011-04-05 09:24:48.368575-04
audit_end_timestamp                 | 2011-04-05 09:24:48.379307-04
confidence_level_percent            | 99
error_tolerance_percent             | 5
used_sampling                       | t
confidence_interval_lower_bound_bytes | 34692956
confidence_interval_upper_bound_bytes | 36536644
sample_count                        | 10000
cell_count                          | 9000000
```

AUDIT_LICENSE_SIZE

Triggers an immediate audit of the database size to determine if it is in compliance with the raw data storage allowance included in your HP Vertica license. The audit is performed in the background, so this function call returns immediately. To see the results of the audit when it is done, use the ***GET_COMPLIANCE_STATUS*** (page 494) function.

Syntax

```
AUDIT_LICENSE_SIZE()
```

Privileges

Must be a superuser

Example

```
=> SELECT audit_license_size();
 audit_license_size
--------------------
 Service hurried
(1 row)
```

AUDIT_LICENSE_TERM

Triggers an immediate audit to determine if the HP Vertica license has expired. The audit happens in the background, so this function returns immediately. To see the result of the audit, use the ***GET_COMPLIANCE_STATUS*** (page 494) function.

Syntax

```
AUDIT_LICENSE_TERM()
```

Privileges

Must be a superuser

Example

```
=> SELECT AUDIT_LICENSE_TERM();
 AUDIT_LICENSE_TERM
--------------------
 Service hurried
(1 row)
```

CANCEL_REBALANCE_CLUSTER

Stops any rebalance task currently in progress.

Syntax

```
CANCEL_REBALANCE_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT CANCEL_REBALANCE_CLUSTER();
 CANCEL_REBALANCE_CLUSTER
--------------------------
 CANCELED
(1 row)
```

See Also

- ***START_REBALANCE_CLUSTER*** (page 537)
- ***REBALANCE_CLUSTER*** (page 522)

CANCEL_REFRESH

Cancels refresh-related internal operations initiated by START_REFRESH().

Syntax

```
CANCEL_REFRESH()
```

Privileges

None

Notes

- Refresh tasks run in a background thread in an internal session, so you cannot use ***INTERRUPT_STATEMENT*** (page 503) to cancel those statements. Instead, use CANCEL_REFRESH to cancel statements that are run by refresh-related internal sessions.
- Run CANCEL_REFRESH() on the same node on which START_REFRESH() was initiated.
- CANCEL_REFRESH() cancels the refresh operation running on a node, waits for the cancelation to complete, and returns SUCCESS.
- Only one set of refresh operations runs on a node at any time.

See Also

INTERRUPT_STATEMENT (page 503)

SESSIONS (page 1095)

START_REFRESH (page 538)

PROJECTION_REFRESHES (page 1056)

CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY

Changes the run-time priority of a query that is actively running.

Syntax

```
CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY(TRANSACTION_ID, 'value')
```

Parameters

TRANSACTION_ID	An identifier for the transaction within the session. TRANSACTION_ID cannot be NULL. You can find the transaction ID in the Sessions table.
'value'	The RUNTIMEPRIORITY value. Can be HIGH, MEDIUM, or LOW.

Privileges

No special privileges required. However, non-super users can change the run-time priority of their own queries only. In addition, non-superusers can never raise the run-time priority of a query to a level higher than that of the resource pool.

Example

```
VMart => SELECT CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY(45035996273705748,
'low')
```

CHANGE_RUNTIME_PRIORITY

Changes the run-time priority of a query that is actively running. Note that, while this function is still valid, you should instead use CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY to change run-time priority. CHANGE_RUNTIME_PRIORITY will be deprecated in a future release of Vertica.

Syntax

```
CHANGE_RUNTIME_PRIORITY(TRANSACTION_ID,STATEMENT_ID, 'value')
```

Parameters

TRANSACTION_ID	An identifier for the transaction within the session. TRANSACTION_ID cannot be NULL. You can find the transaction ID in the Sessions table.
STATEMENT_ID	A unique numeric ID assigned by the HP Vertica catalog, which identifies the currently executing statement. You can find the statement ID in the Sessions table. You can specify NULL to change the run-time priority of the currently running query within the transaction.
'value'	The RUNTIMEPRIORITY value. Can be HIGH, MEDIUM, or LOW.

Privileges

No special privileges required. However, non-super users can change the run-time priority of their own queries only. In addition, non-superusers can never raise the run-time priority of a query to a level higher than that of the resource pool.

Example

```
VMart => SELECT CHANGE_RUNTIME_PRIORITY(45035996273705748, NULL, 'low')
```

CLEAR_CACHES

Clears the HP Vertica internal cache files.

Syntax

```
CLEAR_CACHES ( )
```

Privileges

Must be a superuser

Notes

If you want to run benchmark tests for your queries, in addition to clearing the internal HP Vertica cache files, clear the Linux file system cache. The kernel uses unallocated memory as a cache to hold clean disk blocks. If you are running version 2.6.16 or later of Linux and you have root access, you can clear the kernel filesystem cache as follows:

1. Make sure that all data is the cache is written to disk:
   ```
   # sync
   ```
2. Writing to the `drop_caches` file causes the kernel to drop clean caches, dentries, and inodes from memory, causing that memory to become free, as follows:
   - To clear the page cache:
     ```
     # echo 1 > /proc/sys/vm/drop_caches
     ```
   - To clear the dentries and inodes:
     ```
     # echo 2 > /proc/sys/vm/drop_caches
     ```
   - To clear the page cache, dentries, and inodes:
     ```
     # echo 3 > /proc/sys/vm/drop_caches
     ```

Example

The following example clears the HP Vertica internal cache files:

```
=> CLEAR_CACHES();
 CLEAR_CACHES
--------------
 Cleared
(1 row)
```

CLEAR_DATA_COLLECTOR

Clears all memory and disk records on the Data Collector tables and functions and resets collection statistics in the V_MONITOR.DATA_COLLECTOR system table. A superuser can clear Data Collector data for all components or specify an individual component

After you clear the DataCollector log, the information is no longer available for querying.

Syntax

```
CLEAR_DATA_COLLECTOR( [ 'component' ] )
```

Parameters

component	Clears memory and disk records for the specified component only. If you provide no argument, the function clears all Data Collector memory and disk records for all components. For the current list of component names, query the ***V_MONITOR.DATA_COLLECTOR*** (page 1002) system table.

Privileges

Must be a superuser

Example

The following command clears memory and disk records for the ResourceAcquisitions component:

```
=> SELECT clear_data_collector('ResourceAcquisitions');
 clear_data_collector
----------------------
 CLEAR
(1 row)
```

The following command clears data collection for all components on all nodes:

```
=> SELECT clear_data_collector();
 clear_data_collector
----------------------
 CLEAR
(1 row)
```

See Also

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining Monitoring Information in the Administrator's Guide

CLEAR_PROFILING

HP Vertica stores profiled data is in memory, so depending on how much data you collect, profiling could be memory intensive. You can use this function to clear profiled data from memory.

Syntax

```
CLEAR_PROFILING( 'profiling-type' )
```

Parameters

profiling-type	The type of profiling data you want to clear. Can be one of: ▪ session—clears profiling for basic session parameters and lock time out data ▪ query—clears profiling for general information about queries that ran, such as the query strings used and the duration of queries ▪ ee—clears profiling for information about the execution run of each query

Example

The following statement clears profiled data for queries:

```
=> SELECT CLEAR_PROFILING('query');
```

See also

DISABLE_PROFILING (page 469)

ENABLE_PROFILING (page 483)

Profiling Database Performance in the Administrator's Guide

CLEAR_PROJECTION_REFRESHES

Triggers HP Vertica to clear information about refresh operations for projections immediately.

Syntax

```
CLEAR_PROJECTION_REFRESHES()
```

Notes

Information about a refresh operation—whether successful or unsuccessful—is maintained in the ***PROJECTION_REFRESHES*** (page 1056) system table until either the ***CLEAR_PROJECTION_REFRESHES*** (page 455)() function is executed or the storage quota for the table is exceeded. The `PROJECTION_REFRESHES.IS_EXECUTING` column returns a boolean value that indicates whether the refresh is currently running (t) or occurred in the past (f).

Privileges

Must be a superuser

Example

To immediately purge projection refresh history, use the CLEAR_PROJECTION_REFRESHES() function:

```
=> SELECT CLEAR_PROJECTION_REFRESHES();
 CLEAR_PROJECTION_REFRESHES
----------------------------
 CLEAR
(1 row)
```

Only the rows where the `PROJECTION_REFRESHES.IS_EXECUTING` column equals false are cleared.

See Also

PROJECTION_REFRESHES (page 1056)

REFRESH (page 523)

START_REFRESH (page 538)

Clearing PROJECTION_REFRESHES History in the Administrator's Guide

CLEAR_RESOURCE_REJECTIONS

Clears the content of the ***RESOURCE_REJECTIONS*** (page 1089) and ***DISK_RESOURCE_REJECTIONS*** (page 1013) system tables. Normally, these tables are only cleared during a node restart. This function lets you clear the tables whenever you need. For example, you might want to clear the system tables after you resolved a disk space issue that was causing disk resource rejections.

Syntax

```
CLEAR_RESOURCE_REJECTIONS();
```

Privileges

Must be a superuser

Example

The following command clears the content of the RESOURCE_REJECTIONS and DISK_RESOURCE_REJECTIONS system tables:

```
=> SELECT clear_resource_rejections();
 clear_resource_rejections
---------------------------
 OK
(1 row)
```

See Also

DISK_RESOURCE_REJECTIONS (page 1013)

RESOURCE_REJECTIONS (page 1089)

CLEAR_OBJECT_STORAGE_POLICY

Removes an existing storage policy. The specified object will no longer use a default storage location. Any existing data stored currently at the labeled location in the object's storage policy is moved to default storage during the next TM moveout operation.

Syntax

```
CLEAR_OBJECT_STORAGE_POLICY ( 'object_name' , [', key_min, key_max '])
```

Parameters

object_name	Specifies the database object with a storage policy to clear.
key_min, key_max	Specifies the table partition key value ranges stored at the labeled location. These parameters are applicable only when *object_name* is a table.

Privileges

Must be a superuser

Example

This example clears the storage policy for the object `lineorder`:

```
release=> select clear_object_storage_policy('lineorder');
      clear_object_storage_policy
-----------------------------------
 Default storage policy cleared.
(1 row)
```

See Also

Clearing a Storage Policy in the Administrator's Guide

ALTER_LOCATION_LABEL (page 430)

SET_OBJECT_STORAGE_POLICY (page 534)

CLOSE_SESSION

Interrupts the specified external session, rolls back the current transaction, if any, and closes the socket.

Syntax

```
CLOSE_SESSION ( 'sessionid' )
```

Parameters

sessionid	A string that specifies the session to close. This identifier is unique within the cluster at any point in time but can be reused when the session closes.

Privileges

None; however, a non-superuser can only close his or her own session.

Notes

- Closing of the session is processed asynchronously. It could take some time for the session to be closed. Check the ***SESSIONS*** (page 1095) table for the status.
- Database shutdown is prevented if new sessions connect after the CLOSE_SESSION() command is invoked (and before the database is actually shut down. See **Controlling Sessions** below.

Messages

The following are the messages you could encounter:

- For a badly formatted sessionID

  ```
  close_session | Session close command sent. Check SESSIONS for progress.
  Error: invalid Session ID format
  ```

- For an incorrect sessionID parameter

  ```
  Error: Invalid session ID or statement key
  ```

Examples

User session opened. RECORD 2 shows the user session running COPY DIRECT statement.

```
vmartdb=> SELECT * FROM sessions;
-[ RECORD 1 ]-------------+-----------------------------------------------
node_name                | v_vmartdb_node0001
user_name                | dbadmin
client_hostname          | 127.0.0.1:52110
client_pid               | 4554
login_timestamp          | 2011-01-03 14:05:40.252625-05
session_id               | stress04-4325:0x14
client_label             |
transaction_start        | 2011-01-03 14:05:44.325781
transaction_id           | 45035996273728326
transaction_description  | user dbadmin (SELECT * FROM sessions;)
```

```
statement_start            | 2011-01-03 15:36:13.896288
statement_id               | 10
last_statement_duration_us | 14978
current_statement          | select * from sessions;
ssl_state                  | None
authentication_method      | Trust
-[ RECORD 2 ]--------------+-----------------------------------------------
node_name                  | v_vmartdb_node0002
user_name                  | dbadmin
client_hostname            | 127.0.0.1:57174
client_pid                 | 30117
login_timestamp            | 2011-01-03 15:33:00.842021-05
session_id                 | stress05-27944:0xc1a
client_label               |
transaction_start          | 2011-01-03 15:34:46.538102
transaction_id             | -1
transaction_description    | user dbadmin (COPY ClickStream_Fact FROM
                             '/data/clickstream/1g/ClickStream_Fact.tbl'
                             DELIMITER '|' NULL '\\n' DIRECT;)
statement_start            | 2011-01-03 15:34:46.538862
statement_id               |
last_statement_duration_us | 26250
current_statement          | COPY ClickStream_Fact FROM '/data/clickstream
                             /1g/ClickStream_Fact.tbl' DELIMITER '|' NULL
                             '\\n' DIRECT;
ssl_state                  | None
authentication_method      | Trust
```

Close user session stress05-27944:0xc1a

```
vmartdb=> \x
Expanded display is off.
vmartdb=> SELECT CLOSE_SESSION('stress05-27944:0xc1a');
                         CLOSE_SESSION
-------------------------------------------------------------------
 Session close command sent. Check v_monitor.sessions for progress.
(1 row)
```

Query the sessions table again for current status, and you can see that the second session has been closed:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1 ]--------------+-----------------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
client_pid                 | 4554
login_timestamp            | 2011-01-03 14:05:40.252625-05
session_id                 | stress04-4325:0x14
client_label               |
transaction_start          | 2011-01-03 14:05:44.325781
transaction_id             | 45035996273728326
transaction_description    | user dbadmin (select * from SESSIONS;)
statement_start            | 2011-01-03 16:12:07.841298
statement_id               | 20
last_statement_duration_us | 2099
```

```
current_statement           | SELECT * FROM SESSIONS;
ssl_state                   | None
authentication_method       | Trust
```

Controlling Sessions

The database administrator must be able to disallow new incoming connections in order to shut down the database. On a busy system, database shutdown is prevented if new sessions connect after the CLOSE_SESSION or CLOSE_ALL_SESSIONS() command is invoked — and before the database actually shuts down.

One option is for the administrator to issue the `SHUTDOWN('true')` command, which forces the database to shut down and disallow new connections. See ***SHUTDOWN*** (page 535) in the SQL Reference Manual.

Another option is to modify the `MaxClientSessions` parameter from its original value to 0, in order to prevent new non-dbadmin users from connecting to the database.

1. Determine the original value for the `MaxClientSessions` parameter by querying the V_MONITOR.***CONFIGURATIONS_PARAMETERS*** (page 996) system table:

   ```
   => SELECT CURRENT_VALUE FROM CONFIGURATION_PARAMETERS WHERE
   parameter_name='MaxClientSessions';
    CURRENT_VALUE
   ---------------
    50
   (1 row)
   ```

2. Set the `MaxClientSessions` parameter to 0 to prevent new non-dbadmin connections:

   ```
   => SELECT SET_CONFIG_PARAMETER('MaxClientSessions', 0);
   ```

   Note: The previous command allows up to five administrators to log in.

3. Issue the `CLOSE_ALL_SESSIONS`() command to remove existing sessions:

   ```
   => SELECT CLOSE_ALL_SESSIONS();
   ```

4. Query the `SESSIONS` table:

   ```
   => SELECT * FROM SESSIONS;
   ```

   When the session no longer appears in the `SESSIONS` table, disconnect and run the Stop Database command.

5. Restart the database.

6. Restore the `MaxClientSessions` parameter to its original value:

   ```
   => SELECT SET_CONFIG_PARAMETER('MaxClientSessions', 50);
   ```

See Also

CLOSE_ALL_SESSIONS (page 461), ***CONFIGURATION_PARAMETERS*** (page 996), ***SESSIONS*** (page 1095), ***SHUTDOWN*** (page 535)

Managing Sessions and Configuration Parameters in the Administrator's Guide

CLOSE_ALL_SESSIONS

Closes all external sessions except the one issuing the CLOSE_ALL_SESSIONS functions.

Syntax

```
CLOSE_ALL_SESSIONS()
```

Privileges

None; however, a non-superuser can only close his or her own session.

Notes

Closing of the sessions is processed asynchronously. It might take some time for the session to be closed. Check the ***SESSIONS*** (page 1095) table for the status.

Database shutdown is prevented if new sessions connect after the CLOSE_SESSION or CLOSE_ALL_SESSIONS() command is invoked (and before the database is actually shut down). See **Controlling Sessions** below.

Message

```
close_all_sessions | Close all sessions command sent.
Check SESSIONS for progress.
```

Examples

Two user sessions opened, each on a different node:

```
vmartdb=> SELECT * FROM sessions;
-[ RECORD 1
]-------------+-----------------------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
client_pid                 | 4554
login_timestamp            | 2011-01-03 14:05:40.252625-05
session_id                 | stress04-4325:0x14
client_label               |
transaction_start          | 2011-01-03 14:05:44.325781
transaction_id             | 45035996273728326
transaction_description    | user dbadmin (select * from sessions;)
statement_start            | 2011-01-03 15:36:13.896288
statement_id               | 10
last_statement_duration_us | 14978
current_statement          | select * from sessions;
ssl_state                  | None
authentication_method      | Trust
-[ RECORD 2
]-------------+-----------------------------------------------------
node_name                  | v_vmartdb_node0002
user_name                  | dbadmin
client_hostname            | 127.0.0.1:57174
client_pid                 | 30117
login_timestamp            | 2011-01-03 15:33:00.842021-05
```

```
session_id                  | stress05-27944:0xc1a
client_label                |
transaction_start           | 2011-01-03 15:34:46.538102
transaction_id              | -1
transaction_description     | user dbadmin (COPY Mart_Fact FROM
'/data/mart_Fact.tbl'
                              DELIMITER '|' NULL '\\n';)
statement_start             | 2011-01-03 15:34:46.538862
statement_id                |
last_statement_duration_us | 26250
current_statement           | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER
'|'
                              NULL '\\n';
ssl_state                   | None
authentication_method       | Trust
-[ RECORD 3
]--------------+-----------------------------------------------------
node_name                   | v_vmartdb_node0003
user_name                   | dbadmin
client_hostname             | 127.0.0.1:56367
client_pid                  | 1191
login_timestamp             | 2011-01-03 15:31:44.939302-05
session_id                  | stress06-25663:0xbec
client_label                |
transaction_start           | 2011-01-03 15:34:51.05939
transaction_id              | 54043195528458775
transaction_description     | user dbadmin (COPY Mart_Fact FROM
'/data/Mart_Fact.tbl'
                              DELIMITER '|' NULL '\\n' DIRECT;)
statement_start             | 2011-01-03 15:35:46.436748
statement_id                |
last_statement_duration_us | 1591403
current_statement           | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER
'|'
                              NULL '\\n' DIRECT;
ssl_state                   | None
authentication_method       | Trust
```

Close all sessions:

```
vmartdb=> \x
Expanded display is off.
vmartdb=> SELECT CLOSE_ALL_SESSIONS();
                          CLOSE_ALL_SESSIONS
-------------------------------------------------------------------------
 Close all sessions command sent. Check v_monitor.sessions for progress.
(1 row)
```

Sessions contents after issuing the CLOSE_ALL_SESSIONS() command:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1 ]--------------+---------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
```

```
client_pid                  | 4554
login_timestamp             | 2011-01-03 14:05:40.252625-05
session_id                  | stress04-4325:0x14
client_label                |
transaction_start           | 2011-01-03 14:05:44.325781
transaction_id              | 45035996273728326
transaction_description     | user dbadmin (SELECT * FROM sessions;)
statement_start             | 2011-01-03 16:19:56.720071
statement_id                | 25
last_statement_duration_us | 15605
current_statement           | SELECT * FROM SESSIONS;
ssl_state                   | None
authentication_method       | Trust
```

Controlling Sessions

The database administrator must be able to disallow new incoming connections in order to shut down the database. On a busy system, database shutdown is prevented if new sessions connect after the CLOSE_SESSION or CLOSE_ALL_SESSIONS() command is invoked — and before the database actually shuts down.

One option is for the administrator to issue the `SHUTDOWN('true')` command, which forces the database to shut down and disallow new connections. See ***SHUTDOWN*** (page 535) in the SQL Reference Manual.

Another option is to modify the `MaxClientSessions` parameter from its original value to 0, in order to prevent new non-dbadmin users from connecting to the database.

1. Determine the original value for the `MaxClientSessions` parameter by querying the V_MONITOR.***CONFIGURATIONS_PARAMETERS*** (page 996) system table:

   ```
   => SELECT CURRENT_VALUE FROM CONFIGURATION_PARAMETERS WHERE
   parameter_name='MaxClientSessions';
    CURRENT_VALUE
   ---------------
    50
   (1 row)
   ```

2. Set the `MaxClientSessions` parameter to 0 to prevent new non-dbadmin connections:

   ```
   => SELECT SET_CONFIG_PARAMETER('MaxClientSessions', 0);
   ```

   Note: The previous command allows up to five administrators to log in.

3. Issue the `CLOSE_ALL_SESSIONS`() command to remove existing sessions:

   ```
   => SELECT CLOSE_ALL_SESSIONS();
   ```

4. Query the `SESSIONS` table:

   ```
   => SELECT * FROM SESSIONS;
   ```

   When the session no longer appears in the `SESSIONS` table, disconnect and run the Stop Database command.

5. Restart the database.

6. Restore the `MaxClientSessions` parameter to its original value:

   ```
   => SELECT SET_CONFIG_PARAMETER('MaxClientSessions', 50);
   ```

See Also

CLOSE_SESSION (page 458)

CONFIGURATION_PARAMETERS (page 996)

SHUTDOWN (page 535)

V_MONITOR.SESSIONS (page 1095)

Managing Sessions and Configuration Parameters in the Administrator's Guide

CURRENT_SCHEMA

Returns the name of the current schema.

Behavior Type

Stable

Syntax

```
CURRENT_SCHEMA()
```

Privileges

None

Notes

The CURRENT_SCHEMA function does not require parentheses.

Example

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

The following command returns the same results without the parentheses:

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

The following command shows the current schema, listed after the ***current user*** (page 416), in the search path:

```
=> SHOW SEARCH_PATH;
    name     |                       setting
-------------+-------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

See Also

SET SEARCH_PATH

DATA_COLLECTOR_HELP

Returns online usage instructions about the Data Collector, the V_MONITOR.DATA_COLLECTOR system table, and the Data Collector control functions.

Syntax

```
DATA_COLLECTOR_HELP()
```

Privileges

None

Returns

Invoking DATA_COLLECTOR_HELP() returns the following information:

```
=> SELECT DATA_COLLECTOR_HELP();

   Usage Data Collector
   The data collector retains history of important system activities.
   This data can be used as a reference of what actions have been taken
     by users, but it can also be used to locate performance bottlenecks,
     or identify potential improvements to the Vertica configuration.
   This data is queryable via Vertica system tables.


   The list of data collector components, and some statistics, can be found using:
   SELECT * FROM v_monitor.data_collector;

   The amount of data retained can be controlled with:
      set_data_collector_policy(<component>,
                           <memory retention (KB)>,
                           <disk retention (KB)>);


   The current retention policy for a component can be queried with:
      get_data_collector_policy(<component>);


   Data on disk is kept in the "DataCollector" directory under the Vertica
   catalog path. This directory also contains instructions on how to load
   the monitoring data into another Vertica database.


   Additional commands can be used to affect the data collection logs.
   The log can be cleared with:
     clear_data_collector([<optional component>]);
```

```
The log can be synchronized with the disk storage using:
  flush_data_collector([<optional component>]);
```

Note: Data Collector works in conjunction with the Workload Analyzer, an advisor tool that intelligently monitors the performance of SQL queries and workloads and recommends tuning actions based on observations of the actual workload history.

See Also

V_MONITOR.DATA_COLLECTOR (page 1002)

V_MONITOR.TUNING_RECOMMENDATIONS (page 1120)

Analyzing Workloads in the Administrator's Guide

Retaining Monitoring Information in the Administrator's Guide

DISABLE_DUPLICATE_KEY_ERROR

Disables error messaging when HP Vertica finds duplicate PRIMARY KEY/UNIQUE KEY values at run time. Queries execute as though no constraints are defined on the schema. Effects are session scoped.

CAUTION: When called, DISABLE_DUPLICATE_KEY_ERROR() suppresses data integrity checking and can lead to incorrect query results. Use this function only after you insert duplicate primary keys into a dimension table in the presence of a pre-join projection. Then correct the violations and turn integrity checking back on with ***REENABLE_DUPLICATE_KEY_ERROR*** (page 522)().

Syntax

```
DISABLE_DUPLICATE_KEY_ERROR();
```

Privileges

Must be a superuser

Notes

The following series of commands create a table named `dim` and the corresponding projection:

```
CREATE TABLE dim (pk INTEGER PRIMARY KEY, x INTEGER);
CREATE PROJECTION dim_p (pk, x) AS SELECT * FROM dim ORDER BY x UNSEGMENTED ALL
NODES;
```

The next two statements create a table named `fact` and the pre-join projection that joins `fact` to `dim`.

```
CREATE TABLE fact(fk INTEGER REFERENCES dim(pk));
CREATE PROJECTION prejoin_p (fk, pk, x) AS SELECT * FROM fact, dim WHERE pk=fk ORDER
BY x;
```

The following statements load values into table `dim`. The last statement inserts a duplicate primary key value of 1:

```
INSERT INTO dim values (1,1);
INSERT INTO dim values (2,2);
INSERT INTO dim values (1,2); --Constraint violation
COMMIT;
```

Table `dim` now contains duplicate primary key values, but you cannot delete the violating row because of the presence of the pre-join projection. Any attempt to delete the record results in the following error message:

```
ROLLBACK:  Duplicate primary key detected in FK-PK join Hash-Join (x dim_p), value
1
```

In order to remove the constraint violation `(pk=1)`, use the following sequence of commands, which puts the database back into the state just before the duplicate primary key was added.

To remove the violation:

1. Save the original `dim` rows that match the duplicated primary key:
   ```
   CREATE TEMP TABLE dim_temp(pk integer, x integer);
   INSERT INTO dim_temp SELECT * FROM dim WHERE pk=1 AND x=1; -- original
   dim row
   ```
2. Temporarily disable error messaging on duplicate constraint values:
   ```
   SELECT DISABLE_DUPLICATE_KEY_ERROR();
   ```
   Caution: Remember that running the `DISABLE DUPLICATE KEY ERROR` function suppresses the enforcement of data integrity checking.
3. Remove the original row that contains duplicate values:
   ```
   DELETE FROM dim WHERE pk=1;
   ```
4. Allow the database to resume data integrity checking:
   ```
   SELECT REENABLE_DUPLICATE_KEY_ERROR();
   ```
5. Reinsert the original values back into the dimension table:
   ```
   INSERT INTO dim SELECT * from dim_temp;
   COMMIT;
   ```
6. Validate your dimension and fact tables.

If you receive the following error message, it means that the duplicate records you want to delete are not identical. That is, the records contain values that differ in at least one column that is not a primary key; for example, (1,1) and (1,2).

```
ROLLBACK: Delete: could not find a data row to delete (data integrity violation?)
```

The difference between this message and the rollback message in the previous example is that a fact row contains a foreign key that matches the duplicated primary key, which has been inserted. A row with values from the fact and dimension table is now in the pre-join projection. In order for the DELETE statement (Step 3 in the following example) to complete successfully, extra predicates are required to identify the original dimension table values (the values that are in the pre-join).

This example is nearly identical to the previous example, except that an additional INSERT statement joins the fact table to the dimension table by a primary key value of 1:

```
INSERT INTO dim  values (1,1);
INSERT INTO dim  values (2,2);
```

```
INSERT INTO fact values (1);   -- New insert statement joins fact with dim on
primary key value=1
INSERT INTO dim  values (1,2); -- Duplicate primary key value=1
COMMIT;
```

To remove the violation:

1 Save the original `dim` and `fact` rows that match the duplicated primary key:

```
CREATE TEMP TABLE dim_temp(pk integer, x integer);
CREATE TEMP TABLE fact_temp(fk integer);
INSERT INTO dim_temp SELECT * FROM dim WHERE pk=1 AND x=1; -- original
dim row
INSERT INTO fact_temp SELECT * FROM fact WHERE fk=1;
```

2 Temporarily suppresses the enforcement of data integrity checking:

```
SELECT DISABLE_DUPLICATE_KEY_ERROR();
```

3 Remove the duplicate primary keys. These steps also implicitly remove all fact rows with the matching foreign key.

a) Remove the original row that contains duplicate values:

```
DELETE FROM dim WHERE pk=1 AND x=1;
```

Note: The extra predicate (`x=1`) specifies removal of the original (`1,1`) row, rather than the newly inserted (`1,2`) values that caused the violation.

b) Remove all remaining rows:

```
DELETE FROM dim WHERE pk=1;
```

4 Reenable integrity checking:

```
SELECT REENABLE_DUPLICATE_KEY_ERROR();
```

5 Reinsert the original values back into the fact and dimension table:

```
INSERT INTO dim SELECT * from dim_temp;
INSERT INTO fact SELECT * from fact_temp;
COMMIT;
```

Validate your dimension and fact tables.

See Also

ANALYZE_CONSTRAINTS (page 432)

REENABLE_DUPLICATE_KEY_ERROR (page 522)

DISABLE_ELASTIC_CLUSTER

Disables elastic cluster scaling, which prevents HP Vertica from bundling data into chunks that are easily transportable to other nodes when performing cluster resizing. The main reason to disable elastic clustering is if you find that the slightly unequal data distribution in your cluster caused by grouping data into discrete blocks results in performance issues.

Syntax

```
DISABLE_ELASTIC_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT DISABLE_ELASTIC_CLUSTER();
 DISABLE_ELASTIC_CLUSTER
-------------------------
 DISABLED
(1 row)
```

See Also

- ***ENABLE_ELASTIC_CLUSTER*** (page 482)

DISABLE_LOCAL_SEGMENTS

Disable local data segmentation, which breaks projections segments on nodes into containers that can be easily moved to other nodes. See Local Data Segmentation in the Administrator's Guide for details.

Syntax

```
DISABLE_LOCAL_SEGMENTS()
```

Privileges

Must be a superuser

Example

```
=> SELECT DISABLE_LOCAL_SEGMENTS();
 DISABLE_LOCAL_SEGMENTS
------------------------
 DISABLED
(1 row)
```

DISABLE_PROFILING

Disables profiling for the profiling type you specify.

Syntax

```
DISABLE_PROFILING( 'profiling-type' )
```

Parameters

profiling-type	The type of profiling data you want to disable. Can be one of: ▪ session—disables profiling for basic session parameters and lock time out data ▪ query—disables profiling for general information about queries that ran, such as the query strings used and the duration of queries ▪ ee—disables profiling for information about the execution run of each query

Example

The following statement disables profiling on query execution runs:

```
=> SELECT DISABLE_PROFILING('ee');
   DISABLE_PROFILING
-----------------------
 EE Profiling Disabled
(1 row)
```

See also

CLEAR_PROFILING (page 455)

ENABLE_PROFILING (page 483)

Profiling Database Performance in the Administrator's Guide

DISPLAY_LICENSE

Returns the terms of your HP Vertica license. The information this function displays is:

- The start and end dates for which the license is valid (or "Perpetual" if the license has no expiration).
- The number of days you are allowed to use HP Vertica after your license term expires (the grace period)
- The amount of data your database can store, if your license includes a data allowance.

Syntax

```
DISPLAY_LICENSE()
```

Privileges

None

Examples

```
=> SELECT DISPLAY_LICENSE();
                  DISPLAY_LICENSE
-----------------------------------------------------
 HP Vertica Systems, Inc.
1/1/2011
```

```
12/31/2011
30
50TB

(1 row)
```

DO_TM_TASK

Runs a Tuple Mover operation on one or more projections defined on the specified table.

Tip: You do not need to stop the Tuple Mover to run this function.

Syntax

```
DO_TM_TASK ( 'task' [ , '[[db-name.]schema.]table' |
'[[db-name.]schema.]projection' ] )
```

Parameters

task	Is one of the following tuple mover operations: ▪ `'moveout'` — Moves out all projections on the specified table (if a particular projection is not specified) from WOS to ROS. ▪ `'mergeout'` — Consolidates ROS containers and purges deleted records. ▪ `'analyze_row_count'` — Automatically collects the number of rows in a projection every 60 seconds and aggregates row counts calculated during loads.
[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Runs a tuple mover operation for all projections within the specified table. When using more than one schema, specify the schema that contains the table with the projections you want to affect, as noted above.

projection	If *projection* is not passed as an argument, all projections in the system are used. If *projection* is specified, DO_TM_TASK looks for a projection of that name and, if found, uses it; if a named projection is not found, the function looks for a table with that name and, if found, moves out all projections on that table.

Privileges

- Any INSERT/UPDATE/DELETE privilege on table
- USAGE privileges on schema

Notes

`DO_TM_TASK()` is useful for moving out all projections from a table or database without having to name each projection individually.

Examples

The following example performs a moveout of all projections for table `t1`:

```
=> SELECT DO_TM_TASK('moveout', 't1');
```

The following example performs a moveout for projection t1_proj:

```
=> SELECT DO_TM_TASK('moveout', 't1_proj')
```

See Also

COLUMN_STORAGE (page 992)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

Partitioning Tables in the Administrator's Guide

Collecting Statistics in the Administrator's Guide

DROP_LOCATION

Removes the specified storage location.

Syntax

```
DROP_LOCATION ( 'path' , 'node' )
```

Parameters

path	Specifies where the storage location to drop is mounted.
node	Is the HP Vertica node where the location is available.

Privileges

Must be a superuser

Retiring or Dropping a Storage Location

Dropping a storage location is a permanent operation and cannot be undone. Therefore, HP recommends that you retire a storage location before dropping it. Retiring a storage location lets you verify that you do not need the storage before dropping it. Additionally, you can easily restore a retired storage location if you determine it is still in use.

Storage Locations with Temp and Data Files

Dropping storage locations is limited to storage locations that contain only temp files.

If you use a storage location to store data and then alter it to store only temp files, the location can still contain data files. HP Vertica does not let you drop a storage location containing data files. You can manually merge out the data files from the storage location, and then wait for the ATM to mergeout the data files automatically, or, you can drop partitions. Deleting data files does not work.

Example

The following example drops a storage location on node3 that was used to store temp files:

```
=> SELECT DROP_LOCATION('/secondHP VerticaStorageLocation/' , 'node3');
```

See Also

Dropping Storage Locations and Retiring Storage Locations in the Administrator's Guide

ADD_LOCATION (page 426)

ALTER_LOCATION_USE (page 429)

RESTORE_LOCATION (page 525)

RETIRE_LOCATION (page 526)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

DROP_PARTITION

Forces the partition of projections (if needed) and then drops the specified partition.

Syntax

```
DROP_PARTITION ( table_name , partition_value [ , ignore_moveout_errors,
reorganize_data ])
```

Parameters

table-name	Specifies the name of the table. **Note**: The *table_name* argument cannot be used as a dimension table in a pre-joined projection and cannot contain projections that are not up to date (have not been refreshed).
partition_value	The key of the partition to drop. For example: `DROP_PARTITION('trade', 2006);`
ignore_moveout_errors	Optional Boolean, defaults to `false`. ▪ `true`—Ignores any WOS moveout errors and forces the operation to continue. Set this parameter to `true` only if there is no WOS data for the partition. ▪ `false` (or omit)—Displays any moveout errors and aborts the operation on error. **Note**: If you set this parameter to `true` and the WOS includes data for the partition in WOS, partition data in WOS is not dropped.
reorganize_data	Optional Boolean, defaults to `false`. ▪ `true`—Reorganizes the data if it is not organized, and then drops the partition. ▪ `false`—Does not attempt to reorganize the data before dropping the partition. If this parameter is `false` and the function encounters a ROS without partition keys, an error occurs.

Permissions

- Table owner
- USAGE privilege on schema that contains the table

Notes and Restrictions

The results of a DROP_PARTITION call go into effect immediately. If you drop a partition using DROP_PARTITION and then try to add data to a partition with the same name, HP Vertica creates a new partition.

If the operation cannot obtain an ***O Lock*** (page 1037) on the table(s), HP Vertica attempts to close any internal Tuple Mover (TM) sessions running on the same table(s) so that the operation can proceed. Explicit TM operations that are running in user sessions are not closed. If an explicit TM operation is running on the table, then the operation cannot proceed until the explicit TM operation completes.

In general, if a ROS container has data that belongs to *n*+1 partitions and you want to drop a specific partition, the DROP_PARTITION operation:

1 Forces the partition of data into two containers where
   - One container holds the data that belongs to the partition that is to be dropped.
   - Another container holds the remaining *n* partitions.

2 Drops the specified partition.

You can also use the ***MERGE_PARTITIONS*** (page 513) function to merge ROS containers that have data belonging to partitions in a specified partition key range; for example, `[partitionKeyFrom, partitionKeyTo]`.

DROP_PARTITION forces a moveout if there is data in the WOS (WOS is not partition aware).

DROP_PARTITION acquires an exclusive lock on the table to prevent DELETE | UPDATE | INSERT | COPY statements from affecting the table, as well as any SELECT statements issued at SERIALIZABLE isolation level.

You cannot perform a DROP_PARTITION operation on tables with projections that are not up to date (have not been refreshed).

DROP_PARTITION fails if you do not set the optional third parameter to *true* and the function encounters ROS's that do not have partition keys.

Examples

Using the example schema in Defining Partitions, the following command explicitly drops the 2009 partition key from table `trade`:

```
 SELECT DROP_PARTITION('trade', 2009);
  DROP_PARTITION
-------------------
 Partition dropped
(1 row)
```

Here, the partition key is specified:

```
SELECT DROP_PARTITION('trade', EXTRACT('year' FROM '2009-01-01'::date));
  DROP_PARTITION
-------------------
 Partition dropped
(1 row)
```

The following example creates a table called dates and partitions the table by year:

```
CREATE TABLE dates (
    year INTEGER NOT NULL,
    month VARCHAR(8) NOT NULL)
PARTITION BY year * 12 + month;
```

The following statement drops the partition using a constant for Oct 2010 (2010*12 + 10 = 24130):

```
 SELECT DROP_PARTITION('dates', '24130');
  DROP_PARTITION
-------------------
 Partition dropped
(1 row)
```

Alternatively, the expression can be placed in line: `SELECT DROP_PARTITION('dates', 2010*12 + 10);`

The following command first reorganizes the data if it is unpartitioned and then explicitly drops the 2009 partition key from table `trade`:

```
SELECT DROP_PARTITION('trade', 2009, false, true);

  DROP_PARTITION
-------------------
 Partition dropped
(1 row)
```

See Also

Dropping Partitions in the Administrator's Guide

ADVANCE EPOCH (page 429)

ALTER PROJECTION (page 659)

COLUMN_STORAGE (page 992)

CREATE TABLE (page 770)

DO_TM_TASK (page 471)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

MERGE_PARTITIONS (page 513)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

PROJECTIONS (page 961)

DROP_STATISTICS

Removes statistics for the specified table and lets you optionally specify the category of statistics to drop.

Syntax

```
DROP_STATISTICS { ('') | ('[[db-name.]schema-name.]table' [, {'BASE' |
'HISTOGRAMS' | 'ALL'} ])};
```

Return Value

0 - If successful, DROP_STATISTICS always returns 0. If the command fails, DROP_STATISTICS displays an error message. See `vertica.log` for message details.

Parameters

`''`	Empty string. Drops statistics for all projections.

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table	Drops statistics for all projections within the specified table. When using more than one schema, specify the schema that contains the table with the projections you want to delete, as noted in the syntax.
CATEGORY	Specifies the category of statistics to drop for the named [db-name.]schema-name.]table: ▪ 'BASE' (default) drops histograms and row counts (min/max column values, histogram. ▪ 'HISTOGRAMS' drops only the histograms. Row counts statistics remain. ▪ 'ALL' drops all statistics.

Privileges

- INSERT/UPDATE/DELETE privilege on table
- USAGE privilege on schema that contains the table

Notes

Once dropped, statistics can be time consuming to regenerate.

Example

The following command analyzes all statistics on the VMart schema database:

```
=> SELECT ANALYZE_STATISTICS('');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

This command drops base statistics for table store_sales_fact in the store schema:

```
=> SELECT DROP_STATISTICS('store.store_sales_fact', 'BASE');
 drop_statistics
-----------------
               0
(1 row)
```

Note that this command works the same as the previous command:

```
=> SELECT DROP_STATISTICS('store.store_sales_fact');
 DROP_STATISTICS
-----------------
               0
(1 row)
```

This command also drops statistics for all table projections:

```
=> SELECT DROP_STATISTICS ('');
 DROP_STATISTICS
-----------------
               0
(1 row)
```

For use cases, see Collecting Statistics in the Administrator's Guide

See Also

ANALYZE_STATISTICS (page 440)

EXPORT_STATISTICS (page 490)

IMPORT_STATISTICS (page 502)

DUMP_CATALOG

Returns an internal representation of the HP Vertica catalog. This function is used for diagnostic purposes.

Syntax

```
DUMP_CATALOG()
```

Privileges

None; however, function dumps only the objects visible to the user.

Notes

To obtain an internal representation of the HP Vertica catalog for diagnosis, run the query:

```
=> SELECT DUMP_CATALOG();
```

The output is written to the specified file:

```
\o /tmp/catalog.txt
SELECT DUMP_CATALOG();
\o
```

DUMP_LOCKTABLE

Returns information about deadlocked clients and the resources they are waiting for.

Syntax

```
DUMP_LOCKTABLE()
```

Privileges

None

Notes

Use DUMP_LOCKTABLE if HP Vertica becomes unresponsive:

1. Open an additional vsql connection.
2. Execute the query:

   ```
   => SELECT DUMP_LOCKTABLE();
   ```

   The output is written to vsql. See Monitoring the Log Files.

You can also see who is connected using the following command:

```
=> SELECT * FROM SESSIONS;
```

Close all sessions using the following command:

```
=>SELECT CLOSE_ALL_SESSIONS();
```

Close a single session using the following command:

How to close a single session:

```
=> SELECT CLOSE_SESSION('session_id');
```

You get the session_id value from the ***V_MONITOR.SESSIONS*** (page 1095) system table.

See Also

CLOSE_ALL_SESSIONS (page 461)

CLOSE_SESSION (page 458)

V_MONITOR.LOCKS (page 1037)

V_MONITOR.SESSIONS (page 1095)

DUMP_PARTITION_KEYS

Dumps the partition keys of all projections in the system.

Syntax

```
DUMP_PARTITION_KEYS( )
```

Note: ROS's of partitioned tables without partition keys are ignored by the tuple mover and are not merged during automatic tuple mover operations.

Privileges

None; however function dumps only the tables for which user has SELECT privileges.

Example

```
=> SELECT DUMP_PARTITION_KEYS( );
Partition keys on node v_vmart_node0001
  Projection 'states_b0'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: NH
   Storage [ROS container]
```

```
    No of partition keys: 1
    Partition keys: MA
  Projection 'states_b1'
   Storage [ROS container]
    No of partition keys: 1
    Partition keys: VT
   Storage [ROS container]
    No of partition keys: 1
    Partition keys: ME
   Storage [ROS container]
    No of partition keys: 1
    Partition keys: CT
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

V_MONITOR.PARTITIONS (page 1051)

Partitioning Tables in the Administrator's Guide

DUMP_PROJECTION_PARTITION_KEYS

Dumps the partition keys of the specified projection.

Syntax

```
DUMP_PROJECTION_PARTITION_KEYS(  'projection_name' )
```

Parameters

projection_name	Specifies the name of the projection.

Privileges

- SELECT privilege on table
- USAGE privileges on schema

Example

The following example creates a simple table called `states` and partitions the data by state:

```
=> CREATE TABLE states (
       year INTEGER NOT NULL,
       state VARCHAR NOT NULL)
```

```
      PARTITION BY state;


=> CREATE PROJECTION states_p (state, year) AS SELECT * FROM states
   ORDER BY state, year UNSEGMENTED ALL NODES;
```

Now dump the partition key of the specified projection:

```
=> SELECT DUMP_PROJECTION_PARTITION_KEYS( 'states_p_node0001' );
Partition keys on node helios_node0001
 Projection 'states_p_node0001'
 No of partition keys: 1
Partition keys on node helios_node0002
...
(1 row)
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

PROJECTIONS (page 961) system table

Partitioning Tables in the Administrator's Guide

DUMP_TABLE_PARTITION_KEYS

Dumps the partition keys of all projections anchored on the specified table.

Syntax

```
DUMP_TABLE_PARTITION_KEYS ( 'table_name' )
```

Parameters

table_name	Specifies the name of the table.

Privileges

- SELECT privilege on table
- USAGE privileges on schema

Example

The following example creates a simple table called `states` and partitions the data by state:

```
=> CREATE TABLE states (
       year INTEGER NOT NULL,
       state VARCHAR NOT NULL)
   PARTITION BY state;


=> CREATE PROJECTION states_p (state, year) AS SELECT * FROM states
   ORDER BY state, year UNSEGMENTED ALL NODES;
```

Now dump the partition keys of all projections anchored on table `states`:

```
=> SELECT DUMP_TABLE_PARTITION_KEYS( 'states' );
Partition keys on helios_node0001
Projection 'states_p_node0004'
No of partition keys: 1
Projection 'states_p_node0003'
No of partition keys: 1
Projection 'states_p_node0002'
No of partition keys: 1
Projection 'states_p_node0001'
No of partition keys: 1
Partition keys on helios_node0002
...
(1 row)
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 480)

DUMP_PROJECTION_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

Partitioning Tables in the Administrator's Guide

ENABLE_ELASTIC_CLUSTER

Enables elastic cluster scaling, which makes enlarging or reducing the size of your database cluster more efficient by segmenting a node's data into chunks that can be easily moved to other hosts.

Note: Databases created using HP Vertica Version 5.0 and later have elastic cluster enabled by default. You need to use this function on databases created before version 5.0 in order for them to use the elastic clustering feature.

Syntax

```
ENABLE_ELASTIC_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT ENABLE_ELASTIC_CLUSTER();
 ENABLE_ELASTIC_CLUSTER
------------------------
 ENABLED
(1 row)
```

See Also

- ***DISABLE_ELASTIC_CLUSTER*** (page 468)

ENABLE_LOCAL_SEGMENTS

Enables local storage segmentation, which breaks projections segments on nodes into containers that can be easily moved to other nodes. See Local Data Segmentation in the Administrator's Guide for more information.

Syntax

```
ENABLE_LOCAL_SEGMENTS()
```

Privileges

Must be a superuser

Example

```
=> SELECT ENABLE_LOCAL_SEGMENTS();
 ENABLE_LOCAL_SEGMENTS
-----------------------
 ENABLED
(1 row)
```

ENABLE_PROFILING

Enables profiling for the profiling type you specify.

Note: HP Vertica stores profiled data is in memory, so depending on how much data you collect, profiling could be memory intensive.

Syntax

```
ENABLE_PROFILING( 'profiling-type' )
```

Parameters

profiling-type	The type of profiling data you want to enable. Can be one of: ▪ session—enables profiling for basic session parameters and lock time out data ▪ query—enables profiling for general information about queries that ran, such as the query strings used and the duration of queries ▪ ee—enables profiling for information about the execution run of each query

Example

The following statement enables profiling on query execution runs:

```
=> SELECT ENABLE_PROFILING('ee');
   ENABLE_PROFILING
----------------------
 EE Profiling Enabled
(1 row)
```

See also

CLEAR_PROFILING (page 455)

DISABLE_PROFILING (page 469)

Profiling Database Performance in the Administrator's Guide

EVALUATE_DELETE_PERFORMANCE

Evaluates projections for potential ***DELETE*** (page 807) performance issues. If there are issues found, a warning message is displayed. For steps you can take to resolve delete and update performance issues, see Optimizing Deletes and Updates for Performance in the Administrator's Guide. This function uses data sampling to determine whether there are any issues with a projection. Therefore, it does not generate false-positives warnings, but it can miss some cases where there are performance issues.

Note: Optimizing for delete performance is the same as optimizing for update performance. So, you can use this function to help optimize a projection for updates as well as deletes.

Syntax

```
EVALUATE_DELETE_PERFORMANCE ( 'target' )
```

Parameters

target	The name of a projection or table. If you supply the name of a projection, only that projection is evaluated for DELETE performance issues. If you supply the name of a table, then all of the projections anchored to the table will be evaluated for issues. If you do not provide a projection or table name, EVALUATE_DELETE_PERFORMANCE examines all of the projections that you can access for DELETE performance issues. Depending on the size you your database, this may take a long time.

Privileges

None

Note: When evaluating multiple projections, EVALUATE_DELETE_PERFORMANCE reports up to ten projections that have issues, and refers you to a table that contains the full list of issues it has found.

Example

The following example demonstrates how you can use EVALUATE_DELETE_PERFORMANCE to evaluate your projections for slow DELETE performance.

```
=> create table example (A int, B int,C int);
CREATE TABLE
=> create projection one_sort (A,B,C) as (select A,B,C from example) order by A;
CREATE PROJECTION
=> create projection two_sort (A,B,C) as (select A,B,C from example) order by A,B;
CREATE PROJECTION
=> select evaluate_delete_performance('one_sort');
            evaluate_delete_performance
-----------------------------------------------
 No projection delete performance concerns found.

(1 row)

=> select evaluate_delete_performance('two_sort');
            evaluate_delete_performance
-----------------------------------------------
 No projection delete performance concerns found.

(1 row)
```

The previous example showed that there was no structural issue with the projection that would cause poor DELETE performance. However, the data contained within the projection can create potential delete issues if the sorted columns do not uniquely identify a row or small number of rows. In the following example, Perl is used to populate the table with data using a nested series of loops. The inner loop populates column C, the middle loop populates column B, and the outer loop populates column A. The result is column A contains only three distinct values (0, 1, and 2), while column B slowly varies between 20 and 0 and column C changes in each row. EVALUATE_DELETE_PERFORMANCE is run against the projections again to see if the data within the projections causes any potential DELETE performance issues.

```
=> \! perl -e 'for ($i=0; $i<3; $i++) { for ($j=0; $j<21; $j++) { for ($k=0; $k<19; $k++) { printf
"%d,%d,%d\n", $i,$j,$k;}}}' | /opt/vertica/bin/vsql -c "copy example from stdin delimiter ',' direct;"
Password:

=> select * from example;
```

```
 A | B  | C
---+----+----
 0 | 20 | 18
 0 | 20 | 17
 0 | 20 | 16
 0 | 20 | 15
 0 | 20 | 14
 0 | 20 | 13
 0 | 20 | 12
 0 | 20 | 11
 0 | 20 | 10
 0 | 20 |  9
 0 | 20 |  8
 0 | 20 |  7
 0 | 20 |  6
 0 | 20 |  5
 0 | 20 |  4
 0 | 20 |  3
 0 | 20 |  2
 0 | 20 |  1
 0 | 20 |  0
 0 | 19 | 18
 1157 rows omitted
 2 |  1 |  0
 2 |  0 | 18
 2 |  0 | 17
 2 |  0 | 16
 2 |  0 | 15
 2 |  0 | 14
 2 |  0 | 13
 2 |  0 | 12
 2 |  0 | 11
 2 |  0 | 10
 2 |  0 |  9
 2 |  0 |  8
 2 |  0 |  7
 2 |  0 |  6
 2 |  0 |  5
 2 |  0 |  4
 2 |  0 |  3
 2 |  0 |  2
 2 |  0 |  1
 2 |  0 |  0


=> SELECT COUNT (*) FROM example;
 COUNT
-------
  1197
(1 row)

=> SELECT COUNT (DISTINCT A) FROM example;
 COUNT
-------
     3
(1 row)


=> select evaluate_delete_performance('one_sort');
            evaluate_delete_performance
-----------------------------------------------
 Projection exhibits delete performance concerns.

(1 row)

release=> select evaluate_delete_performance('two_sort');
            evaluate_delete_performance
```

```
-----------------------------------------------
 No projection delete performance concerns found.

(1 row)
```

The one_sort projection has potential delete issues since it only sorts on column A which has few distinct values. This means that each value in the sort column corresponds to many rows in the projection, which negatively impacts DELETE performance. Since the two_sort projection is sorted on columns A and B, each combination of values in the two sort columns identifies just a few rows, allowing deletes to be performed faster.

Not supplying a projection name results in all of the projections you can access being evaluated for DELETE performance issues.

```
=> select evaluate_delete_performance();
                                                    evaluate_delete_performance
-------------------------------------------------------------------------------------------------------------
-----------------------------
 The following projection exhibits delete performance concerns:
        "public"."one_sort"
See v_catalog.projection_delete_concerns for more details.

(1 row)
```

EXPORT_CATALOG

Generates a SQL script that you can use to recreate a physical schema design in its current state on a different cluster. This function always attempts to recreate projection statements with KSAFE clauses, if they exist in the original definitions, or OFFSET clauses if they do not.

Syntax

```
EXPORT_CATALOG ( [ 'destination' ] , [ 'scope' ] )
```

Parameters

destination	Specifies the path and name of the SQL output file. An empty string (''), which is the default, outputs the script to standard output. The function writes the script to the catalog directory if no destination is specified. If you specify a file that does not exist, the function creates one. If the file pre-exists, the function silently overwrites its contents.

scope	Determines what to export: ▪ DESIGN — Exports schemas, tables, constraints, views, and projections to which the user has access. This is the default value. ▪ DESIGN_ALL — Exports all the design objects plus system objects created in Database Designer (for example, design contexts and their tables). The objects that are exported are those to which the user has access. ▪ TABLES — Exports all tables, constraints, and projections for for which the user has permissions. See also ***EXPORT_TABLES*** (page 491).

Privileges

None; however:

- Function exports only the objects visible to the user
- Only a superuser can export output to file

Example

The following example exports the design to standard output:

```
=> SELECT EXPORT_CATALOG('','DESIGN');
```

See Also

EXPORT_OBJECTS

EXPORT_TABLES (page 491)

Exporting the Catalog in the Administrator's Guide

EXPORT_OBJECTS

Generates a SQL script you can use to recreate catalog objects on a different cluster. The generated script includes only the non-virtual objects to which the user has access. The function exports catalog objects in order dependency for correct recreation. Running the generated SQL script on another cluster then creates all referenced objects before their dependent objects.

Note: You cannot use EXPORT_OBJECTS to export a view without its dependencies.

The EXPORT_OBJECTS function always attempts to recreate projection statements with KSAFE clauses, if they existed in the original definitions, or OFFSET clauses, if they did not.

None of the EXPORT_OBJECTS parameters accepts a NULL value as input. EXPORT_OBJECTS returns an error if an explicitly-specified object does not exist, or the user does not have access to the object.

Syntax

```
EXPORT_OBJECTS( [ 'destination' ] , [ 'scope' ] , [ 'ksafe' ] )
```

Parameters

destination	Specifies the path and name of the SQL output file. The default empty string (' ') outputs the script contents to standard output. Non-DBadmin users can specify only an empty string. If you specify a file that does not exist, the function creates one. If the file pre-exists, the function silently overwrites its contents. If you do not specify an explicit path destination, the function outputs the script for the exported objects to the catalog directory.
scope	Determines which catalog objects to export, where you specify *scope* as follows: ▪ An empty string (' ')—exports all non-virtual objects to which the user has access, including constraints. (Note that constraints are not objects that can be passed as individual arguments.) An empty string is the default *scope* value for scope if you do not limit the export. ▪ A comma-delimited list of catalog objects to export, which can include the following: ▪ —' [*dbname*.]*schema.object* '— matches each named schema object. You can optionally qualify the schema with a database prefix. A named schema object can be a table, projection, view, sequence, or user-defined SQL function. ▪ —' [*dbname*.]*schema* — matches the named schema, which you can optionally qualify with a database prefix. For a schema, HP Vertica exports all non-virtual objects that the user has access to within the schema. If a schema and table have the same name, the schema takes precedence.
ksafe	Specifies whether to incorporate a MARK_DESIGN_KSAFE statement with the correct K-safe value for the database. The statement is placed at the end of the output script. Use one of the following: ▪ true—adds the MARK_DESIGN_KSAFE statement to the script. This is the default value. ▪ false—omits the MARK_DESIGN_KSAFE statement from the script. Adding the MARK_DESIGN_KSAFE statement is useful if you plan to import the SQL script into a new database, and you want the new database to inherit the K-safety value of the original database.

Privileges

None; however:

- Function exports only the objects visible to the user
- Only a superuser can export output to a file

Example

The following example exports all the non-virtual objects to which the user has access to standard output. The example uses `false` for the last parameter, indicating that the file will not include the `MARK_DESIGN_KSAFE` statement at the end.

```
=> SELECT EXPORT_OBJECTS(' ',' ',false);
```

EXPORT_STATISTICS

Generates an XML file that contains statistics for the database. You can optionally export statistics on a single database object (table, projection, or table column).

Before you export statistics for the database, run ***ANALYZE_STATISTICS()*** (page 440) to automatically collect the most up to date statistics information.

Note: Use the second argument only if statistics in the database do not match the statistics of data.

Syntax

```
EXPORT_STATISTICS
[ ( 'destination' )
... | ( '[ [ db-name.]schema.]table [.column-name ]' ) ]
```

Parameters

destination	Specifies the path and name of the XML output file. An empty string returns the script to the screen.
[[*db-name*.]*schema*.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the name of the table and exports statistics for all projections of that table. **Note:** If you are using more than one schema, specify the schema that contains the projection, as noted as noted in the [[*db-name*.]*schema*.] entry.
[.*column-name*]	[Optional] Specifies the name of a single column, typically a predicate column. Using this option with a table specification lets you export statistics for only that column.

Privileges

Must be a superuser

Examples

The following command exports statistics on the VMart example database to a file:

```
vmart=> SELECT EXPORT_STATISTICS('/vmart/statistics/vmart_stats.xml');
         export_statistics
-----------------------------------
 Statistics exported successfully

(1 row)
```

The next statement exports statistics on a single column (price) from a table called food:

```
=> SELECT EXPORT_STATISTICS('/vmart/statistics/price.xml', 'food.price');
```

See Also

ANALYZE_STATISTICS (page 440)

DROP_STATISTICS (page 476)

IMPORT_STATISTICS (page 502)

Collecting Statistics in the Administrator's Guide

EXPORT_TABLES

Generates a SQL script that can be used to recreate a logical schema (schemas, tables, constraints, and views) on a different cluster.

Syntax

```
EXPORT_TABLES ( [ 'destination' ] , [ 'scope' ] )
```

Parameters

destination	Specifies the path and name of the SQL output file. An empty string (''), which is the default, outputs the script to standard output. The function writes the script to the catalog directory if no destination is specified. If you specify a file that does not exist, the function creates one. If the file pre-exists, the function silently overwrites its contents.

scope	Determines the tables to export. Specify the scope as follows: ▪ An empty string (' ') — exports all non-virtual table objects to which the user has access, including table schemas, sequences, and constraints. Exporting all non-virtual objects is the default scope, and what the function exports if you do not specify a scope. ▪ A comma-delimited list of objects, which can include the following: ▪ —' [*dbname*.][*schema*.]*object* '— matches the named objects, which can be schemas, tables, or views, in the schema. You can optionally qualify a schema with a database prefix, and objects with a schema. You cannot pass constraints as individual arguments. ▪ —' [*dbname*.]*object* '— matches a named object, which can be a schema, table, or view. You can optionally qualify a schema with a database prefix, and an object with its schema. For a schema, HP Vertica exports all non-virtual objects to which the user has access within the schema. If a schema and table both have the same name, the schema takes precedence.

Privileges

None; however:

- Function exports only the objects visible to the user
- Only a superuser can export output to file

Example

The following example exports the `store_orders_fact` table of the `store` schema (in the current database) to standard output:

```
=> SELECT EXPORT_TABLES(' ','store.store_orders_fact');
```

EXPORT_TABLES returns an error if:

- You explicitly specify an object that does not exist
- The current user does not have access to a specified object

See Also

EXPORT_CATALOG (page 487)

EXPORT_OBJECTS

Exporting Tables in the Administrator's Guide

FLUSH_DATA_COLLECTOR

Waits until memory logs are moved to disk and then flushes the Data Collector, synchronizing the log with the disk storage. A superuser can flush Data Collector information for an individual component or for all components.

Syntax

```
FLUSH_DATA_COLLECTOR( [ 'component' ] )
```

Parameters

component	Flushes the specified component. If you provide no argument, the function flushes the Data Collector in full. For the current list of component names, query the V_MONITOR.DATA_COLLECTOR system table.

Privileges

Must be a superuser

Example

The following command flushes the Data Collector for the ResourceAcquisitions component:

```
=> SELECT flush_data_collector('ResourceAcquisitions');
 flush_data_collector
----------------------
 FLUSH
(1 row)
```

The following command flushes data collection for all components:

```
=> SELECT flush_data_collector();
 flush_data_collector
----------------------
 FLUSH
(1 row)
```

See Also

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining Monitoring Information in the Administrator's Guide

GET_AHM_EPOCH

Returns the number of the epoch in which the Ancient History Mark is located. Data deleted up to and including the AHM epoch can be purged from physical storage.

Syntax

```
GET_AHM_EPOCH()
```

Note: The AHM epoch is 0 (zero) by default (purge is disabled).

Privileges

None

Examples

```
SELECT GET_AHM_EPOCH();
    get_ahm_epoch
----------------------
 Current AHM epoch: 0
(1 row)
```

GET_AHM_TIME

Returns a TIMESTAMP value representing the Ancient History Mark. Data deleted up to and including the AHM epoch can be purged from physical storage.

Syntax

```
GET_AHM_TIME()
```

Privileges

None

Examples

```
SELECT GET_AHM_TIME();
                  GET_AHM_TIME
-----------------------------------------------
 Current AHM Time: 2010-05-13 12:48:10.532332-04
(1 row)
```

See Also

SET DATESTYLE (page 903) for information about valid ***TIMESTAMP*** (page 97) values.

GET_COMPLIANCE_STATUS

Displays whether your database is in compliance with your HP Vertica license agreement. This information includes the results of HP Vertica's most recent audit of the database size (if your license has a data allowance as part of its terms), and the license term (if your license has an end date).

The information displayed by GET_COMPLIANCE_STATUS includes:

- The estimated size of the database (see How HP Vertica Calculates Database Size in the Administrator's Guide for an explanation of the size estimate).
- The raw data size allowed by your HP Vertica license.
- The percentage of your allowance that your database is currently using.
- The date and time of the last audit.
- Whether your database complies with the data allowance terms of your license agreement.

- The end date of your license.
- How many days remain until your license expires.

Note: If your license does not have a data allowance or end date, some of the values may not appear in the output for GET_COMPLIANCE_STATUS.

If the audit shows your license is not in compliance with your data allowance, you should either delete data to bring the size of the database under the licensed amount, or upgrade your license. If your license term has expired, you should contact HP immediately to renew your license. See Managing Your License Key in the Administrator's Guide for further details.

Syntax

```
GET_COMPLIANCE_STATUS()
```

Privileges

None

Example

```
                                GET_COMPLIANCE_STATUS
--------------------------------------------------------------------------------
---
 Raw Data Size: 2.00GB +/- 0.003GB
 License Size : 4.000GB
 Utilization  : 50%
 Audit Time   : 2011-03-09 09:54:09.538704+00
 Compliance Status : The database is in compliance with respect to raw data size.

 License End Date: 04/06/2011
 Days Remaining: 28.59
(1 row)
```

GET_AUDIT_TIME

Reports the time when the automatic audit of database size occurs. HP Vertica performs this audit if your HP Vertica license includes a data size allowance. For details of this audit, see Managing Your License Key in the Administrator's Guide. To change the time the audit runs, use the ***SET_AUDIT_TIME*** (page 530) function.

Syntax

```
GET_AUDIT_TIME()
```

Privileges

None

Example

```
=> SELECT get_audit_time();
                   get_audit_time
-----------------------------------------------------
```

```
 The audit is scheduled to run at 11:59 PM each day.
(1 row)
```

GET_CURRENT_EPOCH

Returns the number of the current epoch. The epoch into which data (COPY, INSERT, UPDATE, and DELETE operations) is currently being written. The current epoch advances automatically every three minutes.

Syntax

```
GET_CURRENT_EPOCH()
```

Privileges

None

Examples

```
SELECT GET_CURRENT_EPOCH();
 GET_CURRENT_EPOCH
-------------------
               683
(1 row)
```

GET_DATA_COLLECTOR_POLICY

Retrieves a brief statement about the retention policy for the specified component.

Syntax

```
GET_DATA_COLLECTOR_POLICY( 'component' )
```

Parameters

component	Returns the retention policy for the specified component. For a current list of component names, query the V_MONITOR.DATA_COLLECTOR system table

Privileges

None

Example

The following query returns the history of all resource acquisitions by specifying the `ResourceAcquisitions` component:

```
=> SELECT get_data_collector_policy('ResourceAcquisitions');
          get_data_collector_policy
-----------------------------------------------
```

```
 1000KB kept in memory, 10000KB kept on disk.
(1 row)
```

See Also

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining Monitoring Information in the Administrator's Guide

GET_LAST_GOOD_EPOCH

Returns the number of the last good epoch. A term used in manual recovery, LGE (Last Good Epoch) refers to the most recent epoch that can be recovered.

Syntax

```
GET_LAST_GOOD_EPOCH()
```

Privileges

None

Examples

```
SELECT GET_LAST_GOOD_EPOCH();
 GET_LAST_GOOD_EPOCH
---------------------
                 682
(1 row)
```

GET_NUM_ACCEPTED_ROWS

Returns the number of rows loaded into the database for the last completed load for the current session. GET_NUM_ACCEPTED_ROWS is a meta-function. Do not use it as a value in an INSERT query.

The number of accepted rows is not available for a load that is currently in process. Check the ***LOAD_STREAMS*** (page 1031) system table for its status.

Also, this meta-function supports only loads from STDIN or a single file on the initiator. You cannot use GET_NUM_ACCEPTED_ROWS for multi-node loads.

Syntax

```
GET_NUM_ACCEPTED_ROWS();
```

Privileges

None

NOTE: The data regarding accepted rows from the last load during the current session does not persist, and is lost when you initiate a new load.

See Also

GET_NUM_REJECTED_ROWS (page 498)

GET_NUM_REJECTED_ROWS

Returns the number of rows that were rejected during the last completed load for the current session. GET_NUM_REJECTED_ROWS is a meta-function. Do not use it as a value in an INSERT query.

Rejected row information is unavailable for a load that is currently running. The number of rejected rows is not available for a load that is currently in process. Check the ***LOAD_STREAMS*** (page 1031) system table for its status.

Also, this meta-function supports only loads from STDIN or a single file on the initiator. You cannot use GET_NUM_REJECTED_ROWS for multi-node loads.

Syntax

```
GET_NUM_REJECTED_ROWS();
```

Privileges

None

Note: The data regarding rejected rows from the last load during the current session does not persist, and is dropped when you initiate a new load.

See Also

GET_NUM_ACCEPTED_ROWS (page 497)

GET_PROJECTION_STATUS

Returns information relevant to the status of a projection.

Syntax

```
GET_PROJECTION_STATUS ( '[[db-name.]schema-name.]projection' );
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection	Is the name of the projection for which to display status. When using more than one schema, specify the schema that contains the projection, as noted above.

Privileges

None

Description

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 the projection is segmented
- The number and names of buddy projections
- Whether the projection is safe
- Whether the projection is up-to-date
- Whether statistics have been computed for the projection

Notes

- You can use GET_PROJECTION_STATUS to monitor the progress of a projection data refresh. See ***ALTER PROJECTION*** (page 659).
- To view a list of the nodes in a database, use the View Database Command in the Administration Tools.

Examples

```
=> SELECT GET_PROJECTION_STATUS('public.customer_dimension_site01');

                                    GET_PROJECTION_STATUS
-------------------------------------------------------------------------------------------------
 Current system K is 1.
# of Nodes: 4.
public.customer_dimension_site01 [Segmented: No] [Seg Cols: ] [K: 3]
[public.customer_dimension_site04, public.customer_dimension_site03,
public.customer_dimension_site02] [Safe: Yes] [UptoDate: Yes][Stats: Yes]
```

See Also

ALTER PROJECTION (page 659)

GET_PROJECTIONS (page 499)

GET_PROJECTIONS, GET_TABLE_PROJECTIONS

Note: This function was formerly named GET_TABLE_PROJECTIONS(). HP Vertica still supports the former function name.

Returns information relevant to the status of a table:

- The current K-safety status of the database
- The number of sites (nodes) in the database
- The number of projections for which the specified table is the anchor table
- For each projection:
  - The projection's buddy projections

- Whether the projection is segmented
- Whether the projection is safe
- Whether the projection is up-to-date

Syntax

```
GET_PROJECTIONS ( '[[db-name.]schema-name.]table' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table	Is the name of the table for which to list projections. When using more than one schema, specify the schema that contains the table.

Privileges

None

Notes

- You can use GET_PROJECTIONS to monitor the progress of a projection data refresh. See ***ALTER PROJECTION*** (page 659).
- To view a list of the nodes in a database, use the View Database Command in the Administration Tools.

Examples

The following example gets information about the store_dimension table in the VMart schema:

```
=> SELECT GET_PROJECTIONS('store.store_dimension');
-------------------------------------------------------------------------------
Current system K is 1.
# of Nodes: 4.
Table store.store_dimension has 4 projections.

Projection Name: [Segmented] [Seg Cols] [# of Buddies] [Buddy Projections] [Safe] [UptoDate]
-----------------------------------------------------
store.store_dimension_node0004 [Segmented: No] [Seg Cols: ] [K: 3] [store.store_dimension_node0003,
store.store_dimension_node0002, store.store_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
store.store_dimension_node0003 [Segmented: No] [Seg Cols: ] [K: 3] [store.store_dimension_node0004,
store.store_dimension_node0002, store.store_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
store.store_dimension_node0002 [Segmented: No] [Seg Cols: ] [K: 3] [store.store_dimension_node0004,
store.store_dimension_node0003, store.store_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
```

```
store.store_dimension_node0001 [Segmented: No] [Seg Cols: ] [K: 3] [store.store_dimension_node0004,
store.store_dimension_node0003, store.store_dimension_node0002] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]

(1 row)
```

See Also

ALTER PROJECTION (page 659)

GET_PROJECTION_STATUS (page 498)

HAS_ROLE

Indicates, by a boolean value, whether a role has been assigned to a user. This function is useful for letting you check your own role membership.

Behavior Type

Stable

Syntax 1

```
HAS_ROLE( [ 'user_name' ,] 'role_name' );
```

Syntax 2

```
HAS_ROLE( 'role_name' );
```

Parameters

user_name	[Optional] The name of a user to look up. Currently, only a superuser can supply the user_name argument.
role_name	The name of the role you want to verify has been granted.

Privileges

Users can check their own role membership by calling HAS_ROLE('role_name'), but only a superuser can look up other users' memberships using the optional user_name parameter.

Notes

You can query V_CATALOG system tables ***ROLES*** (page 967), ***GRANTS*** (page 944), and ***USERS*** (page 985) to show any directly-assigned roles; however, these tables do not indicate whether a role is available to a user when roles may be available through other roles (indirectly).

Examples

User Bob wants to see if he has been granted the commentor role:

```
=> SELECT HAS_ROLE('commentor');
```

Output *t* for true indicates that Bob has been assigned the commentor role:

```
 HAS_ROLE
----------
```

```
 t
(1 row)
```

In the following function call, a superuser checks if the logadmin role has been granted to user Bob:

```
=> SELECT HAS_ROLE('Bob', 'logadmin');
 HAS_ROLE
----------
 t
(1 row)
```

To view the names of all roles users can access, along with any roles that have been assigned to those roles, query the ***V_CATALOG.ROLES*** (page 967) system table. An asterisk in the output means role granted WITH ADMIN OPTION.

```
=> SELECT * FROM roles;
      name        |    assigned_roles
------------------+----------------------
 public           |
 dbadmin          | dbduser*
 pseudosuperuser | dbadmin
 dbduser          |
 logreader        |
 logwriter        |
 logadmin         | logreader, logwriter
(7 rows)
```

Note: The dbduser role in output above is internal only; you can ignore it.

See Also

GRANTS (page 944)

ROLES (page 967)

USERS (page 985)

Managing Privileges and Roles and Viewing a User's Role in the Administrator's Guide

IMPORT_STATISTICS

Imports statistics from the XML file generated by the EXPORT_STATISTICS command.

Syntax

```
IMPORT_STATISTICS ( 'destination' )
```

Parameters

destination	Specifies the path and name of the XML input file (which is the output of EXPORT_STATISTICS function).

Privileges

Must be a superuser

Notes

- Imported statistics override existing statistics for all projections on the specified table.
- For use cases, see Collecting Statistics in the Administrator's Guide

See Also

ANALYZE_STATISTICS (page 440)

DROP_STATISTICS (page 476)

EXPORT_STATISTICS (page 490)

INTERRUPT_STATEMENT

Interrupts the specified statement (within an external session), rolls back the current transaction, and writes a success or failure message to the log file.

Syntax

```
INTERRUPT_STATEMENT( 'session_id ', statement_id )
```

Parameters

session_id	Specifies the session to interrupt. This identifier is unique within the cluster at any point in time.
statement_id	Specifies the statement to interrupt

Privileges

Must be a superuser

Notes

- Only statements run by external sessions can be interrupted.
- Sessions can be interrupted during statement execution.
- If the *statement_id* is valid, the statement is interruptible. The command is successfully sent and returns a success message. Otherwise the system returns an error.

Messages

The following list describes messages you might encounter:

Message	Meaning
`Statement interrupt sent. Check SESSIONS for progress.`	This message indicates success.

`Session <id> could not be successfully interrupted: session not found.`	The session ID argument to the interrupt command does not match a running session.
`Session <id> could not be successfully interrupted: statement not found.`	The statement ID does not match (or no longer matches) the ID of a running statement (if any).
`No interruptible statement running`	The statement is DDL or otherwise non-interruptible.
`Internal (system) sessions cannot be interrupted.`	The session is internal, and only statements run by external sessions can be interrupted.

Examples

Two user sessions are open. RECORD 1 shows user session running `SELECT FROM SESSION`, and RECORD 2 shows user session running `COPY DIRECT`:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1
]--------------+-----------------------------------------------------
node_name                   | v_vmartdb_node0001
user_name                   | dbadmin
client_hostname             | 127.0.0.1:52110
client_pid                  | 4554
login_timestamp             | 2011-01-03 14:05:40.252625-05
session_id                  | stress04-4325:0x14
client_label                |
transaction_start           | 2011-01-03 14:05:44.325781
transaction_id              | 45035996273728326
transaction_description     | user dbadmin (select * from sessions;)
statement_start             | 2011-01-03 15:36:13.896288
statement_id                | 10
last_statement_duration_us | 14978
current_statement           | select * from sessions;
ssl_state                   | None
authentication_method       | Trust
-[ RECORD 2
]--------------+-----------------------------------------------------
node_name                   | v_vmartdb_node0003
user_name                   | dbadmin
client_hostname             | 127.0.0.1:56367
client_pid                  | 1191
login_timestamp             | 2011-01-03 15:31:44.939302-05
session_id                  | stress06-25663:0xbec
client_label                |
transaction_start           | 2011-01-03 15:34:51.05939
transaction_id              | 54043195528458775
transaction_description     | user dbadmin (COPY Mart_Fact FROM
'/data/Mart_Fact.tbl'
                              DELIMITER '|' NULL '\\n' DIRECT;)
statement_start             | 2011-01-03 15:35:46.436748
statement_id                | 5
```

```
last_statement_duration_us | 1591403
current_statement          | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER
'|'
                             NULL '\\n' DIRECT;
ssl_state                  | None
authentication_method      | Trust
```

Interrupt the COPY DIRECT statement running in stress06-25663:0xbec:

```
=> \x
Expanded display is off.
=> SELECT INTERRUPT_STATEMENT('stress06-25663:0x1537', 5);
                      interrupt_statement
-----------------------------------------------------------------
 Statement interrupt sent. Check v_monitor.sessions for progress.
(1 row)
```

Verify that the interrupted statement is no longer active by looking at the current_statement column in the SESSIONS system table. This column becomes blank when the statement has been interrupted:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1
]-------------+-----------------------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
client_pid                 | 4554
login_timestamp            | 2011-01-03 14:05:40.252625-05
session_id                 | stress04-4325:0x14
client_label               |
transaction_start          | 2011-01-03 14:05:44.325781
transaction_id             | 45035996273728326
transaction_description    | user dbadmin (select * from sessions;)
statement_start            | 2011-01-03 15:36:13.896288
statement_id               | 10
last_statement_duration_us | 14978
current_statement          | select * from sessions;
ssl_state                  | None
authentication_method      | Trust
-[ RECORD 2
]-------------+-----------------------------------------------------
node_name                  | v_vmartdb_node0003
user_name                  | dbadmin
client_hostname            | 127.0.0.1:56367
client_pid                 | 1191
login_timestamp            | 2011-01-03 15:31:44.939302-05
session_id                 | stress06-25663:0xbec
client_label               |
transaction_start          | 2011-01-03 15:34:51.05939
transaction_id             | 54043195528458775
transaction_description    | user dbadmin (COPY Mart_Fact FROM
'/data/Mart_Fact.tbl'
                             DELIMITER '|' NULL '\\n' DIRECT;)
statement_start            | 2011-01-03 15:35:46.436748
```

```
statement_id                | 5
last_statement_duration_us  | 1591403
current_statement           |
ssl_state                   | None
authentication_method       | Trust
```

See Also

SESSIONS (page 1095)

Managing Sessions and Configuration Parameters in the Administrator's Guide

INSTALL_LICENSE

Installs the license key in the global catalog.

description

Syntax

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

Parameters

filename	specifies the absolute pathname of a valid license file.

Privileges

Must be a superuser

Notes

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

Examples

```
=> SELECT INSTALL_LICENSE('/tmp/vlicense.dat');
```

LAST_INSERT_ID

Returns the last value of a column whose value is automatically incremented through the AUTO_INCREMENT or IDENTITY ***column-constraint*** (page 783). If multiple sessions concurrently load the same table, the returned value is the last value generated for an AUTO_INCREMENT column by an insert in that session.

Behavior Type

Volatile

Syntax

```
LAST_INSERT_ID()
```

Privileges

- Table owner
- USAGE privileges on schema

Notes

- This function works only with AUTO_INCREMENT and IDENTITY columns. See ***column-constraints*** (page 783) for the ***CREATE TABLE*** (page 770) statement.
- LAST_INSERT_ID does not work with sequence generators created through the ***CREATE SEQUENCE*** (page 765) statement.

Examples

Create a sample table called `customer4`.

```
=> CREATE TABLE customer4(
     ID IDENTITY(2,2),
     lname VARCHAR(25),
     fname VARCHAR(25),
     membership_card INTEGER
   );
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Gupta', 'Saleem', 475987);
```

Notice that the IDENTITY column has a seed of 2, which specifies the value for the first row loaded into the table, and an increment of 2, which specifies the value that is added to the IDENTITY value of the previous row.

Query the table you just created:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
(1 row)
```

Insert some additional values:

```
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Lee', 'Chen', 598742);
```

Call the LAST_INSERT_ID function:

```
=> SELECT LAST_INSERT_ID();
last_insert_id
----------------
             4
(1 row)
```

Query the table again:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
  4 | Lee   | Chen   |          598742
```

```
(2 rows)
```

Add another row:

```
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Davis', 'Bill', 469543);
```

Call the LAST_INSERT_ID function:

```
=> SELECT LAST_INSERT_ID();
 LAST_INSERT_ID
----------------
              6
(1 row)
```

Query the table again:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
  4 | Lee   | Chen   |          598742
  6 | Davis | Bill   |          469543
(3 rows)
```

See Also

ALTER SEQUENCE (page 669)

CREATE SEQUENCE (page 765)

DROP SEQUENCE (page 822)

V_CATALOG.SEQUENCES (page 969)

Using Sequences and Sequence Privileges in the Administrator's Guide

MAKE_AHM_NOW

Sets the Ancient History Mark (AHM) to the greatest allowable value, and lets you drop any projections that existed before the issue occurred.

Caution: This function is intended for use by Administrators only.

Syntax

```
MAKE_AHM_NOW ( [ true ] )
```

Parameters

true	[Optional] Allows AHM to advance when nodes are down. **Note:** If the AHM is advanced after the last good epoch of the failed nodes, those nodes must recover all data from scratch. Use with care.

Privileges

Must be a superuser

Notes

- The MAKE_AHM_NOW function performs the following operations:
  - Advances the epoch.
  - Performs a moveout operation on all projections.
  - Sets the AHM to LGE — at least to the current epoch at the time MAKE_AHM_NOW() was issued.
- All history is lost and you cannot perform historical queries prior to the current epoch.

Example

```
=> SELECT MAKE_AHM_NOW();
         MAKE_AHM_NOW
------------------------------
 AHM set (New AHM Epoch: 683)
(1 row)
```

The following command allows the AHM to advance, even though node 2 is down:

```
=> SELECT MAKE_AHM_NOW(true);
WARNING:  Received no response from v_vmartdb_node0002 in get cluster LGE
WARNING:  Received no response from v_vmartdb_node0002 in get cluster LGE
WARNING:  Received no response from v_vmartdb_node0002 in set AHM
         MAKE_AHM_NOW
------------------------------
 AHM set (New AHM Epoch: 684)
(1 row)
```

See Also

DROP PROJECTION (page 818)

MARK_DESIGN_KSAFE (page 510)

SET_AHM_EPOCH (page 527)

SET_AHM_TIME (page 528)

MARK_DESIGN_KSAFE

Enables or disables high availability in your environment, in case of a failure. Before enabling recovery, MARK_DESIGN_KSAFE queries the catalog to determine whether a cluster's physical schema design meets the following requirements:

- Small, unsegmented tables are replicated on all nodes.
- Large table superprojections are segmented with each segment on a different node.
- Each large table projection has at least one buddy projection for K-safety=1 (or two buddy projections for K-safety=2).

  Buddy projections are also segmented across database nodes, but the distribution is modified so that segments that contain the same data are distributed to different nodes. See High Availability Through Projections in the Concepts Guide.

  Note: 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

```
MARK_DESIGN_KSAFE ( k )
```

Parameters

k	2 enables high availability if the schema design meets requirements for K-safety=2 1 enables high availability if the schema design meets requirements for K-safety=1 0 disables high availability

If you specify a *k* value of one (1) or two (2), HP Vertica returns one of the following messages.

Success:

```
Marked design n-safe
```

Failure:

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

n in the message is 1 or 2 and represents the *k* value.

Privileges

Must be a superuser

Notes

- The database's internal recovery state persists across database restarts but it is not checked at startup time.
- If a database has automatic recovery enabled, you must temporarily disable automatic recovery before creating a new table.

- When one node fails on a system marked K-safe=1, the remaining nodes are available for DML operations.

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)
```

See Also

SYSTEM (page 1111)

High Availability and Recovery in the Concepts Guide

SQL System Tables (Monitoring APIs) (page 933) topic in the Administrator's Guide

Using Identically Segmented Projections in the Programmer's Guide

Failure Recovery in the Administrator's Guide

MEASURE_LOCATION_PERFORMANCE

Measures disk performance for the location specified.

Syntax

```
MEASURE_LOCATION_PERFORMANCE ( 'path' , 'node' )
```

Parameters

path	Specifies where the storage location to measure is mounted.
node	Is the HP Vertica node where the location to be measured is available.

Privileges

Must be a superuser

Notes

- To get a list of all node names on your cluster, query the ***V_MONITOR.DISK_STORAGE*** (page 1014) system table:

```
=> SELECT node_name from DISK_STORAGE;
   node_name
------------------
 v_vmartdb_node0004
 v_vmartdb_node0004
 v_vmartdb_node0005
 v_vmartdb_node0005
 v_vmartdb_node0006
 v_vmartdb_node0006
(6 rows)
```

- If you intend to create a tiered disk architecture in which projections, columns, and partitions are stored on different disks based on predicted or measured access patterns, you need to measure storage location performance for each location in which data is stored. You do not need to measure storage location performance for temp data storage locations because temporary files are stored based on available space.
- The method of measuring storage location performance applies only to configured clusters. If you want to measure a disk before configuring a cluster see Measuring Location Performance.
- Storage location performance equates to the amount of time it takes to read and write 1MB of data from the disk. This time equates to:

  IO time = Time to read/write 1MB + Time to seek = 1/Throughput + 1/Latency

  Throughput is the average throughput of sequential reads/writes (units in MB per second)

  Latency is for random reads only in seeks (units in seeks per second)

  Note: The IO time of a faster storage location is less than a slower storage location.

Example

The following example measures the performance of a storage location on v_vmartdb_node0004:

```
=> SELECT MEASURE_LOCATION_PERFORMANCE('/secondVerticaStorageLocation/' ,
'v_vmartdb_node0004');

WARNING:  measure_location_performance can take a long time. Please check logs for
progress
           measure_location_performance
------------------------------------------------
 Throughput : 122 MB/sec. Latency : 140 seeks/sec
```

See Also

ADD_LOCATION (page 426)

ALTER_LOCATION_USE (page 429)

RESTORE_LOCATION (page 525)

RETIRE_LOCATION (page 526)

Measuring Location Performance in the Administrator's Guide

MERGE_PARTITIONS

Merges ROS containers that have data belonging to partitions in a specified partition key range: `partitionKeyFrom` to `partitionKeyTo`.

Note: This function is deprecated in HP Vertica 7.0.

Syntax

```
MERGE_PARTITIONS ( table_name , partition_key_from , partition_key_to )
```

Parameters

table_name	Specifies the name of the table
partition_key_from	Specifies the start point of the partition
partition_key_to	Specifies the end point of the partition

Privileges

- Table owner
- USAGE privilege on schema that contains the table

Notes

- You cannot run MERGE_PARTITIONS() on a table with data that is not reorganized. You must reorganize the data first using `ALTER_TABLE table REORGANIZE`, or `PARTITION_TABLE(table)`.
- The edge values are included in the range, and `partition_key_from` must be less than or equal to `partition_key_to`.
- Inclusion of partitions in the range is based on the application of less than(<)/greater than(>) operators of the corresponding data type.

  Note: No restrictions are placed on a partition key's data type.
- If `partition_key_from` is the same as `partition_key_to`, all ROS containers of the partition key are merged into one ROS.

Examples

```
=> SELECT MERGE_PARTITIONS('T1', '200', '400');
=> SELECT MERGE_PARTITIONS('T1', '800', '800');
=> SELECT MERGE_PARTITIONS('T1', 'CA', 'MA');
=> SELECT MERGE_PARTITIONS('T1', 'false', 'true');
=> SELECT MERGE_PARTITIONS('T1', '06/06/2008', '06/07/2008');
=> SELECT MERGE_PARTITIONS('T1', '02:01:10', '04:20:40');
=> SELECT MERGE_PARTITIONS('T1', '06/06/2008 02:01:10', '06/07/2008 02:01:10');
=> SELECT MERGE_PARTITIONS('T1', '8 hours', '1 day 4 hours 20 seconds');
```

MOVE_PARTITIONS_TO_TABLE

Moves partitions from a source table to a target table. The target table must have the same projection column definitions, segmentation, and partition expressions as the source table. If the target table does not exist, the function creates a new table based on the source definition. The function requires both minimum and maximum range values, indicating what partition values to move.

Syntax

```
MOVE_PARTITIONS_TO_TABLE ( '[[db-name.]schema.]source_table',
'min_range_value', 'max_range_value', '[[db-name.]schema.]target_table' )
```

Parameters

[[db-name.]schema.]source_table	The source table (optionally qualified), from which you want to move partitions.
min_range_value	The minimum value in the partition to move.
max_range_value	The maximum value of the partition being moved.
target_table	The table to which the partitions are being moved.

Privileges

- Table owner
- If target table is created as part of moving partitions, the new table has the same owner as the target. If the target table exists, user must have own the target table, and have ability to call this function.

Example

If you call `move_partitions_to_table` and the destination table does not exist, the function will create the table automatically:

```
VMART=> select move_partitions_to_table ('prod_trades', '200801', '200801',
'partn_backup.trades_200801');
                         move_partitions_to_table
-------------------------------------------------------------------------------
 1 distinct partition values moved at epoch 15. Effective move epoch: 14.

(1 row)
```

See Also

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

Moving Partitions and Creating a Table Like Another in the Administrator's Guide

PARTITION_PROJECTION

Forces a split of ROS containers of the specified projection.

Syntax

```
PARTITION_PROJECTION ( '[[db-name.]schema.]projection_name' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection_name	Specifies the name of the projection.

Privileges

- Table owner
- USAGE privilege on schema

Notes

Partitioning expressions take immutable functions only, in order that the same information be available across all nodes.

PARTITION_PROJECTION() is similar to ***PARTITION_TABLE*** (page 516)(), except that PARTITION_PROJECTION works only on the specified projection, instead of the table.

Users must have USAGE privilege on schema that contains the table.PARTITION_PROJECTION() purges data while partitioning ROS containers if deletes were applied before the AHM epoch.

Example

The following command forces a split of ROS containers on the `states_p_node01` projection:

```
=> SELECT PARTITION_PROJECTION ('states_p_node01');
  partition_projection
------------------------
 Projection partitioned
```

```
(1 row)
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_TABLE (page 516)

Partitioning Tables in the Administrator's Guide

PARTITION_TABLE

Forces the system to break up any ROS containers that contain multiple distinct values of the partitioning expression. Only ROS containers with more than one distinct value participate in the split.

Syntax

```
PARTITION_TABLE ( '[[db-name.]schema.]table_name' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	Specifies the name of the table.

Privileges

- Table owner
- USAGE privilege on schema

Notes

PARTITION_TABLE is similar to ***PARTITION_PROJECTION*** (page 515), except that PARTITION_TABLE works on the specified table.

Users must have USAGE privilege on schema that contains the table.Partitioning functions take immutable functions only, in order that the same information be available across all nodes.

Example

The following example creates a simple table called `states` and partitions data by state.

```
=> CREATE TABLE states (
          year INTEGER NOT NULL,
          state VARCHAR NOT NULL)
   PARTITION BY state;
=> CREATE PROJECTION states_p (state, year) AS
   SELECT * FROM states
   ORDER BY state, year UNSEGMENTED ALL NODES;
```

Now issue the command to partition table `states`:

```
=> SELECT PARTITION_TABLE('states');
                  PARTITION_TABLE
-----------------------------------------------------
partition operation for projection 'states_p_node0004'
partition operation for projection 'states_p_node0003'
partition operation for projection 'states_p_node0002'
partition operation for projection 'states_p_node0001'
(1 row)
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

Partitioning Tables in the Administrator's Guide

PURGE

Purges all projections in the physical schema. Permanently removes deleted data from physical storage so that the disk space can be reused. You can purge historical data up to and including the epoch in which the Ancient History Mark is contained.

Syntax

```
PURGE()
```

Privileges

- Table owner
- USAGE privilege on schema

Note

- PURGE() was formerly named PURGE_ALL_PROJECTIONS. HP Vertica supports both function calls.

Caution: PURGE could temporarily take up significant disk space while the data is being purged.

See Also

MERGE_PARTITIONS (page 513)

PARTITION_TABLE (page 516)

PURGE_PROJECTION (page 520)

PURGE_TABLE (page 520)

STORAGE_CONTAINERS (page 1098)

Purging Deleted Data in the Administrator's Guide

PURGE_PARTITION

Purges a table partition of deleted rows. Similar to PURGE and PURGE_PROJECTION, this function removes deleted data from physical storage so that the disk space can be reused. It only removes data from the AHM epoch and earlier.

Syntax

```
PURGE_PARTITION ( '[[db_name.]schema_name.]table_name', partition_key )
```

Parameters

`[[db_name.]schema_name.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	The name of the partitioned table
partition_key	The key of the partition to be purged of deleted rows

Privileges

- Table owner
- USAGE privilege on schema

Example

The following example lists the count of deleted rows for each partition in a table, then calls PURGE_PARTITION to purge the deleted rows from the data.

```
=> SELECT partition_key,table_schema,projection_name,sum(deleted_row_count) AS deleted_row_count
FROM partitions
-> GROUP BY partition_key,table_schema,projection_name  ORDER BY partition_key;

 partition_key | table_schema | projection_name | deleted_row_count
---------------+--------------+-----------------+-------------------
 0             | public       | t_super         |                 2
 1             | public       | t_super         |                 2
 2             | public       | t_super         |                 2
 3             | public       | t_super         |                 2
 4             | public       | t_super         |                 2
 5             | public       | t_super         |                 2
 6             | public       | t_super         |                 2
 7             | public       | t_super         |                 2
 8             | public       | t_super         |                 2
 9             | public       | t_super         |                 1
(10 rows)


=> SELECT PURGE_PARTITION('t',5); -- Purge partition with key 5.
                          purge_partition
-------------------------------------------------------------------------
 Task: merge partitions
(Table: public.t) (Projection: public.t_super)

(1 row)


=> SELECT partition_key,table_schema,projection_name,sum(deleted_row_count) AS deleted_row_count
FROM partitions
-> GROUP BY partition_key,table_schema,projection_name ORDER BY partition_key;

 partition_key | table_schema | projection_name | deleted_row_count
---------------+--------------+-----------------+-------------------
 0             | public       | t_super         |                 2
 1             | public       | t_super         |                 2
 2             | public       | t_super         |                 2
 3             | public       | t_super         |                 2
 4             | public       | t_super         |                 2
 5             | public       | t_super         |                 0
 6             | public       | t_super         |                 2
 7             | public       | t_super         |                 2
 8             | public       | t_super         |                 2
 9             | public       | t_super         |                 1
(10 rows)
```

See Also

PURGE (page 517)

PURGE_PROJECTION (on page 520)

PURGE_TABLE (page 520)

MERGE_PARTITIONS (page 513)

STORAGE_CONTAINERS (page 1098)

PURGE_PROJECTION

Purges the specified projection. Permanently removes deleted data from physical storage so that the disk space can be reused. You can purge historical data up to and including the epoch in which the Ancient History Mark is contained.

Caution: PURGE_PROJECTION could temporarily take up significant disk space while purging the data.

Syntax

```
PURGE_PROJECTION ( '[[db-name.]schema.]projection_name' )
```

Parameters

`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection_name	Identifies the projection name. When using more than one schema, specify the schema that contains the projection, as noted above.

Privileges

- Table owner
- USAGE privilege on schema

Notes

See ***PURGE*** (page 517) for notes about the outcome of purge operations.

See Also

MERGE_PARTITIONS (page 513)

PURGE_TABLE (page 520)

STORAGE_CONTAINERS (page 1098)

Purging Deleted Data in the Administrator's Guide

PURGE_TABLE

Note: This function was formerly named PURGE_TABLE_PROJECTIONS(). HP Vertica still supports the former function name.

Purges all projections of the specified table. You cannot use this function to purge temporary tables. Permanently removes deleted data from physical storage so that the disk space can be reused. You can purge historical data up to and including the epoch in which the Ancient History Mark is contained.

Syntax

```
PURGE_TABLE ( '[[db-name.]schema.]table_name' )
```

Parameters

`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	Specifies the table to purge.

Privileges

- Table owner
- USAGE privilege on schema

Caution: PURGE_TABLE could temporarily take up significant disk space while the data is being purged.

Example

The following example purges all projections for the store sales fact table located in the Vmart schema:

```
=> SELECT PURGE_TABLE('store.store_sales_fact');
```

See Also

PURGE (page 517) for notes about the outcome of purge operations.

MERGE_PARTITIONS (page 513)

PURGE_TABLE (page 520)

STORAGE_CONTAINERS (page 1098)

Purging Deleted Data in the Administrator's Guide

REBALANCE_CLUSTER

Starts rebalancing data in the cluster synchronously. Rebalancing redistributes your database projections' data across all nodes, refreshes projections, sets the Ancient History Mark, and drops projections that are no longer needed. Rebalancing is useful after you:

- mark one or more nodes as ephemeral in preparation of removing them from the cluster, so that HP Vertica migrates the data on the ephemeral nodes away to other nodes.
- add one or more nodes to the cluster, so that HP Vertica can populate the empty nodes with data.
- change the scaling factor, which determines the number of storage containers used to store a projection across the database.

Since function runs the rebalance task synchronously, it does not return until the data has been rebalanced. Closing or dropping the session cancels the rebalance task.

Syntax

```
REBALANCE_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT REBALANCE_CLUSTER();
 REBALANCE_CLUSTER
-------------------
 REBALANCED
(1 row)
```

See Also

- Rebalancing Data Across Nodes
- ***START_REBALANCE_CLUSTER*** (page 537)
- ***CANCEL_REBALANCE_CLUSTER*** (page 450)

REENABLE_DUPLICATE_KEY_ERROR

Restores the default behavior of error reporting by reversing the effects of DISABLE_DUPLICATE_KEY_ERROR. Effects are session scoped.

Syntax

```
REENABLE_DUPLICATE_KEY_ERROR();
```

Privileges

Must be a superuser

Examples

For examples and usage see ***DISABLE_DUPLICATE_KEY_ERROR*** (page 466).

See Also

ANALYZE_CONSTRAINTS (page 432)

REFRESH

Performs a synchronous, optionally-targeted refresh of a specified table's projections.

Information about a refresh operation—whether successful or unsuccessful—is maintained in the ***PROJECTION_REFRESHES*** (page 1056) system table until either the ***CLEAR_PROJECTION_REFRESHES*** (page 455)() function is executed or the storage quota for the table is exceeded. The `PROJECTION_REFRESHES.IS_EXECUTING` column returns a boolean value that indicates whether the refresh is currently running (t) or occurred in the past (f).

Syntax

```
REFRESH ( '[[db-name.]schema.]table_name [ , ... ]' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	Specifies the name of a specific table containing the projections to be refreshed. The REFRESH() function attempts to refresh all the tables provided as arguments in parallel. Such calls will be part of the Database Designer deployment (and deployment script). When using more than one schema, specify the schema that contains the table, as noted above.

Returns

Column Name	Description
`Projection Name`	The name of the projection that is targeted for refresh.
`Anchor Table`	The name of the projection's associated anchor table.
`Status`	The status of the projection: ▪ Queued — Indicates that a projection is queued for refresh. ▪ Refreshing — Indicates that a refresh for a projection is in process. ▪ Refreshed — Indicates that a refresh for a projection has successfully completed.

	▪ Failed — Indicates that a refresh for a projection did not successfully complete.
Refresh Method	The method used to refresh the projection: ▪ Buddy – Uses the contents of a buddy to refresh the projection. This method maintains historical data. This enables the projection to be used for historical queries. ▪ Scratch – Refreshes the projection without using a buddy. This method does not generate historical data. This means that the projection cannot participate in historical queries from any point before the projection was refreshed.
Error Count	The number of times a refresh failed for the projection.
Duration (sec)	The length of time that the projection refresh ran in seconds.

Privileges

REFRESH() works only if invoked on tables owned by the calling user.

Notes

- Unlike START_REFRESH(), which runs in the background, REFRESH() runs in the foreground of the caller's session.
- The REFRESH() function refreshes only the projections in the specified table.
- If you run REFRESH() without arguments, it refreshes all non up-to-date projections. If the function returns a header string with no results, then no projections needed refreshing.

Example

The following command refreshes the projections in tables `t1` and `t2`:

```
=> SELECT REFRESH('t1, t2');
refresh
-------------------------------------------------------------------------------
 Refresh completed with the following outcomes:
Projection Name: [Anchor Table] [Status] [Refresh Method] [Error Count] [Duration (sec)]
-------------------------------------------------------------------------------
"public"."t1_p": [t1] [refreshed] [scratch] [0] [0]
"public"."t2_p": [t2] [refreshed] [scratch] [0] [0]
```

This next command shows that only the projection on table `t` was refreshed:

```
=> SELECT REFRESH('allow, public.deny, t');"
refresh
-------------------------------------------------------------------------------
 Refresh completed with the following outcomes:
Projection Name: [Anchor Table] [Status] [Refresh Method] [Error Count] [Duration (sec)]
-------------------------------------------------------------------------------
"n/a"."n/a": [n/a] [failed: insufficient permissions on table "allow"] [] [1] [0]
"n/a"."n/a": [n/a] [failed: insufficient permissions on table "public.deny"] [] [1] [0]
"public"."t_p1": [t] [refreshed] [scratch] [0] [0]
```

See Also

CLEAR_PROJECTION_REFRESHES (page 455)

PROJECTION_REFRESHES (page 1056)

START_REFRESH (page 538)

Clearing PROJECTION_REFRESHES History in the Administrator's Guide

RESTORE_LOCATION

Restores a storage location that was previously retired with ***RETIRE_LOCATION*** (page 526).

Syntax

```
RESTORE_LOCATION ( 'path'  , 'node' )
```

Parameters

path	Specifies where the retired storage location is mounted.
node	Is the HP Vertica node where the retired location is available.

Privileges

Must be a superuser

Effects of Restoring a Previously Retired Location

After restoring a storage location, HP Vertica re-ranks all of the cluster storage locations and uses the newly-restored location to process queries as determined by its rank.

Monitoring Storage Locations

Disk storage information that the database uses on each node is available in the ***V_MONITOR.DISK_STORAGE*** (page 1014) system table.

Example

The following example restores the retired storage location on node3:

```
=> SELECT RESTORE_LOCATION ('/thirdHP VerticaStorageLocation/' ,
'v_vmartdb_node0004');
```

See Also

Modifying Storage Locations in the Administrator's Guide

ADD_LOCATION (page 426)

ALTER_LOCATION_USE (page 429)

DROP_LOCATION (page 472)

RETIRE_LOCATION (page 526)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

RETIRE_LOCATION

Makes the specified storage location inactive.

Syntax

```
RETIRE_LOCATION ( 'path'  , 'node' )
```

Parameters

path	Specifies where the storage location to retire is mounted.
node	Is the HP Vertica node where the location is available.

Privileges

Must be a superuser

Effects of Retiring a Storage Location

When you use this function, HP Vertica checks that the location is not the only storage for data and temp files. At least one location must exist on each node to store data and temp files, though you can store both sorts of files in either the same location, or separate locations.

NOTE: You cannot retire a location if it is used in a storage policy, *and* is the last available storage for its associated objects.

When you retire a storage location:

- No new data is stored at the retired location, unless you first restore it with the ***RESTORE_LOCATION()*** (page 525) function.
- If the storage location being retired contains stored data, the data is not moved, so you cannot drop the storage location. Instead, HP Vertica removes the stored data through one or more mergeouts.
- If the storage location being retired was used only for temp files, you can drop the location. See Dropping Storage Locations in the Administrators Guide and the ***DROP_LOCATION()*** (page 472) function.

Monitoring Storage Locations

Disk storage information that the database uses on each node is available in the ***V_MONITOR.DISK_STORAGE*** (page 1014) system table.

Example

The following example retires a storage location:

```
=> SELECT RETIRE_LOCATION ('/secondVerticaStorageLocation/' ,
'v_vmartdb_node0004');
```

See Also

Retiring Storage Locations in the Administrator's Guide

ADD_LOCATION (page 426)

ALTER_LOCATION_USE (page 429)

DROP_LOCATION (page 472)

RESTORE_LOCATION (page 525)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

SET_AHM_EPOCH

Sets the Ancient History Mark (AHM) to the specified epoch. This function allows deleted data up to and including the AHM epoch to be purged from physical storage.

SET_AHM_EPOCH is normally used for testing purposes. Consider ***SET_AHM_TIME*** (page 528) instead, which is easier to use.

Syntax

```
SET_AHM_EPOCH ( epoch, [ true ] )
```

Parameters

epoch	Specifies one of the following: ▪ The number of the epoch in which to set the AHM ▪ Zero (0) (the default) disables ***purge*** (page 517)
true	Optionally allows the AHM to advance when nodes are down. **Note:** If the AHM is advanced after the last good epoch of the failed nodes, those nodes must recover all data from scratch. Use with care.

Privileges

Must be a superuser

Notes

If you use SET_AHM_EPOCH , the number of the specified epoch must be:

- Greater than the current AHM epoch
- Less than the current epoch
- Less than or equal to the cluster last good epoch (the minimum of the last good epochs of the individual nodes in the cluster)
- Less than or equal to the cluster refresh epoch (the minimum of the refresh epochs of the individual nodes in the cluster)

Use the ***SYSTEM*** (page 1111) table to see current values of various epochs related to the AHM; for example:

```
=> SELECT * from SYSTEM;
```

```
-[ RECORD 1 ]------------+---------------------------
current_timestamp        | 2009-08-11 17:09:54.651413
current_epoch            | 1512
ahm_epoch                | 961
last_good_epoch          | 1510
refresh_epoch            | -1
designed_fault_tolerance | 1
node_count               | 4
node_down_count          | 0
current_fault_tolerance  | 1
catalog_revision_number  | 1590
wos_used_bytes           | 0
wos_row_count            | 0
ros_used_bytes           | 41490783
ros_row_count            | 1298104
total_used_bytes         | 41490783
total_row_count          | 1298104
```

All nodes must be up. You cannot use SET_AHM_EPOCH when any node in the cluster is down, except by using the optional *true* parameter.

When a node is down and you issue `SELECT MAKE_AHM_NOW()`, the following error is printed to the vertica.log:

```
Some nodes were excluded from setAHM. If their LGE is before the AHM they will perform full recovery.
```

Examples

The following command sets the AHM to a specified epoch of 12:

```
=> SELECT SET_AHM_EPOCH(12);
```

The following command sets the AHM to a specified epoch of 2 and allows the AHM to advance despite a failed node:

```
=> SELECT SET_AHM_EPOCH(2, true);
```

See Also

MAKE_AHM_NOW (page 508)

SET_AHM_TIME (page 528)

SYSTEM (page 1111)

SET_AHM_TIME

Sets the Ancient History Mark (AHM) to the epoch corresponding to the specified time on the initiator node. This function allows historical data up to and including the AHM epoch to be purged from physical storage.

Syntax

```
SET_AHM_TIME ( time , [ true ] )
```

Parameters

time	Is a ***TIMESTAMP*** (page 97) value that is automatically converted to the appropriate epoch number.
true	[Optional] Allows the AHM to advance when nodes are down. **Note:** If the AHM is advanced after the last good epoch of the failed nodes, those nodes must recover all data from scratch.

Privileges

Must be a superuser

Notes

- SET_AHM_TIME returns a TIMESTAMP WITH TIME ZONE value representing the end point of the AHM epoch.
- You cannot change the AHM when any node in the cluster is down, except by using the optional *true* parameter.
- When a node is down and you issue `SELECT MAKE_AHM_NOW()`, the following error is printed to the vertica.log:

  ```
  Some nodes were excluded from setAHM. If their LGE is before the AHM they
  will perform full recovery.
  ```

Examples

Epochs depend on a configured epoch advancement interval. If an epoch includes a three-minute range of time, the purge operation is accurate only to within minus three minutes of the specified timestamp:

```
=> SELECT SET_AHM_TIME('2008-02-27 18:13');
           set_ahm_time
-----------------------------------
 AHM set to '2008-02-27 18:11:50-05'
(1 row)
```

Note: The -05 part of the output string is a time zone value, an offset in hours from UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, or GMT).

In the above example, the actual AHM epoch ends at 18:11:50, roughly one minute before the specified timestamp. This is because SET_AHM_TIME selects the epoch that ends at or before the specified timestamp. It does not select the epoch that ends after the specified timestamp because that would purge data deleted as much as three minutes after the AHM.

For example, using only hours and minutes, suppose that epoch 9000 runs from 08:50 to 11:50 and epoch 9001 runs from 11:50 to 15:50. `SET_AHM_TIME('11:51')` chooses epoch 9000 because it ends roughly one minute before the specified timestamp.

In the next example, if given an environment variable set as `date =`date`;` the following command fails if a node is down:

```
=> SELECT SET_AHM_TIME('$date');
```

In order to force the AHM to advance, issue the following command instead:

```
=> SELECT SET_AHM_TIME('$date', true);
```

See Also

MAKE_AHM_NOW (page 508)

SET_AHM_EPOCH (page 527) for a description of the range of valid epoch numbers.

SET DATESTYLE (page 903) for information about specifying a ***TIMESTAMP*** (page 97) value.

SET_AUDIT_TIME

Sets the time that HP Vertica performs automatic database size audit to determine if the size of the database is compliant with the raw data allowance in your HP Vertica license. Use this function if the audits are currently scheduled to occur during your database's peak activity time. This is normally not a concern, since the automatic audit has little impact on database performance.

Note: Audits are scheduled by the preceding audit, so changing the audit time does not affect the next scheduled audit. For example, if your next audit is scheduled to take place at 11:59PM and you use SET_AUDIT_TIME to change the audit schedule 3AM, the previously scheduled 11:59PM audit still runs. As that audit finishes, it schedules the next audit to occur at 3AM.

If you want to prevent the next scheduled audit from running at its scheduled time, you can change the scheduled time using SET_AUDIT_TIME then manually trigger an audit to run immediately using ***AUDIT_LICENSE_SIZE*** (page 450). As the manually-triggered audit finishes, it schedules the next audit to occur at the time you set using SET_AUDIT_TIME (effectively overriding the previously scheduled audit).

Syntax

```
SET_AUDIT_TIME(time)
```

time	A string containing the time in `'HH:MM AM/PM'` format (for example, `'1:00 AM'`) when the audit should run daily.

Privileges

Must be a superuser

Example

```
=> SELECT SET_AUDIT_TIME('3:00 AM');
                              SET_AUDIT_TIME
-----------------------------------------------------------------------
 The scheduled audit time will be set to 3:00 AM after the next audit.
(1 row)
```

SET_DATA_COLLECTOR_POLICY

Sets the retention policy for the specified component on all nodes. Failed nodes receive the setting when they rejoin the cluster.

Syntax

```
SET_DATA_COLLECTOR_POLICY('component', 'memoryKB', 'diskKB' )
```

Parameters

component	Returns the retention policy for the specified component.
memoryKB	Specifies the memory size to retain in kilobytes.
diskKB	Specifies the disk size in kilobytes.

Privileges

Must be a superuser

Notes

- Only a superuser can configure the Data Collector.
- Before you change a retention policy, view its current setting by calling the GET_DATA_COLLECTOR_POLICY() function.
- If you don't know the name of a component, query the V_MONITOR.DATA_COLLECTOR system table for a list; for example:

  ```
  => SELECT DISTINCT component, description FROM data_collector ORDER BY
  1 ASC;
  ```

Example

The following command returns the retention policy for the ResourceAcquisitions component:

```
=> SELECT get_data_collector_policy('ResourceAcquisitions');
          get_data_collector_policy
-----------------------------------------------
 1000KB kept in memory, 10000KB kept on disk.
(1 row)
```

This command changes the memory and disk setting for ResourceAcquisitions from their current setting of 1,000 KB and 10,000 KB to 1,500 KB and 25,000 KB, respectively:

```
=> SELECT set_data_collector_policy('ResourceAcquisitions', '1500', '25000');
 set_data_collector_policy
---------------------------
 SET
(1 row)
```

To verify the setting, call the GET_DATA_COLLECTOR_POLICY() function on the specified component:

```
=> SELECT get_data_collector_policy('ResourceAcquisitions');
          get_data_collector_policy
-----------------------------------------------
 1500KB kept in memory, 25000KB kept on disk.
(1 row)
```

See Also

GET_DATA_COLLECTOR_POLICY() (page 496)

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining Monitoring Information in the Administrator's Guide

SET_LOCATION_PERFORMANCE

Sets disk performance for the location specified.

Syntax

```
SET_LOCATION_PERFORMANCE ( 'path' , 'node' , 'throughput' , 'average_latency' )
```

Parameters

path	Specifies where the storage location to set is mounted.
node	Is the HP Vertica node where the location to be set is available. If this parameter is omitted, *node* defaults to the initiator.
throughput	Specifies the throughput for the location, which must be 1 or more.
average_latency	Specifies the average latency for the location. The *average_latency* must be 1 or more.

Privileges

Must be a superuser

Notes

To obtain the throughput and average latency for the location, run the ***MEASURE_LOCATION_PERFORMANCE()*** (page 511) function before you attempt to set the location's performance.

Example

The following example sets the performance of a storage location on node2 to a throughput of 122 megabytes per second and a latency of 140 seeks per second.

```
=> SELECT SET_LOCATION_PERFORMANCE('/secondVerticaStorageLocation/','node2','122','140');
```

See Also

ADD_LOCATION (page 426)

MEASURE_LOCATION_PERFORMANCE (page 511)

Measuring Location Performance and Setting Location Performance in the Administrator's Guide

SET_LOGLEVEL

Use SET_LOGLEVEL to set the logging level in the HP Vertica database log files.

Syntax

```
SELECT SET_LOGLEVEL(n)
```

Parameters

n	Logging Level	Description
0	DISABLE	No logging
1	CRITICAL	Errors requiring database recovery
2	WARNING	Errors indicating problems of lesser magnitude
3	INFO	Informational messages
4	DEBUG	Debugging messages
5	TRACE	Verbose debugging messages
6	TIMING	Verbose debugging messages

Privileges

Must be a superuser

SET_SCALING_FACTOR

Sets the scaling factor used to determine the size of the storage containers used when rebalancing the database and when using local data segmentation is enabled. See Cluster Scaling for details.

Syntax

```
SET_SCALING_FACTOR(factor)
```

Parameters

factor	An integer value between 1 and 32. HP Vertica uses this value to calculate the number of storage containers each projection is broken into when rebalancing or when local data segmentation is enabled.

Note: Setting the scaling factor value too high can cause nodes to create too many small container files, greatly reducing efficiency and potentially causing a Too Many ROS containers error (also known as "ROS pushback"). The scaling factor should be set high enough so that rebalance can transfer local segments to satisfy the skew threshold, but small enough that the number of storage containers does not exceed ROS pushback. The number of storage containers should be greater than or equal to the number of partitions multiplied by the number local of segments (# storage containers >= # partitions * # local segments).

Privileges

Must be a superuser

Example

```
=> SELECT SET_SCALING_FACTOR(12);
 SET_SCALING_FACTOR
--------------------
 SET
(1 row)
```

SET_OBJECT_STORAGE_POLICY

Creates or changes an object storage policy by associating a database object with a labeled storage location.

NOTE: You cannot create a storage policy on a USER type storage location.

Syntax

```
SET_OBJECT_STORAGE_POLICY ( 'object_name', 'location_label' [, 'key_min,
key_max'] [, 'enforce_storage_move' ] )
```

Parameters

object_name	Identifies the database object assigned to a labeled storage location. The *object_name* can resolve to a database, schema, or table.
location_label	The label of the storage location with which *object_name* is being associated.
key_min, key_max	Applicable only when *object_name* is a table, key_min and key_max specify the table partition key value range to be stored at the location.
enforce_storage_move={true | false}	[Optional] Applicable only when setting a storage policy for an object that has data stored at another labeled location. Specify this parameter as `true` to move all existing storage data to the target location within this function's transaction.

Privileges

Must be the object owner to set the storage policy, and have access to the storage location.

New Storage Policy

If an object does not have a storage policy, this function creates a new policy. The labeled location is then used as the default storage location during TM operations, such as moveout and mergeout.

Existing Storage Policy

If the object already has an active storage policy, calling this function changes the default storage for the object to the new labeled location. Any existing data stored on the previous storage location is marked to move to the new location during the next TM moveout operations, unless you use the *enforce_storage_move* option.

Forcing Existing Data Storage to a New Storage Location

You can optionally use this function to move existing data storage to a new location as part of completing the current transaction, by specifying the last parameter as `true`.

To move existing data as part of the next TM moveout, either omit the parameter, or specify its value as f`alse`.

NOTE: Specifying the parameter as `true` performs a cluster-wide operation. If an error occurs on any node, the function displays a warning message, skips the offending node, and continues execution on the remaining nodes.

Example

This example sets a storage policy for the table `states` to use the storage labeled `SSD` as its default location:

```
VMART=> select set_object_storage_policy ('states', 'SSD');
     set_object_storage_policy
---------------------------------
 Default storage policy set.
(1 row)
```

See Also

ALTER_LOCATION_LABEL (page 430)

CLEAR_OBJECT_STORAGE_POLICY (page 457)

Creating Storage Policies in the Administrator's Guide

Moving Data Storage Locations in the Administrator's Guide

SHUTDOWN

Forces a database to shut down, even if there are users connected.

Syntax

```
SHUTDOWN ( [ 'false' | 'true' ] )
```

Parameters

false	[Default] Returns a message if users are connected. Has the same effect as supplying no parameters.
true	Performs a moveout operation and forces the database to shut down, disallowing further connections.

Privileges

Must be a superuser

Notes

- Quotes around the `true` or `false` arguments are optional.
- Issuing the shutdown command without arguments or with the default (false) argument returns a message if users are connected, and the shutdown fails. If no users are connected, the database performs a moveout operation and shuts down.
- Issuing the `SHUTDOWN('true')` command forces the database to shut down whether users are connected or not.
- You can check the status of the shutdown operation in the `vertica.log file`: `2010-03-09 16:51:52.625 unknown:0x7fc6d6d2e700 [Init] <INFO> Shutdown complete. Exiting.`
- As an alternative to SHUTDOWN(), you can also temporarily set MaxClientSessions to 0 and then use CLOSE_ALL_SESSIONS(). New client connections cannot connect unless they connect using the dbadmin account. See ***CLOSE_ALL_SESSIONS*** (page 461) for details.

Examples

The following command attempts to shut down the database. Because users are connected, the command fails:

```
=> SELECT SHUTDOWN('false');
NOTICE:  Cannot shut down while users are connected
          SHUTDOWN
----------------------------
 Shutdown: aborting shutdown
(1 row)
```

Note that `SHUTDOWN()` and `SHUTDOWN('false')` perform the same operation:

```
=> SELECT SHUTDOWN();
NOTICE:  Cannot shut down while users are connected
          SHUTDOWN
----------------------------
 Shutdown: aborting shutdown
(1 row)
```

Using the `'true'` parameter forces the database to shut down, even though clients might be connected:

```
=> SELECT SHUTDOWN('true');
```

```
          SHUTDOWN
---------------------------
 Shutdown: moveout complete
(1 row)
```

See Also

SESSIONS (page 1095)

SLEEP

Waits a specified number of seconds before executing another statement or command.

Syntax

```
SLEEP( seconds )
```

Parameters

seconds	The wait time, specified in one or more seconds (0 or higher) expressed as a positive integer. Single quotes are optional; for example, `SLEEP(3)` is the same as `SLEEP('3')`.

Notes

- This function returns value 0 when successful; otherwise it returns an error message due to syntax errors.
- You cannot cancel a sleep operation.
- Be cautious when using SLEEP() in an environment with shared resources, such as in combination with transactions that take exclusive locks.

Example

The following command suspends execution for 100 seconds:

```
=> SELECT SLEEP(100);
 sleep
-------
     0
(1 row)
```

START_REBALANCE_CLUSTER

Asynchronously starts a data rebalance task. Rebalancing redistributes your database projections' data across all nodes, refreshes projections, sets the Ancient History Mark, and drops projections that are no longer needed. Rebalancing is useful after you:

- mark one or more nodes as ephemeral in preparation of removing them from the cluster, so that HP Vertica migrates the data on the ephemeral nodes away to other nodes.

- add one or more nodes to the cluster, so that HP Vertica can populate the empty nodes with data.
- change the scaling factor, which determines the number of storage containers used to store a projection across the database.

Since this function starts the rebalance task in the background, it returns immediately after the task has started. Since it is a background task, rebalancing will continue even if the session that started it is closed. It even continues after a cluster recovery if the database shuts down while it is in progress. The only way to stop the task is by the CANCEL_REBALANCE_CLUSTER function.

Syntax

```
START_REBALANCE_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT START_REBALANCE_CLUSTER();
 START_REBALANCE_CLUSTER
-------------------------
 REBALANCING
(1 row)
```

See Also

- Rebalancing Data Across Nodes
- ***CANCEL_REBALANCE_CLUSTER*** (page 450)
- ***REBALANCE_CLUSTER*** (page 522)

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

- When a design is deployed through the Database Designer, it is automatically refreshed. See Deploying Designs in the Administrator's Guide.
- All nodes must be up in order to start a refresh.
- START_REFRESH() has no effect if a refresh is already running.
- A refresh is run asynchronously.
- Shutting down the database ends the refresh.
- To view the progress of the refresh, see the ***PROJECTION_REFRESHES*** (page 1056) and ***PROJECTIONS*** (page 961) system tables.

- If a projection is updated from scratch, the data stored in the projection represents the table columns as of the epoch in which the refresh commits. As a result, the query optimizer might not choose the new projection for AT EPOCH queries that request historical data at epochs older than the refresh epoch of the projection. Projections refreshed from buddies retain history and can be used to answer historical queries.

Privileges

None

Example

The following command starts the refresh operation:

```
=> SELECT START_REFRESH();
             start_refresh
---------------------------------------
 Starting refresh background process.
```

See Also

CLEAR_PROJECTION_REFRESHES (page 455)

MARK_DESIGN_KSAFE (page 510)

PROJECTION_REFRESHES (page 1056)

PROJECTIONS (page 961)

Clearing PROJECTION_REFRESHES History in the Administrator's Guide

Catalog Management Functions

This section contains catalog management functions specific to HP Vertica.

DUMP_CATALOG

Returns an internal representation of the HP Vertica catalog. This function is used for diagnostic purposes.

Syntax

```
DUMP_CATALOG()
```

Privileges

None; however, function dumps only the objects visible to the user.

Notes

To obtain an internal representation of the HP Vertica catalog for diagnosis, run the query:

```
=> SELECT DUMP_CATALOG();
```

The output is written to the specified file:

```
\o /tmp/catalog.txt
SELECT DUMP_CATALOG();
\o
```

EXPORT_CATALOG

Generates a SQL script that you can use to recreate a physical schema design in its current state on a different cluster. This function always attempts to recreate projection statements with KSAFE clauses, if they exist in the original definitions, or OFFSET clauses if they do not.

Syntax

```
EXPORT_CATALOG ( [ 'destination' ] , [ 'scope' ] )
```

Parameters

destination	Specifies the path and name of the SQL output file. An empty string (''), which is the default, outputs the script to standard output. The function writes the script to the catalog directory if no destination is specified. If you specify a file that does not exist, the function creates one. If the file pre-exists, the function silently overwrites its contents.
scope	Determines what to export: ▪ DESIGN — Exports schemas, tables, constraints, views, and projections to which the user has access. This is the default value. ▪ DESIGN_ALL — Exports all the design objects plus system objects created in Database Designer (for example, design contexts and their tables). The objects that are exported are those to which the user has access. ▪ TABLES — Exports all tables, constraints, and projections for for which the user has permissions. See also ***EXPORT_TABLES*** (page 491).

Privileges

None; however:

- Function exports only the objects visible to the user
- Only a superuser can export output to file

Example

The following example exports the design to standard output:

```
=> SELECT EXPORT_CATALOG('','DESIGN');
```

See Also

EXPORT_OBJECTS

EXPORT_TABLES (page 491)

Exporting the Catalog in the Administrator's Guide

EXPORT_OBJECTS

Generates a SQL script you can use to recreate catalog objects on a different cluster. The generated script includes only the non-virtual objects to which the user has access. The function exports catalog objects in order dependency for correct recreation. Running the generated SQL script on another cluster then creates all referenced objects before their dependent objects.

Note: You cannot use EXPORT_OBJECTS to export a view without its dependencies.

The EXPORT_OBJECTS function always attempts to recreate projection statements with KSAFE clauses, if they existed in the original definitions, or OFFSET clauses, if they did not.

None of the EXPORT_OBJECTS parameters accepts a NULL value as input. EXPORT_OBJECTS returns an error if an explicitly-specified object does not exist, or the user does not have access to the object.

Syntax

```
EXPORT_OBJECTS( [ 'destination' ] , [ 'scope' ] , [ 'ksafe' ] )
```

Parameters

destination	Specifies the path and name of the SQL output file. The default empty string (`''`) outputs the script contents to standard output. Non-DBadmin users can specify only an empty string. If you specify a file that does not exist, the function creates one. If the file pre-exists, the function silently overwrites its contents. If you do not specify an explicit path destination, the function outputs the script for the exported objects to the catalog directory.

scope	Determines which catalog objects to export, where you specify *scope* as follows: ▪ An empty string (' ')—exports all non-virtual objects to which the user has access, including constraints. (Note that constraints are not objects that can be passed as individual arguments.) An empty string is the default *scope* value for scope if you do not limit the export. ▪ A comma-delimited list of catalog objects to export, which can include the following: ▪ —' [*dbname*.]*schema*.*object* '— matches each named schema object. You can optionally qualify the schema with a database prefix. A named schema object can be a table, projection, view, sequence, or user-defined SQL function. ▪ —' [*dbname*.]*schema* — matches the named schema, which you can optionally qualify with a database prefix. For a schema, HP Vertica exports all non-virtual objects that the user has access to within the schema. If a schema and table have the same name, the schema takes precedence.
ksafe	Specifies whether to incorporate a MARK_DESIGN_KSAFE statement with the correct K-safe value for the database. The statement is placed at the end of the output script. Use one of the following: ▪ true—adds the MARK_DESIGN_KSAFE statement to the script. This is the default value. ▪ false—omits the MARK_DESIGN_KSAFE statement from the script. Adding the MARK_DESIGN_KSAFE statement is useful if you plan to import the SQL script into a new database, and you want the new database to inherit the K-safety value of the original database.

Privileges

None; however:

- Function exports only the objects visible to the user
- Only a superuser can export output to a file

Example

The following example exports all the non-virtual objects to which the user has access to standard output. The example uses `false` for the last parameter, indicating that the file will not include the `MARK_DESIGN_KSAFE` statement at the end.

```
=> SELECT EXPORT_OBJECTS(' ',' ',false);
```

INSTALL_LICENSE

Installs the license key in the global catalog.

description

Syntax

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

Parameters

filename	specifies the absolute pathname of a valid license file.

Privileges

Must be a superuser

Notes

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

Examples

```
=> SELECT INSTALL_LICENSE('/tmp/vlicense.dat');
```

MARK_DESIGN_KSAFE

Enables or disables high availability in your environment, in case of a failure. Before enabling recovery, MARK_DESIGN_KSAFE queries the catalog to determine whether a cluster's physical schema design meets the following requirements:

- Small, unsegmented tables are replicated on all nodes.
- Large table superprojections are segmented with each segment on a different node.
- Each large table projection has at least one buddy projection for K-safety=1 (or two buddy projections for K-safety=2).

  Buddy projections are also segmented across database nodes, but the distribution is modified so that segments that contain the same data are distributed to different nodes. See High Availability Through Projections in the Concepts Guide.

  Note: 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

```
MARK_DESIGN_KSAFE ( k )
```

Parameters

k	2 enables high availability if the schema design meets requirements for K-safety=2 1 enables high availability if the schema design meets requirements for K-safety=1 0 disables high availability

If you specify a *k* value of one (1) or two (2), HP Vertica returns one of the following messages.

Success:

```
Marked design n-safe
```

Failure:

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

n in the message is 1 or 2 and represents the *k* value.

Privileges

Must be a superuser

Notes

- The database's internal recovery state persists across database restarts but it is not checked at startup time.
- If a database has automatic recovery enabled, you must temporarily disable automatic recovery before creating a new table.

- When one node fails on a system marked K-safe=1, the remaining nodes are available for DML operations.

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)
```

See Also

SYSTEM (page 1111)

High Availability and Recovery in the Concepts Guide

SQL System Tables (Monitoring APIs) (page 933) topic in the Administrator's Guide

Using Identically Segmented Projections in the Programmer's Guide

Failure Recovery in the Administrator's Guide

Cluster Scaling Functions

This section contains functions that control how the cluster organizes data for rebalancing.

CANCEL_REBALANCE_CLUSTER

Stops any rebalance task currently in progress.

Syntax

```
CANCEL_REBALANCE_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT CANCEL_REBALANCE_CLUSTER();
 CANCEL_REBALANCE_CLUSTER
--------------------------
 CANCELED
(1 row)
```

See Also

- ***START_REBALANCE_CLUSTER*** (page 537)
- ***REBALANCE_CLUSTER*** (page 522)

DISABLE_ELASTIC_CLUSTER

Disables elastic cluster scaling, which prevents HP Vertica from bundling data into chunks that are easily transportable to other nodes when performing cluster resizing. The main reason to disable elastic clustering is if you find that the slightly unequal data distribution in your cluster caused by grouping data into discrete blocks results in performance issues.

Syntax

```
DISABLE_ELASTIC_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT DISABLE_ELASTIC_CLUSTER();
 DISABLE_ELASTIC_CLUSTER
-------------------------
 DISABLED
(1 row)
```

See Also

- ***ENABLE_ELASTIC_CLUSTER*** (page 482)

DISABLE_LOCAL_SEGMENTS

Disable local data segmentation, which breaks projections segments on nodes into containers that can be easily moved to other nodes. See Local Data Segmentation in the Administrator's Guide for details.

Syntax

```
DISABLE_LOCAL_SEGMENTS()
```

Privileges

Must be a superuser

Example

```
=> SELECT DISABLE_LOCAL_SEGMENTS();
 DISABLE_LOCAL_SEGMENTS
------------------------
 DISABLED
(1 row)
```

ENABLE_ELASTIC_CLUSTER

Enables elastic cluster scaling, which makes enlarging or reducing the size of your database cluster more efficient by segmenting a node's data into chunks that can be easily moved to other hosts.

Note: Databases created using HP Vertica Version 5.0 and later have elastic cluster enabled by default. You need to use this function on databases created before version 5.0 in order for them to use the elastic clustering feature.

Syntax

```
ENABLE_ELASTIC_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT ENABLE_ELASTIC_CLUSTER();
 ENABLE_ELASTIC_CLUSTER
------------------------
 ENABLED
(1 row)
```

See Also

- ***DISABLE_ELASTIC_CLUSTER*** (page 468)

ENABLE_LOCAL_SEGMENTS

Enables local storage segmentation, which breaks projections segments on nodes into containers that can be easily moved to other nodes. See Local Data Segmentation in the Administrator's Guide for more information.

Syntax

```
ENABLE_LOCAL_SEGMENTS()
```

Privileges

Must be a superuser

Example

```
=> SELECT ENABLE_LOCAL_SEGMENTS();
 ENABLE_LOCAL_SEGMENTS
```

```
-----------------------
 ENABLED
(1 row)
```

REBALANCE_CLUSTER

Starts rebalancing data in the cluster synchronously. Rebalancing redistributes your database projections' data across all nodes, refreshes projections, sets the Ancient History Mark, and drops projections that are no longer needed. Rebalancing is useful after you:

- mark one or more nodes as ephemeral in preparation of removing them from the cluster, so that HP Vertica migrates the data on the ephemeral nodes away to other nodes.
- add one or more nodes to the cluster, so that HP Vertica can populate the empty nodes with data.
- change the scaling factor, which determines the number of storage containers used to store a projection across the database.

Since function runs the rebalance task synchronously, it does not return until the data has been rebalanced. Closing or dropping the session cancels the rebalance task.

Syntax

```
REBALANCE_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT REBALANCE_CLUSTER();
 REBALANCE_CLUSTER
-------------------
 REBALANCED
(1 row)
```

See Also

- Rebalancing Data Across Nodes
- ***START_REBALANCE_CLUSTER*** (page 537)
- ***CANCEL_REBALANCE_CLUSTER*** (page 450)

SET_SCALING_FACTOR

Sets the scaling factor used to determine the size of the storage containers used when rebalancing the database and when using local data segmentation is enabled. See Cluster Scaling for details.

Syntax

```
SET_SCALING_FACTOR(factor)
```

Parameters

factor	An integer value between 1 and 32. HP Vertica uses this value to calculate the number of storage containers each projection is broken into when rebalancing or when local data segmentation is enabled.

Note: Setting the scaling factor value too high can cause nodes to create too many small container files, greatly reducing efficiency and potentially causing a Too Many ROS containers error (also known as "ROS pushback"). The scaling factor should be set high enough so that rebalance can transfer local segments to satisfy the skew threshold, but small enough that the number of storage containers does not exceed ROS pushback. The number of storage containers should be greater than or equal to the number of partitions multiplied by the number local of segments (# storage containers >= # partitions * # local segments).

Privileges

Must be a superuser

Example

```
=> SELECT SET_SCALING_FACTOR(12);
 SET_SCALING_FACTOR
--------------------
 SET
(1 row)
```

START_REBALANCE_CLUSTER

Asynchronously starts a data rebalance task. Rebalancing redistributes your database projections' data across all nodes, refreshes projections, sets the Ancient History Mark, and drops projections that are no longer needed. Rebalancing is useful after you:

- mark one or more nodes as ephemeral in preparation of removing them from the cluster, so that HP Vertica migrates the data on the ephemeral nodes away to other nodes.
- add one or more nodes to the cluster, so that HP Vertica can populate the empty nodes with data.
- change the scaling factor, which determines the number of storage containers used to store a projection across the database.

Since this function starts the rebalance task in the background, it returns immediately after the task has started. Since it is a background task, rebalancing will continue even if the session that started it is closed. It even continues after a cluster recovery if the database shuts down while it is in progress. The only way to stop the task is by the CANCEL_REBALANCE_CLUSTER function.

Syntax

```
START_REBALANCE_CLUSTER()
```

Privileges

Must be a superuser

Example

```
=> SELECT START_REBALANCE_CLUSTER();
 START_REBALANCE_CLUSTER
-------------------------
 REBALANCING
(1 row)
```

See Also

- Rebalancing Data Across Nodes
- ***CANCEL_REBALANCE_CLUSTER*** (page 450)
- ***REBALANCE_CLUSTER*** (page 522)

Constraint Management Functions

This section contains constraint management functions specific to HP Vertica.

See also SQL system table ***V_CATALOG.TABLE_CONSTRAINTS*** (page 977)

ANALYZE_CONSTRAINTS

Analyzes and reports on constraint violations within the current schema search path, or external to that path if you specify a database name (noted in the syntax statement and parameter table).

You can check for constraint violations by passing arguments to the function as follows:

1. An empty argument (' '), which returns violations on all tables within the current schema
2. One argument, referencing a table
3. Two arguments, referencing a table name and a column or list of columns

Syntax

```
ANALYZE_CONSTRAINTS [ ( '' )
... | ( '[[db-name.]schema.]table [.column_name]' )
... | ( '[[db-name.]schema.]table' ,  'column' ) ]
```

Parameters

`('')`	Analyzes and reports on all tables within the current schema search path.
`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).

table	Analyzes and reports on all constraints referring to the specified table.
column	Analyzes and reports on all constraints referring to the specified table that contains the column.

Privileges

- SELECT privilege on table
- USAGE privilege on schema

Notes

ANALYZE_CONSTRAINTS() performs a lock in the same way that `SELECT * FROM t1` holds a lock on table `t1`. See ***LOCKS*** (page 1037) for additional information.

Detecting Constraint Violations During a Load Process

HP Vertica checks for constraint violations when queries are run, not when data is loaded. To detect constraint violations as part of the load process, use a ***COPY*** (page 699) statement with the NO COMMIT option. By loading data without committing it, you can run a post-load check of your data using the ANALYZE_CONSTRAINTS function. If the function finds constraint violations, you can roll back the load because you have not committed it.

If ANALYZE_CONSTRAINTS finds violations, such as when you insert a duplicate value into a primary key, you can correct errors using the following functions. Effects last until the end of the session only:

- SELECT ***DISABLE_DUPLICATE_KEY_ERROR*** (page 466)
- SELECT ***REENABLE_DUPLICATE_KEY_ERROR*** (page 522)

Return Values

ANALYZE_CONSTRAINTS returns results in a structured set (see table below) that lists the schema name, table name, column name, constraint name, constraint type, and the column values that caused the violation.

If the result set is empty, then no constraint violations exist; for example:

```
=> SELECT ANALYZE_CONSTRAINTS ('public.product_dimension', 'product_key');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
(0 rows)
```

The following result set shows a primary key violation, along with the value that caused the violation (`'10'`):

```
=> SELECT ANALYZE_CONSTRAINTS ('');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 store         t1           c1             pk_t1             PRIMARY           ('10')
(1 row)
```

The result set columns are described in further detail in the following table:

Column Name	Data Type	Description
Schema Name	VARCHAR	The name of the schema.
Table Name	VARCHAR	The name of the table, if specified.
Column Names	VARCHAR	Names of columns containing constraints. Multiple columns are in a comma-separated list: store_key, store_key, date_key,
Constraint Name	VARCHAR	The given name of the primary key, foreign key, unique, or not null constraint, if specified.
Constraint Type	VARCHAR	Identified by one of the following strings: 'PRIMARY KEY', 'FOREIGN KEY', 'UNIQUE', or 'NOT NULL'.
Column Values	VARCHAR	Value of the constraint column, in the same order in which Column Names contains the value of that column in the violating row. When interpreted as SQL, the value of this column forms a list of values of the same type as the columns in Column Names; for example: ('1'), ('1', 'z')

Understanding Function Failures

If ANALYZE_CONSTRAINTS() fails, HP Vertica returns an error identifying the failure condition. For example, if there are insufficient resources, the database cannot perform constraint checks and ANALYZE_CONSTRAINTS() fails.

If you specify the wrong table, the system returns an error message:

```
=> SELECT ANALYZE_CONSTRAINTS('abc');
   ERROR 2069: 'abc' is not a table in the current search_path
```

If you issue the function with incorrect syntax, the system returns an error message with a hint:

```
ANALYZE ALL CONSTRAINT;
```

Or

```
ANALYZE CONSTRAINT abc;
ERROR: ANALYZE CONSTRAINT is not supported.
HINT: You may consider using analyze_constraints().
```

If you run ANALYZE_CONSTRAINTS from a non-default locale, the function returns an error with a hint:

```
=> \locale LEN
INFO 2567:  Canonical locale: 'en'
Standard collation: 'LEN'
English
=> SELECT ANALYZE_CONSTRAINTS('t1');
```

```
   ERROR:  ANALYZE_CONSTRAINTS is currently not supported in non-default
   locales
   HINT:  Set the locale in this session to en_US@collation=binary using
   the
   command "\locale en_US@collation=binary"
```

Examples

Given the following inputs, HP Vertica returns one row, indicating one violation, because the same primary key value (10) was inserted into table t1 twice:

```
=> CREATE TABLE t1(c1 INT);
=> ALTER TABLE t1 ADD CONSTRAINT pk_t1 PRIMARY KEY (c1);
=> CREATE PROJECTION t1_p (c1) AS SELECT * FROM t1 UNSEGMENTED ALL NODES;
=> INSERT INTO t1 values (10);
=> INSERT INTO t1 values (10); --Duplicate primary key value
=> SELECT ANALYZE_CONSTRAINTS('t1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      t1           c1             pk_t1             PRIMARY           ('10')
(1 row)
```

If the second INSERT statement above had contained any different value, the result would have been 0 rows (no violations).

In the following example, create a table that contains three integer columns, one a unique key and one a primary key:

```
=> CREATE TABLE fact_1(
     f INTEGER,
     f_UK INTEGER UNIQUE,
     f_PK INTEGER PRIMARY KEY
   );
```

Issue a command that refers to a nonexistent table and column:

```
=> SELECT ANALYZE_CONSTRAINTS('f_BB');
   ERROR: 'f_BB' is not a table name in the current search path
```

Issue a command that refers to a nonexistent column:

```
=> SELECT ANALYZE_CONSTRAINTS('fact_1','x');
   ERROR 41614:  Nonexistent columns: 'x '
```

Insert some values into table `fact_1` and commit the changes:

```
=> INSERT INTO fact_1 values (1, 1, 1);
=> COMMIT;
```

Run ANALYZE_CONSTRAINTS on table `fact_1`. No constraint violations are reported:

```
=> SELECT ANALYZE_CONSTRAINTS('fact_1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
(0 rows)
```

Insert duplicate unique and primary key values and run ANALYZE_CONSTRAINTS on table `fact_1` again. The system shows two violations: one against the primary key and one against the unique key:

```
=> INSERT INTO fact_1 VALUES (1, 1, 1);
=> COMMIT;
=> SELECT ANALYZE_CONSTRAINTS('fact_1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
```

```
 public      | fact_1     | f_pk         | -               | PRIMARY         | ('1')
 public      | fact_1     | f_uk         | -               | UNIQUE          | ('1')
(2 rows)
```

The following command looks for constraint validations on only the unique key in the table `fact_1`, qualified with its schema name:

```
=> SELECT ANALYZE_CONSTRAINTS('public.fact_1', 'f_UK');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | fact_1     | f_uk         | C_UNIQUE        | UNIQUE          | ('1')
(1 row)
```

The following example shows that you can specify the same column more than once; ANALYZE_CONSTRAINTS, however, returns the violation only once:

```
=> SELECT ANALYZE_CONSTRAINTS('fact_1', 'f_PK, F_PK');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | fact_1     | f_pk         | C_PRIMARY       | PRIMARY         | ('1')
(1 row)
```

The following example creates a new dimension table, `dim_1`, and inserts a foreign key and different (character) data types:

```
=> CREATE TABLE dim_1 (b VARCHAR(3), b_PK VARCHAR(4), b_FK INTEGER REFERENCES fact_1(f_PK));
```

Alter the table to create a multicolumn unique key and multicolumn foreign key and create superprojections:

```
=> ALTER TABLE dim_1 ADD CONSTRAINT dim_1_multiuk PRIMARY KEY (b, b_PK);
```

The following command inserts a missing foreign key (0) into table `dim_1` and commits the changes:

```
=> INSERT INTO dim_1 VALUES ('r1', 'Xpk1', 0);
=> COMMIT;
```

Checking for constraints on the table `dim_1` in the `public` schema detects a foreign key violation:

```
=> SELECT ANALYZE_CONSTRAINTS('public.dim_1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | dim_1      | b_fk         | C_FOREIGN       | FOREIGN         | ('0')
(1 row)
```

Now add a duplicate value into the unique key and commit the changes:

```
=> INSERT INTO dim_1 values ('r2', 'Xpk1', 1);
=> INSERT INTO dim_1 values ('r1', 'Xpk1', 1);
=> COMMIT;
```

Checking for constraint violations on table `dim_1` detects the duplicate unique key error:

```
=> SELECT ANALYZE_CONSTRAINTS('dim_1');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | dim_1      | b, b_pk      | dim_1_multiuk   | PRIMARY         | ('r1', 'Xpk1')
 public      | dim_1      | b_fk         | C_FOREIGN       | FOREIGN         | ('0')
(2 rows)
```

Create a table with multicolumn foreign key and create the superprojections:

```
=> CREATE TABLE dim_2(z_fk1 VARCHAR(3), z_fk2 VARCHAR(4));
=> ALTER TABLE dim_2 ADD CONSTRAINT dim_2_multifk FOREIGN KEY (z_fk1, z_fk2) REFERENCES dim_1(b, b_PK);
```

Insert a foreign key that matches a foreign key in table `dim_1` and commit the changes:

```
=> INSERT INTO dim_2 VALUES ('r1', 'Xpk1');
=> COMMIT;
```

Checking for constraints on table `dim_2` detects no violations:

```
=> SELECT ANALYZE_CONSTRAINTS('dim_2');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
(0 rows)
```

Add a value that does not match and commit the change:

```
=> INSERT INTO dim_2 values ('r1', 'NONE');
=> COMMIT;
```

Checking for constraints on table `dim_2` detects a foreign key violation:

```
=> SELECT ANALYZE_CONSTRAINTS('dim_2');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | dim_2      | z_fk1, z_fk2 | dim_2_multifk   | FOREIGN         | ('r1', 'NONE')
(1 row)
```

Analyze all constraints on all tables:

```
=> SELECT ANALYZE_CONSTRAINTS('');
 Schema Name | Table Name | Column Names | Constraint Name | Constraint Type | Column Values
-------------+------------+--------------+-----------------+-----------------+---------------
 public      | dim_1      | b, b_pk      | dim_1_multiuk   | PRIMARY         | ('r1', 'Xpk1')
 public      | dim_1      | b_fk         | C_FOREIGN       | FOREIGN         | ('0')
 public      | dim_2      | z_fk1, z_fk2 | dim_2_multifk   | FOREIGN         | ('r1', 'NONE')
 public      | fact_1     | f_pk         | C_PRIMARY       | PRIMARY         | ('1')
 public      | fact_1     | f_uk         | C_UNIQUE        | UNIQUE          | ('1')
(5 rows)
```

To quickly clean up your database, issue the following command:

```
=> DROP TABLE fact_1 cascade;
=> DROP TABLE dim_1 cascade;
=> DROP TABLE dim_2 cascade;
```

To learn how to remove violating rows, see the ***DISABLE_DUPLICATE_KEY_ERROR*** (page 466) function.

See Also

Adding Constraints in the Administrator's Guide

COPY (page 699)

ALTER TABLE (page 672)

CREATE TABLE (page 770)

DISABLE_DUPLICATE_KEY_ERROR

Disables error messaging when HP Vertica finds duplicate PRIMARY KEY/UNIQUE KEY values at run time. Queries execute as though no constraints are defined on the schema. Effects are session scoped.

CAUTION: When called, DISABLE_DUPLICATE_KEY_ERROR() suppresses data integrity checking and can lead to incorrect query results. Use this function only after you insert duplicate primary keys into a dimension table in the presence of a pre-join projection. Then correct the violations and turn integrity checking back on with ***REENABLE_DUPLICATE_KEY_ERROR*** (page 522)().

Syntax

```
DISABLE_DUPLICATE_KEY_ERROR();
```

Privileges

Must be a superuser

Notes

The following series of commands create a table named `dim` and the corresponding projection:

```
CREATE TABLE dim (pk INTEGER PRIMARY KEY, x INTEGER);
CREATE PROJECTION dim_p (pk, x) AS SELECT * FROM dim ORDER BY x UNSEGMENTED ALL
NODES;
```

The next two statements create a table named `fact` and the pre-join projection that joins `fact` to `dim`.

```
CREATE TABLE fact(fk INTEGER REFERENCES dim(pk));
CREATE PROJECTION prejoin_p (fk, pk, x) AS SELECT * FROM fact, dim WHERE pk=fk ORDER
BY x;
```

The following statements load values into table `dim`. The last statement inserts a duplicate primary key value of 1:

```
INSERT INTO dim values (1,1);
INSERT INTO dim values (2,2);
INSERT INTO dim values (1,2); --Constraint violation
COMMIT;
```

Table `dim` now contains duplicate primary key values, but you cannot delete the violating row because of the presence of the pre-join projection. Any attempt to delete the record results in the following error message:

```
ROLLBACK:  Duplicate primary key detected in FK-PK join Hash-Join (x dim_p), value
1
```

In order to remove the constraint violation `(pk=1)`, use the following sequence of commands, which puts the database back into the state just before the duplicate primary key was added.

To remove the violation:

1. Save the original `dim` rows that match the duplicated primary key:
   ```
   CREATE TEMP TABLE dim_temp(pk integer, x integer);
   INSERT INTO dim_temp SELECT * FROM dim WHERE pk=1 AND x=1; -- original
   dim row
   ```
2. Temporarily disable error messaging on duplicate constraint values:
   ```
   SELECT DISABLE_DUPLICATE_KEY_ERROR();
   ```

Caution: Remember that running the `DISABLE_DUPLICATE_KEY_ERROR` function suppresses the enforcement of data integrity checking.

3 Remove the original row that contains duplicate values:

```
DELETE FROM dim WHERE pk=1;
```

4 Allow the database to resume data integrity checking:

```
SELECT REENABLE_DUPLICATE_KEY_ERROR();
```

5 Reinsert the original values back into the dimension table:

```
INSERT INTO dim SELECT * from dim_temp;
COMMIT;
```

6 Validate your dimension and fact tables.

If you receive the following error message, it means that the duplicate records you want to delete are not identical. That is, the records contain values that differ in at least one column that is not a primary key; for example, (1,1) and (1,2).

```
ROLLBACK: Delete: could not find a data row to delete (data integrity violation?)
```

The difference between this message and the rollback message in the previous example is that a fact row contains a foreign key that matches the duplicated primary key, which has been inserted. A row with values from the fact and dimension table is now in the pre-join projection. In order for the DELETE statement (Step 3 in the following example) to complete successfully, extra predicates are required to identify the original dimension table values (the values that are in the pre-join).

This example is nearly identical to the previous example, except that an additional INSERT statement joins the fact table to the dimension table by a primary key value of 1:

```
INSERT INTO dim  values (1,1);
INSERT INTO dim  values (2,2);
INSERT INTO fact values (1);   -- New insert statement joins fact with dim on
primary key value=1
INSERT INTO dim  values (1,2); -- Duplicate primary key value=1
COMMIT;
```

To remove the violation:

1 Save the original `dim` and `fact` rows that match the duplicated primary key:

```
CREATE TEMP TABLE dim_temp(pk integer, x integer);
CREATE TEMP TABLE fact_temp(fk integer);
INSERT INTO dim_temp SELECT * FROM dim WHERE pk=1 AND x=1; -- original
dim row
INSERT INTO fact_temp SELECT * FROM fact WHERE fk=1;
```

2 Temporarily suppresses the enforcement of data integrity checking:

```
SELECT DISABLE_DUPLICATE_KEY_ERROR();
```

3 Remove the duplicate primary keys. These steps also implicitly remove all fact rows with the matching foreign key.

   a) Remove the original row that contains duplicate values:

```
DELETE FROM dim WHERE pk=1 AND x=1;
```

Note: The extra predicate (`x=1`) specifies removal of the original (`1,1`) row, rather than the newly inserted (`1,2`) values that caused the violation.

b) Remove all remaining rows:

```
DELETE FROM dim WHERE pk=1;
```

4 Reenable integrity checking:

```
SELECT REENABLE_DUPLICATE_KEY_ERROR();
```

5 Reinsert the original values back into the fact and dimension table:

```
INSERT INTO dim SELECT * from dim_temp;
INSERT INTO fact SELECT * from fact_temp;
COMMIT;
```

Validate your dimension and fact tables.

See Also

ANALYZE_CONSTRAINTS (page 432)

REENABLE_DUPLICATE_KEY_ERROR (page 522)

LAST_INSERT_ID

Returns the last value of a column whose value is automatically incremented through the AUTO_INCREMENT or IDENTITY ***column-constraint*** (page 783). If multiple sessions concurrently load the same table, the returned value is the last value generated for an AUTO_INCREMENT column by an insert in that session.

Behavior Type

Volatile

Syntax

```
LAST_INSERT_ID()
```

Privileges

- Table owner
- USAGE privileges on schema

Notes

- This function works only with AUTO_INCREMENT and IDENTITY columns. See ***column-constraints*** (page 783) for the ***CREATE TABLE*** (page 770) statement.
- LAST_INSERT_ID does not work with sequence generators created through the ***CREATE SEQUENCE*** (page 765) statement.

Examples

Create a sample table called `customer4`.

```
=> CREATE TABLE customer4(
     ID IDENTITY(2,2),
     lname VARCHAR(25),
```

```
     fname VARCHAR(25),
     membership_card INTEGER
   );
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Gupta', 'Saleem', 475987);
```

Notice that the IDENTITY column has a seed of 2, which specifies the value for the first row loaded into the table, and an increment of 2, which specifies the value that is added to the IDENTITY value of the previous row.

Query the table you just created:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
(1 row)
```

Insert some additional values:

```
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Lee', 'Chen', 598742);
```

Call the LAST_INSERT_ID function:

```
=> SELECT LAST_INSERT_ID();
last_insert_id
---------------
              4
(1 row)
```

Query the table again:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
  4 | Lee   | Chen   |          598742
(2 rows)
```

Add another row:

```
=> INSERT INTO customer4(lname, fname, membership_card)
   VALUES ('Davis', 'Bill', 469543);
```

Call the LAST_INSERT_ID function:

```
=> SELECT LAST_INSERT_ID();
 LAST_INSERT_ID
----------------
             6
(1 row)
```

Query the table again:

```
=> SELECT * FROM customer4;
 ID | lname | fname  | membership_card
----+-------+--------+-----------------
  2 | Gupta | Saleem |          475987
  4 | Lee   | Chen   |          598742
```

```
  6 | Davis | Bill    |          469543
(3 rows)
```

See Also

ALTER SEQUENCE (page 669)

CREATE SEQUENCE (page 765)

DROP SEQUENCE (page 822)

V_CATALOG.SEQUENCES (page 969)

Using Sequences and Sequence Privileges in the Administrator's Guide

REENABLE_DUPLICATE_KEY_ERROR

Restores the default behavior of error reporting by reversing the effects of DISABLE_DUPLICATE_KEY_ERROR. Effects are session scoped.

Syntax

```
REENABLE_DUPLICATE_KEY_ERROR();
```

Privileges

Must be a superuser

Examples

For examples and usage see ***DISABLE_DUPLICATE_KEY_ERROR*** (page 466).

See Also

ANALYZE_CONSTRAINTS (page 432)

Data Collector Functions

The HP Vertica Data Collector is a utility that extends ***system table*** (page 933) functionality by providing a framework for recording events. It gathers and retains monitoring information about your database cluster and makes that information available in system tables, requiring few configuration parameter tweaks, and having negligible impact on performance.

Collected data is stored on disk in the `DataCollector` directory under the HP Vertica /catalog path. You can use the information the Data Collector retains to query the past state of system tables and extract aggregate information, as well as do the following:

- See what actions users have taken
- Locate performance bottlenecks
- Identify potential improvements to HP Vertica configuration

Data Collector works in conjunction with an advisor tool called Workload Analyzer, which intelligently monitors the performance of SQL queries and workloads and recommends tuning actions based on observations of the actual workload history.

By default, Data Collector is on and retains information for all sessions. If performance issues arise, a superuser can disable DC. See Data Collector Parameters and Enabling and Disabling Data Collector in the Administrator's Guide.

This section describes the Data Collection control functions.

Related topics

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining monitoring information and Analyzing Workloads in the Administrator's Guide

CLEAR_DATA_COLLECTOR

Clears all memory and disk records on the Data Collector tables and functions and resets collection statistics in the V_MONITOR.DATA_COLLECTOR system table. A superuser can clear Data Collector data for all components or specify an individual component

After you clear the DataCollector log, the information is no longer available for querying.

Syntax

```
CLEAR_DATA_COLLECTOR( [ 'component' ] )
```

Parameters

component	Clears memory and disk records for the specified component only. If you provide no argument, the function clears all Data Collector memory and disk records for all components. For the current list of component names, query the ***V_MONITOR.DATA_COLLECTOR*** (page 1002) system table.

Privileges

Must be a superuser

Example

The following command clears memory and disk records for the ResourceAcquisitions component:

```
=> SELECT clear_data_collector('ResourceAcquisitions');
 clear_data_collector
----------------------
 CLEAR
```

```
(1 row)
```

The following command clears data collection for all components on all nodes:

```
=> SELECT clear_data_collector();
 clear_data_collector
----------------------
 CLEAR
(1 row)
```

See Also

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining Monitoring Information in the Administrator's Guide

DATA_COLLECTOR_HELP

Returns online usage instructions about the Data Collector, the V_MONITOR.DATA_COLLECTOR system table, and the Data Collector control functions.

Syntax

```
DATA_COLLECTOR_HELP()
```

Privileges

None

Returns

Invoking DATA_COLLECTOR_HELP() returns the following information:

```
=> SELECT DATA_COLLECTOR_HELP();

   Usage Data Collector
   The data collector retains history of important system activities.
   This data can be used as a reference of what actions have been taken
     by users, but it can also be used to locate performance bottlenecks,
     or identify potential improvements to the Vertica configuration.
   This data is queryable via Vertica system tables.


   The list of data collector components, and some statistics, can be found using:
   SELECT * FROM v_monitor.data_collector;

   The amount of data retained can be controlled with:
      set_data_collector_policy(<component>,
                           <memory retention (KB)>,
                           <disk retention (KB)>);


   The current retention policy for a component can be queried with:
      get_data_collector_policy(<component>);
```

```
Data on disk is kept in the "DataCollector" directory under the Vertica
catalog path. This directory also contains instructions on how to load
the monitoring data into another Vertica database.


Additional commands can be used to affect the data collection logs.
The log can be cleared with:
  clear_data_collector([<optional component>]);
The log can be synchronized with the disk storage using:
  flush_data_collector([<optional component>]);
```

Note: Data Collector works in conjunction with the Workload Analyzer, an advisor tool that intelligently monitors the performance of SQL queries and workloads and recommends tuning actions based on observations of the actual workload history.

See Also

V_MONITOR.DATA_COLLECTOR (page 1002)

V_MONITOR.TUNING_RECOMMENDATIONS (page 1120)

Analyzing Workloads in the Administrator's Guide

Retaining Monitoring Information in the Administrator's Guide

FLUSH_DATA_COLLECTOR

Waits until memory logs are moved to disk and then flushes the Data Collector, synchronizing the log with the disk storage. A superuser can flush Data Collector information for an individual component or for all components.

Syntax

```
FLUSH_DATA_COLLECTOR( [ 'component' ] )
```

Parameters

component	Flushes the specified component. If you provide no argument, the function flushes the Data Collector in full. For the current list of component names, query the V_MONITOR.DATA_COLLECTOR system table.

Privileges

Must be a superuser

Example

The following command flushes the Data Collector for the ResourceAcquisitions component:

```
=> SELECT flush_data_collector('ResourceAcquisitions');
 flush_data_collector
```

```
----------------------
 FLUSH
(1 row)
```

The following command flushes data collection for all components:

```
=> SELECT flush_data_collector();
 flush_data_collector
----------------------
 FLUSH
(1 row)
```

See Also

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining Monitoring Information in the Administrator's Guide

GET_DATA_COLLECTOR_POLICY

Retrieves a brief statement about the retention policy for the specified component.

Syntax

```
GET_DATA_COLLECTOR_POLICY( 'component' )
```

Parameters

component	Returns the retention policy for the specified component. For a current list of component names, query the V_MONITOR.DATA_COLLECTOR system table

Privileges

None

Example

The following query returns the history of all resource acquisitions by specifying the `ResourceAcquisitions` component:

```
=> SELECT get_data_collector_policy('ResourceAcquisitions');
          get_data_collector_policy
-------------------------------------------
 1000KB kept in memory, 10000KB kept on disk.
(1 row)
```

See Also

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining Monitoring Information in the Administrator's Guide

SET_DATA_COLLECTOR_POLICY

Sets the retention policy for the specified component on all nodes. Failed nodes receive the setting when they rejoin the cluster.

Syntax

```
SET_DATA_COLLECTOR_POLICY('component', 'memoryKB', 'diskKB' )
```

Parameters

component	Returns the retention policy for the specified component.
memoryKB	Specifies the memory size to retain in kilobytes.
diskKB	Specifies the disk size in kilobytes.

Privileges

Must be a superuser

Notes

- Only a superuser can configure the Data Collector.
- Before you change a retention policy, view its current setting by calling the GET_DATA_COLLECTOR_POLICY() function.
- If you don't know the name of a component, query the V_MONITOR.DATA_COLLECTOR system table for a list; for example:

  ```
  => SELECT DISTINCT component, description FROM data_collector ORDER BY
  1 ASC;
  ```

Example

The following command returns the retention policy for the ResourceAcquisitions component:

```
=> SELECT get_data_collector_policy('ResourceAcquisitions');
          get_data_collector_policy
-----------------------------------------------
 1000KB kept in memory, 10000KB kept on disk.
(1 row)
```

This command changes the memory and disk setting for ResourceAcquisitions from their current setting of 1,000 KB and 10,000 KB to 1,500 KB and 25,000 KB, respectively:

```
=> SELECT set_data_collector_policy('ResourceAcquisitions', '1500', '25000');
 set_data_collector_policy
---------------------------
 SET
(1 row)
```

To verify the setting, call the GET_DATA_COLLECTOR_POLICY() function on the specified component:

```
=> SELECT get_data_collector_policy('ResourceAcquisitions');
```

```
          get_data_collector_policy
-----------------------------------------------
 1500KB kept in memory, 25000KB kept on disk.
(1 row)
```

See Also

GET_DATA_COLLECTOR_POLICY() (page 496)

V_MONITOR.DATA_COLLECTOR (page 1002)

Retaining Monitoring Information in the Administrator's Guide

Database Management Functions

This section contains the database management functions specific to HP Vertica.

CLEAR_RESOURCE_REJECTIONS

Clears the content of the ***RESOURCE_REJECTIONS*** (page 1089) and ***DISK_RESOURCE_REJECTIONS*** (page 1013) system tables. Normally, these tables are only cleared during a node restart. This function lets you clear the tables whenever you need. For example, you might want to clear the system tables after you resolved a disk space issue that was causing disk resource rejections.

Syntax

```
CLEAR_RESOURCE_REJECTIONS();
```

Privileges

Must be a superuser

Example

The following command clears the content of the RESOURCE_REJECTIONS and DISK_RESOURCE_REJECTIONS system tables:

```
=> SELECT clear_resource_rejections();
 clear_resource_rejections
---------------------------
 OK
(1 row)
```

See Also

DISK_RESOURCE_REJECTIONS (page 1013)

RESOURCE_REJECTIONS (page 1089)

DUMP_LOCKTABLE

Returns information about deadlocked clients and the resources they are waiting for.

Syntax

```
DUMP_LOCKTABLE()
```

Privileges

None

Notes

Use DUMP_LOCKTABLE if HP Vertica becomes unresponsive:

1. Open an additional vsql connection.
2. Execute the query:

   ```
   => SELECT DUMP_LOCKTABLE();
   ```

   The output is written to vsql. See Monitoring the Log Files.

You can also see who is connected using the following command:

```
=> SELECT * FROM SESSIONS;
```

Close all sessions using the following command:

```
=>SELECT CLOSE_ALL_SESSIONS();
```

Close a single session using the following command:

How to close a single session:

```
=> SELECT CLOSE_SESSION('session_id');
```

You get the session_id value from the ***V_MONITOR.SESSIONS*** (page 1095) system table.

See Also

CLOSE_ALL_SESSIONS (page 461)

CLOSE_SESSION (page 458)

V_MONITOR.LOCKS (page 1037)

V_MONITOR.SESSIONS (page 1095)

DUMP_PARTITION_KEYS

Dumps the partition keys of all projections in the system.

Syntax

```
DUMP_PARTITION_KEYS( )
```

Note: ROS's of partitioned tables without partition keys are ignored by the tuple mover and are not merged during automatic tuple mover operations.

Privileges

None; however function dumps only the tables for which user has SELECT privileges.

Example

```
=> SELECT DUMP_PARTITION_KEYS( );
Partition keys on node v_vmart_node0001
  Projection 'states_b0'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: NH
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: MA
  Projection 'states_b1'
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: VT
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: ME
   Storage [ROS container]
     No of partition keys: 1
     Partition keys: CT
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

V_MONITOR.PARTITIONS (page 1051)

Partitioning Tables in the Administrator's Guide

EXPORT_TABLES

Generates a SQL script that can be used to recreate a logical schema (schemas, tables, constraints, and views) on a different cluster.

Syntax

```
EXPORT_TABLES ( [ 'destination' ] , [ 'scope' ] )
```

Parameters

destination	Specifies the path and name of the SQL output file. An empty string (''), which is the default, outputs the script to standard output. The function writes the script to the catalog directory if no destination is specified. If you specify a file that does not exist, the function creates one. If the file pre-exists, the function silently overwrites its contents.
scope	Determines the tables to export. Specify the scope as follows: ▪ An empty string (' ') — exports all non-virtual table objects to which the user has access, including table schemas, sequences, and constraints. Exporting all non-virtual objects is the default scope, and what the function exports if you do not specify a scope. ▪ A comma-delimited list of objects, which can include the following: ▪ —' [*dbname*.][*schema*.]*object* '— matches the named objects, which can be schemas, tables, or views, in the schema. You can optionally qualify a schema with a database prefix, and objects with a schema. You cannot pass constraints as individual arguments. ▪ —' [*dbname*.]*object* '— matches a named object, which can be a schema, table, or view. You can optionally qualify a schema with a database prefix, and an object with its schema. For a schema, HP Vertica exports all non-virtual objects to which the user has access within the schema. If a schema and table both have the same name, the schema takes precedence.

Privileges

None; however:

- Function exports only the objects visible to the user
- Only a superuser can export output to file

Example

The following example exports the `store_orders_fact` table of the `store` schema (in the current database) to standard output:

```
=> SELECT EXPORT_TABLES(' ','store.store_orders_fact');
```

EXPORT_TABLES returns an error if:

- You explicitly specify an object that does not exist
- The current user does not have access to a specified object

See Also

EXPORT_CATALOG (page 487)

EXPORT_OBJECTS

Exporting Tables in the Administrator's Guide

HAS_ROLE

Indicates, by a boolean value, whether a role has been assigned to a user. This function is useful for letting you check your own role membership.

Behavior Type

Stable

Syntax 1

```
HAS_ROLE( [ 'user_name' ,] 'role_name' );
```

Syntax 2

```
HAS_ROLE( 'role_name' );
```

Parameters

user_name	[Optional] The name of a user to look up. Currently, only a superuser can supply the user_name argument.
role_name	The name of the role you want to verify has been granted.

Privileges

Users can check their own role membership by calling HAS_ROLE('role_name'), but only a superuser can look up other users' memberships using the optional user_name parameter.

Notes

You can query V_CATALOG system tables ***ROLES*** (page 967), ***GRANTS*** (page 944), and ***USERS*** (page 985) to show any directly-assigned roles; however, these tables do not indicate whether a role is available to a user when roles may be available through other roles (indirectly).

Examples

User Bob wants to see if he has been granted the commentor role:

```
=> SELECT HAS_ROLE('commentor');
```

Output *t* for true indicates that Bob has been assigned the commentor role:

```
 HAS_ROLE
----------
 t
(1 row)
```

In the following function call, a superuser checks if the logadmin role has been granted to user Bob:

```
=> SELECT HAS_ROLE('Bob', 'logadmin');
 HAS_ROLE
----------
```

```
 t
(1 row)
```

To view the names of all roles users can access, along with any roles that have been assigned to those roles, query the ***V_CATALOG.ROLES*** (page 967) system table. An asterisk in the output means role granted WITH ADMIN OPTION.

```
=> SELECT * FROM roles;
      name        |    assigned_roles
------------------+----------------------
 public           |
 dbadmin          | dbduser*
 pseudosuperuser | dbadmin
 dbduser          |
 logreader        |
 logwriter        |
 logadmin         | logreader, logwriter
(7 rows)
```

Note: The dbduser role in output above is internal only; you can ignore it.

See Also

GRANTS (page 944)

ROLES (page 967)

USERS (page 985)

Managing Privileges and Roles and Viewing a User's Role in the Administrator's Guide

SET_CONFIG_PARAMETER

Use SET_CONFIG_PARAMETER to set a configuration parameter.

Note: HP Vertica is designed to operate with minimal configuration changes. Use this function sparingly and carefully follow any documented guidelines for that parameter.

If a node is down when you invoke this function, changes will occur on UP nodes only. You must re-issue the function after down nodes recover in order for the changes to take effect on those nodes. Alternatively, use the Administration Tools to copy the files. Redistributing Configuration Files to Nodes.

Syntax

```
SET_CONFIG_PARAMETER( 'parameter', value )
```

Parameters

parameter	Specifies the name of the parameter value being set. See Configuration Parameters in the Administrator's Guide for a list of supported parameters, their function, and usage

	examples.
value	Specifies the value of the supplied parameter argument. Syntax for this argument will vary depending upon the parameter and its expected data type. For strings, you must enclose the argument in single quotes; integer arguments can be unquoted.

You can also query the ***V_MONITOR.CONFIGURATION_PARAMETERS*** (page 996) system table to get information about configuration parameters currently in use by the system.

For example, the following statement returns all current configuration parameters in HP Vertica:

```
=> SELECT * FROM CONFIGURATION_PARAMETERS;
```

SET_LOGLEVEL

Use SET_LOGLEVEL to set the logging level in the HP Vertica database log files.

Syntax

```
SELECT SET_LOGLEVEL(n)
```

Parameters

n	Logging Level	Description
0	DISABLE	No logging
1	CRITICAL	Errors requiring database recovery
2	WARNING	Errors indicating problems of lesser magnitude
3	INFO	Informational messages
4	DEBUG	Debugging messages
5	TRACE	Verbose debugging messages
6	TIMING	Verbose debugging messages

Privileges

Must be a superuser

SHUTDOWN

Forces a database to shut down, even if there are users connected.

Syntax

```
SHUTDOWN ( [ 'false' | 'true' ] )
```

Parameters

false	[Default] Returns a message if users are connected. Has the same effect as supplying no parameters.
true	Performs a moveout operation and forces the database to shut down, disallowing further connections.

Privileges

Must be a superuser

Notes

- Quotes around the `true` or `false` arguments are optional.
- Issuing the shutdown command without arguments or with the default (false) argument returns a message if users are connected, and the shutdown fails. If no users are connected, the database performs a moveout operation and shuts down.
- Issuing the `SHUTDOWN('true')` command forces the database to shut down whether users are connected or not.
- You can check the status of the shutdown operation in the `vertica.log file`: `2010-03-09 16:51:52.625 unknown:0x7fc6d6d2e700 [Init] <INFO> Shutdown complete. Exiting.`
- As an alternative to SHUTDOWN(), you can also temporarily set MaxClientSessions to 0 and then use CLOSE_ALL_SESSIONS(). New client connections cannot connect unless they connect using the dbadmin account. See ***CLOSE_ALL_SESSIONS*** (page 461) for details.

Examples

The following command attempts to shut down the database. Because users are connected, the command fails:

```
=> SELECT SHUTDOWN('false');
NOTICE:  Cannot shut down while users are connected
          SHUTDOWN
----------------------------
 Shutdown: aborting shutdown
(1 row)
```

Note that `SHUTDOWN()` and `SHUTDOWN('false')` perform the same operation:

```
=> SELECT SHUTDOWN();
NOTICE:  Cannot shut down while users are connected
          SHUTDOWN
----------------------------
 Shutdown: aborting shutdown
(1 row)
```

Using the `'true'` parameter forces the database to shut down, even though clients might be connected:

```
=> SELECT SHUTDOWN('true');
```

```
           SHUTDOWN
----------------------------
 Shutdown: moveout complete
(1 row)
```

See Also

SESSIONS (page 1095)

Epoch Management Functions

This section contains the epoch management functions specific to HP Vertica.

ADVANCE_EPOCH

Manually closes the current epoch and begins a new epoch.

Syntax

```
ADVANCE_EPOCH ( [ integer ] )
```

Parameters

integer	Specifies the number of epochs to advance.

Privileges

Must be a superuser

Note

This function is primarily maintained for backward compatibility with earlier versions of HP Vertica.

Example

The following command increments the epoch number by 1:

```
=> SELECT ADVANCE_EPOCH(1);
```

See Also

ALTER PROJECTION (page 659)

GET_AHM_EPOCH

Returns the number of the epoch in which the Ancient History Mark is located. Data deleted up to and including the AHM epoch can be purged from physical storage.

Syntax

```
GET_AHM_EPOCH()
```

Note: The AHM epoch is 0 (zero) by default (purge is disabled).

Privileges

None

Examples

```
SELECT GET_AHM_EPOCH();
    get_ahm_epoch
----------------------
 Current AHM epoch: 0
(1 row)
```

GET_AHM_TIME

Returns a TIMESTAMP value representing the Ancient History Mark. Data deleted up to and including the AHM epoch can be purged from physical storage.

Syntax

```
GET_AHM_TIME()
```

Privileges

None

Examples

```
SELECT GET_AHM_TIME();
                  GET_AHM_TIME
-------------------------------------------------
 Current AHM Time: 2010-05-13 12:48:10.532332-04
(1 row)
```

See Also

SET DATESTYLE (page 903) for information about valid ***TIMESTAMP*** (page 97) values.

GET_CURRENT_EPOCH

Returns the number of the current epoch. The epoch into which data (COPY, INSERT, UPDATE, and DELETE operations) is currently being written. The current epoch advances automatically every three minutes.

Syntax

```
GET_CURRENT_EPOCH()
```

Privileges

None

Examples

```
SELECT GET_CURRENT_EPOCH();
 GET_CURRENT_EPOCH
-------------------
               683
(1 row)
```

GET_LAST_GOOD_EPOCH

Returns the number of the last good epoch. A term used in manual recovery, LGE (Last Good Epoch) refers to the most recent epoch that can be recovered.

Syntax
```
GET_LAST_GOOD_EPOCH()
```

Privileges
None

Examples
```
SELECT GET_LAST_GOOD_EPOCH();
 GET_LAST_GOOD_EPOCH
---------------------
                 682
(1 row)
```

MAKE_AHM_NOW
Sets the Ancient History Mark (AHM) to the greatest allowable value, and lets you drop any projections that existed before the issue occurred.

Caution: This function is intended for use by Administrators only.

Syntax
```
MAKE_AHM_NOW ( [ true ] )
```

Parameters

true	[Optional] Allows AHM to advance when nodes are down. **Note:** If the AHM is advanced after the last good epoch of the failed nodes, those nodes must recover all data from scratch. Use with care.

Privileges
Must be a superuser

Notes
- The MAKE_AHM_NOW function performs the following operations:
  - Advances the epoch.
  - Performs a moveout operation on all projections.
  - Sets the AHM to LGE — at least to the current epoch at the time MAKE_AHM_NOW() was issued.
- All history is lost and you cannot perform historical queries prior to the current epoch.

Example
```
=> SELECT MAKE_AHM_NOW();
          MAKE_AHM_NOW
-------------------------------
 AHM set (New AHM Epoch: 683)
(1 row)
```
The following command allows the AHM to advance, even though node 2 is down:

```
=> SELECT MAKE_AHM_NOW(true);
WARNING:  Received no response from v_vmartdb_node0002 in get cluster LGE
WARNING:  Received no response from v_vmartdb_node0002 in get cluster LGE
WARNING:  Received no response from v_vmartdb_node0002 in set AHM
         MAKE_AHM_NOW
------------------------------
 AHM set (New AHM Epoch: 684)
(1 row)
```

See Also

DROP PROJECTION (page 818)

MARK_DESIGN_KSAFE (page 510)

SET_AHM_EPOCH (page 527)

SET_AHM_TIME (page 528)

SET_AHM_EPOCH

Sets the Ancient History Mark (AHM) to the specified epoch. This function allows deleted data up to and including the AHM epoch to be purged from physical storage.

SET_AHM_EPOCH is normally used for testing purposes. Consider ***SET_AHM_TIME*** (page 528) instead, which is easier to use.

Syntax

```
SET_AHM_EPOCH ( epoch, [ true ] )
```

Parameters

epoch	Specifies one of the following: ▪ The number of the epoch in which to set the AHM ▪ Zero (0) (the default) disables ***purge*** (page 517)
true	Optionally allows the AHM to advance when nodes are down. **Note:** If the AHM is advanced after the last good epoch of the failed nodes, those nodes must recover all data from scratch. Use with care.

Privileges

Must be a superuser

Notes

If you use SET_AHM_EPOCH , the number of the specified epoch must be:

- Greater than the current AHM epoch
- Less than the current epoch
- Less than or equal to the cluster last good epoch (the minimum of the last good epochs of the individual nodes in the cluster)

- Less than or equal to the cluster refresh epoch (the minimum of the refresh epochs of the individual nodes in the cluster)

Use the ***SYSTEM*** (page 1111) table to see current values of various epochs related to the AHM; for example:

```
=> SELECT * from SYSTEM;
-[ RECORD 1 ]------------+--------------------------
current_timestamp        | 2009-08-11 17:09:54.651413
current_epoch            | 1512
ahm_epoch                | 961
last_good_epoch          | 1510
refresh_epoch            | -1
designed_fault_tolerance | 1
node_count               | 4
node_down_count          | 0
current_fault_tolerance  | 1
catalog_revision_number  | 1590
wos_used_bytes           | 0
wos_row_count            | 0
ros_used_bytes           | 41490783
ros_row_count            | 1298104
total_used_bytes         | 41490783
total_row_count          | 1298104
```

All nodes must be up. You cannot use SET_AHM_EPOCH when any node in the cluster is down, except by using the optional *true* parameter.

When a node is down and you issue `SELECT MAKE_AHM_NOW()`, the following error is printed to the vertica.log:

```
Some nodes were excluded from setAHM. If their LGE is before the AHM they will perform full recovery.
```

Examples

The following command sets the AHM to a specified epoch of 12:

```
=> SELECT SET_AHM_EPOCH(12);
```

The following command sets the AHM to a specified epoch of 2 and allows the AHM to advance despite a failed node:

```
=> SELECT SET_AHM_EPOCH(2, true);
```

See Also

MAKE_AHM_NOW (page 508)

SET_AHM_TIME (page 528)

SYSTEM (page 1111)

SET_AHM_TIME

Sets the Ancient History Mark (AHM) to the epoch corresponding to the specified time on the initiator node. This function allows historical data up to and including the AHM epoch to be purged from physical storage.

Syntax

```
SET_AHM_TIME ( time , [ true ] )
```

Parameters

time	Is a ***TIMESTAMP*** (page 97) value that is automatically converted to the appropriate epoch number.
true	[Optional] Allows the AHM to advance when nodes are down. **Note:** If the AHM is advanced after the last good epoch of the failed nodes, those nodes must recover all data from scratch.

Privileges

Must be a superuser

Notes

- SET_AHM_TIME returns a TIMESTAMP WITH TIME ZONE value representing the end point of the AHM epoch.
- You cannot change the AHM when any node in the cluster is down, except by using the optional *true* parameter.
- When a node is down and you issue `SELECT MAKE_AHM_NOW()`, the following error is printed to the vertica.log:

  ```
  Some nodes were excluded from setAHM. If their LGE is before the AHM they will perform full recovery.
  ```

Examples

Epochs depend on a configured epoch advancement interval. If an epoch includes a three-minute range of time, the purge operation is accurate only to within minus three minutes of the specified timestamp:

```
=> SELECT SET_AHM_TIME('2008-02-27 18:13');
           set_ahm_time
-----------------------------------
 AHM set to '2008-02-27 18:11:50-05'
(1 row)
```

Note: The -05 part of the output string is a time zone value, an offset in hours from UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, or GMT).

In the above example, the actual AHM epoch ends at 18:11:50, roughly one minute before the specified timestamp. This is because SET_AHM_TIME selects the epoch that ends at or before the specified timestamp. It does not select the epoch that ends after the specified timestamp because that would purge data deleted as much as three minutes after the AHM.

For example, using only hours and minutes, suppose that epoch 9000 runs from 08:50 to 11:50 and epoch 9001 runs from 11:50 to 15:50. `SET_AHM_TIME('11:51')` chooses epoch 9000 because it ends roughly one minute before the specified timestamp.

In the next example, if given an environment variable set as `date =`date`;` the following command fails if a node is down:

```
=> SELECT SET_AHM_TIME('$date');
```

In order to force the AHM to advance, issue the following command instead:

```
=> SELECT SET_AHM_TIME('$date', true);
```

See Also

MAKE_AHM_NOW (page 508)

SET_AHM_EPOCH (page 527) for a description of the range of valid epoch numbers.

SET DATESTYLE (page 903) for information about specifying a ***TIMESTAMP*** (page 97) value.

License Management Functions

This section contains function that monitor HP Vertica license status and compliance.

AUDIT

Estimates the raw data size of a database, a schema, a projection, or a table as it is counted in an audit of the database size.

The AUDIT function estimates the size using the same data sampling method as the audit that HP Vertica performs to determine if a database is compliant with the database size allowances in its license. The results of this function are not considered when HP Vertica determines whether the size of the database complies with the HP Vertica license's data allowance. See How HP Vertica Calculates Database Size in the Administrator's Guide for details.

Note: This function can only audit the size of tables, projections, schemas, and databases which the user has permission to access. If a non-superuser attempts to audit the entire database, the audit will only estimate the size of the data that the user is allowed to read.

Syntax

```
AUDIT([name] [, granularity] [, error_tolerance [, confidence_level]])
```

Parameters

name	Specifies the schema, projection, or table to audit. Enter name as a string, in single quotes ('). If the name string is empty ('), the entire database is audited.

granularity	Indicates the level at which the audit reports its results. The recognized levels are: ▪ `'schema'` ▪ `'table'` ▪ `'projection'` By default, the granularity is the same level as *name*. For example, if *name* is a schema, then the size of the entire schema is reported. If you instead specify `'table'` as the granularity, AUDIT reports the size of each table in the schema. The granularity must be finer than that of object: specifying `'schema'` for an audit of a table has no effect. The results of an audit with a granularity are reported in the V_CATALOG.USER_AUDITS system table.
error_tolerance	Specifies the percentage margin of error allowed in the audit estimate. Enter the tolerance value as a decimal number, between 0 and 100. The default value is 5, for a 5% margin of error. **Note:** The lower this value is, the more resources the audit uses since it will perform more data sampling. Setting this value to 0 results in a full audit of the database, which is very resource intensive, as all of the data in the database is analyzed. Doing a full audit of the database significantly impacts performance and is not recommended on a production database.
confidence_level	Specifies the statistical confidence level percentage of the estimate. Enter the confidence value as a decimal number, between 0 and 100. The default value is 99, indicating a confidence level of 99%. **Note:** The higher the confidence value, the more resources the function uses since it will perform more data sampling. Setting this value to 1 results in a full audit of the database, which is very resource intensive, as all of the database is analyzed. Doing a full audit of the database significantly impacts performance and is not recommended on a production database.

Permissions

- SELECT privilege on table
- USAGE privilege on schema

Note: AUDIT() works only on the tables where the user calling the function has SELECT permissions.

Notes

Due to the iterative sampling used in the auditing process, making the error tolerance a small fraction of a percent (0.00001, for example) can cause the AUDIT function to run for a longer period than a full database audit.

Examples

To audit the entire database:

```
=> SELECT AUDIT('');
  AUDIT
----------
 76376696
(1 row)
```

To audit the database with a 25% error tolerance:

```
=> SELECT AUDIT('',25);
  AUDIT
----------
 75797126
(1 row)
```

To audit the database with a 25% level of tolerance and a 90% confidence level:

```
=> SELECT AUDIT('',25,90);
  AUDIT
----------
 76402672
(1 row)
```

To audit just the online_sales schema in the VMart example database:

```
VMart=> SELECT AUDIT('online_sales');
  AUDIT
----------
 35716504
(1 row)
```

To audit the online_sales schema and report the results by table:

```
=> SELECT AUDIT('online_sales','table');
                             AUDIT
-------------------------------------------------------------
 See table sizes in v_catalog.user_audits for schema online_sales
(1 row)

=> \x
Expanded display is on.

=> SELECT * FROM user_audits WHERE object_schema = 'online_sales';
-[ RECORD 1 ]-------------------------+-----------------------------
size_bytes                            | 64960
user_id                               | 45035996273704962
user_name                             | dbadmin
```

```
object_id                              | 45035996273717636
object_type                            | TABLE
object_schema                          | online_sales
object_name                            | online_page_dimension
audit_start_timestamp                  | 2011-04-05 09:24:48.224081-04
audit_end_timestamp                    | 2011-04-05 09:24:48.337551-04
confidence_level_percent               | 99
error_tolerance_percent                | 5
used_sampling                          | f
confidence_interval_lower_bound_bytes | 64960
confidence_interval_upper_bound_bytes | 64960
sample_count                           | 0
cell_count                             | 0
-[ RECORD 2 ]-------------------------+-----------------------------
size_bytes                             | 20197
user_id                                | 45035996273704962
user_name                              | dbadmin
object_id                              | 45035996273717640
object_type                            | TABLE
object_schema                          | online_sales
object_name                            | call_center_dimension
audit_start_timestamp                  | 2011-04-05 09:24:48.340206-04
audit_end_timestamp                    | 2011-04-05 09:24:48.365915-04
confidence_level_percent               | 99
error_tolerance_percent                | 5
used_sampling                          | f
confidence_interval_lower_bound_bytes | 20197
confidence_interval_upper_bound_bytes | 20197
sample_count                           | 0
cell_count                             | 0
-[ RECORD 3 ]-------------------------+-----------------------------
size_bytes                             | 35614800
user_id                                | 45035996273704962
user_name                              | dbadmin
object_id                              | 45035996273717644
object_type                            | TABLE
object_schema                          | online_sales
object_name                            | online_sales_fact
audit_start_timestamp                  | 2011-04-05 09:24:48.368575-04
audit_end_timestamp                    | 2011-04-05 09:24:48.379307-04
confidence_level_percent               | 99
error_tolerance_percent                | 5
used_sampling                          | t
confidence_interval_lower_bound_bytes | 34692956
confidence_interval_upper_bound_bytes | 36536644
sample_count                           | 10000
cell_count                             | 9000000
```

AUDIT_LICENSE_SIZE

Triggers an immediate audit of the database size to determine if it is in compliance with the raw data storage allowance included in your HP Vertica license. The audit is performed in the background, so this function call returns immediately. To see the results of the audit when it is done, use the ***GET_COMPLIANCE_STATUS*** (page 494) function.

Syntax

```
AUDIT_LICENSE_SIZE()
```

Privileges

Must be a superuser

Example

```
=> SELECT audit_license_size();
 audit_license_size
--------------------
 Service hurried
(1 row)
```

AUDIT_LICENSE_TERM

Triggers an immediate audit to determine if the HP Vertica license has expired. The audit happens in the background, so this function returns immediately. To see the result of the audit, use the ***GET_COMPLIANCE_STATUS*** (page 494) function.

Syntax

```
AUDIT_LICENSE_TERM()
```

Privileges

Must be a superuser

Example

```
=> SELECT AUDIT_LICENSE_TERM();
 AUDIT_LICENSE_TERM
--------------------
 Service hurried
(1 row)
```

GET_AUDIT_TIME

Reports the time when the automatic audit of database size occurs. HP Vertica performs this audit if your HP Vertica license includes a data size allowance. For details of this audit, see Managing Your License Key in the Administrator's Guide. To change the time the audit runs, use the ***SET_AUDIT_TIME*** (page 530) function.

Syntax

```
GET_AUDIT_TIME()
```

Privileges

None

Example

```
=> SELECT get_audit_time();
                   get_audit_time
-----------------------------------------------------
 The audit is scheduled to run at 11:59 PM each day.
(1 row)
```

GET_COMPLIANCE_STATUS

Displays whether your database is in compliance with your HP Vertica license agreement. This information includes the results of HP Vertica's most recent audit of the database size (if your license has a data allowance as part of its terms), and the license term (if your license has an end date).

The information displayed by GET_COMPLIANCE_STATUS includes:

- The estimated size of the database (see How HP Vertica Calculates Database Size in the Administrator's Guide for an explanation of the size estimate).
- The raw data size allowed by your HP Vertica license.
- The percentage of your allowance that your database is currently using.
- The date and time of the last audit.
- Whether your database complies with the data allowance terms of your license agreement.
- The end date of your license.
- How many days remain until your license expires.

Note: If your license does not have a data allowance or end date, some of the values may not appear in the output for GET_COMPLIANCE_STATUS.

If the audit shows your license is not in compliance with your data allowance, you should either delete data to bring the size of the database under the licensed amount, or upgrade your license. If your license term has expired, you should contact HP immediately to renew your license. See Managing Your License Key in the Administrator's Guide for further details.

Syntax

```
GET_COMPLIANCE_STATUS()
```

Privileges

None

Example

```
                    GET_COMPLIANCE_STATUS
```

```
-------------------------------------------------------------------------------
---
 Raw Data Size: 2.00GB +/- 0.003GB
 License Size : 4.000GB
 Utilization  : 50%
 Audit Time   : 2011-03-09 09:54:09.538704+00
 Compliance Status : The database is in compliance with respect to raw data size.

 License End Date: 04/06/2011
 Days Remaining: 28.59
(1 row)
```

DISPLAY_LICENSE

Returns the terms of your HP Vertica license. The information this function displays is:

- The start and end dates for which the license is valid (or "Perpetual" if the license has no expiration).
- The number of days you are allowed to use HP Vertica after your license term expires (the grace period)
- The amount of data your database can store, if your license includes a data allowance.

Syntax

```
DISPLAY_LICENSE()
```

Privileges

None

Examples

```
=> SELECT DISPLAY_LICENSE();
                  DISPLAY_LICENSE
-----------------------------------------------------
 HP Vertica Systems, Inc.
1/1/2011
12/31/2011
30
50TB

(1 row)
```

SET_AUDIT_TIME

Sets the time that HP Vertica performs automatic database size audit to determine if the size of the database is compliant with the raw data allowance in your HP Vertica license. Use this function if the audits are currently scheduled to occur during your database's peak activity time. This is normally not a concern, since the automatic audit has little impact on database performance.

Note: Audits are scheduled by the preceding audit, so changing the audit time does not affect the next scheduled audit. For example, if your next audit is scheduled to take place at 11:59PM and you use SET_AUDIT_TIME to change the audit schedule 3AM, the previously scheduled 11:59PM audit still runs. As that audit finishes, it schedules the next audit to occur at 3AM.

If you want to prevent the next scheduled audit from running at its scheduled time, you can change the scheduled time using SET_AUDIT_TIME then manually trigger an audit to run immediately using ***AUDIT_LICENSE_SIZE*** (page 450). As the manually-triggered audit finishes, it schedules the next audit to occur at the time you set using SET_AUDIT_TIME (effectively overriding the previously scheduled audit).

Syntax

```
SET_AUDIT_TIME(time)
```

time	A string containing the time in `'HH:MM AM/PM'` format (for example, `'1:00 AM'`) when the audit should run daily.

Privileges

Must be a superuser

Example

```
=> SELECT SET_AUDIT_TIME('3:00 AM');
                              SET_AUDIT_TIME
-------------------------------------------------------------------------
 The scheduled audit time will be set to 3:00 AM after the next audit.
(1 row)
```

Partition Management Functions

This section contains partition management functions specific to HP Vertica.

DROP_PARTITION

Forces the partition of projections (if needed) and then drops the specified partition.

Syntax

```
DROP_PARTITION ( table_name , partition_value [ , ignore_moveout_errors,
reorganize_data ])
```

Parameters

table-name	Specifies the name of the table. **Note**: The *table_name* argument cannot be used as a dimension table in a pre-joined projection and cannot contain projections that are not up to date (have not been refreshed).

partition_value	The key of the partition to drop. For example: `DROP_PARTITION('trade', 2006);`
ignore_moveout_errors	Optional Boolean, defaults to `false`. ▪ `true`—Ignores any WOS moveout errors and forces the operation to continue. Set this parameter to `true` only if there is no WOS data for the partition. ▪ `false` (or omit)—Displays any moveout errors and aborts the operation on error. **Note**: If you set this parameter to `true` and the WOS includes data for the partition in WOS, partition data in WOS is not dropped.
reorganize_data	Optional Boolean, defaults to `false`. ▪ `true`—Reorganizes the data if it is not organized, and then drops the partition. ▪ `false`—Does not attempt to reorganize the data before dropping the partition. If this parameter is `false` and the function encounters a ROS without partition keys, an error occurs.

Permissions

- Table owner
- USAGE privilege on schema that contains the table

Notes and Restrictions

The results of a DROP_PARTITION call go into effect immediately. If you drop a partition using DROP_PARTITION and then try to add data to a partition with the same name, HP Vertica creates a new partition.

If the operation cannot obtain an ***O Lock*** (page 1037) on the table(s), HP Vertica attempts to close any internal Tuple Mover (TM) sessions running on the same table(s) so that the operation can proceed. Explicit TM operations that are running in user sessions are not closed. If an explicit TM operation is running on the table, then the operation cannot proceed until the explicit TM operation completes.

In general, if a ROS container has data that belongs to *n*+1 partitions and you want to drop a specific partition, the DROP_PARTITION operation:

1. Forces the partition of data into two containers where
   - One container holds the data that belongs to the partition that is to be dropped.
   - Another container holds the remaining *n* partitions.
2. Drops the specified partition.

You can also use the ***MERGE_PARTITIONS*** (page 513) function to merge ROS containers that have data belonging to partitions in a specified partition key range; for example, `[partitionKeyFrom, partitionKeyTo]`.

DROP_PARTITION forces a moveout if there is data in the WOS (WOS is not partition aware).

DROP_PARTITION acquires an exclusive lock on the table to prevent DELETE | UPDATE | INSERT | COPY statements from affecting the table, as well as any SELECT statements issued at SERIALIZABLE isolation level.

You cannot perform a DROP_PARTITION operation on tables with projections that are not up to date (have not been refreshed).

DROP_PARTITION fails if you do not set the optional third parameter to *true* and the function encounters ROS's that do not have partition keys.

Examples

Using the example schema in Defining Partitions, the following command explicitly drops the 2009 partition key from table `trade`:

```
 SELECT DROP_PARTITION('trade', 2009);
  DROP_PARTITION
------------------
 Partition dropped
(1 row)
```

Here, the partition key is specified:

```
SELECT DROP_PARTITION('trade', EXTRACT('year' FROM '2009-01-01'::date));
  DROP_PARTITION
------------------
 Partition dropped
(1 row)
```

The following example creates a table called dates and partitions the table by year:

```
CREATE TABLE dates (
    year INTEGER NOT NULL,
    month VARCHAR(8) NOT NULL)
PARTITION BY year * 12 + month;
```

The following statement drops the partition using a constant for Oct 2010 (2010*12 + 10 = 24130):

```
 SELECT DROP_PARTITION('dates', '24130');
  DROP_PARTITION
------------------
 Partition dropped
(1 row)
```

Alternatively, the expression can be placed in line: `SELECT DROP_PARTITION('dates', 2010*12 + 10);`

The following command first reorganizes the data if it is unpartitioned and then explicitly drops the 2009 partition key from table `trade`:

```
SELECT DROP_PARTITION('trade', 2009, false, true);

  DROP_PARTITION
------------------
 Partition dropped
(1 row)
```

See Also

Dropping Partitions in the Administrator's Guide

ADVANCE EPOCH (page 429)

ALTER PROJECTION (page 659)

COLUMN_STORAGE (page 992)

CREATE TABLE (page 770)

DO_TM_TASK (page 471)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

MERGE_PARTITIONS (page 513)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

PROJECTIONS (page 961)

DUMP_PROJECTION_PARTITION_KEYS

Dumps the partition keys of the specified projection.

Syntax

```
DUMP_PROJECTION_PARTITION_KEYS(  'projection_name' )
```

Parameters

projection_name	Specifies the name of the projection.

Privileges

- SELECT privilege on table
- USAGE privileges on schema

Example

The following example creates a simple table called `states` and partitions the data by state:

```
=> CREATE TABLE states (
       year INTEGER NOT NULL,
       state VARCHAR NOT NULL)
      PARTITION BY state;


=> CREATE PROJECTION states_p (state, year) AS SELECT * FROM states
```

```
   ORDER BY state, year UNSEGMENTED ALL NODES;
```

Now dump the partition key of the specified projection:

```
=> SELECT DUMP_PROJECTION_PARTITION_KEYS( 'states_p_node0001' );
Partition keys on node helios_node0001
 Projection 'states_p_node0001'
 No of partition keys: 1
Partition keys on node helios_node0002
...
(1 row)
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

PROJECTIONS (page 961) system table

Partitioning Tables in the Administrator's Guide

DUMP_TABLE_PARTITION_KEYS

Dumps the partition keys of all projections anchored on the specified table.

Syntax

```
DUMP_TABLE_PARTITION_KEYS ( 'table_name' )
```

Parameters

table_name	Specifies the name of the table.

Privileges

- SELECT privilege on table
- USAGE privileges on schema

Example

The following example creates a simple table called `states` and partitions the data by state:

```
=> CREATE TABLE states (
       year INTEGER NOT NULL,
       state VARCHAR NOT NULL)
   PARTITION BY state;
```

```
=> CREATE PROJECTION states_p (state, year) AS SELECT * FROM states
   ORDER BY state, year UNSEGMENTED ALL NODES;
```

Now dump the partition keys of all projections anchored on table `states`:

```
=> SELECT DUMP_TABLE_PARTITION_KEYS( 'states' );
Partition keys on helios_node0001
Projection 'states_p_node0004'
No of partition keys: 1
Projection 'states_p_node0003'
No of partition keys: 1
Projection 'states_p_node0002'
No of partition keys: 1
Projection 'states_p_node0001'
No of partition keys: 1
Partition keys on helios_node0002
...
(1 row)
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 480)

DUMP_PROJECTION_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

Partitioning Tables in the Administrator's Guide

MERGE_PARTITIONS

Merges ROS containers that have data belonging to partitions in a specified partition key range: `partitionKeyFrom` to `partitionKeyTo`.

Note: This function is deprecated in HP Vertica 7.0.

Syntax

```
MERGE_PARTITIONS ( table_name , partition_key_from , partition_key_to )
```

Parameters

table_name	Specifies the name of the table
partition_key_from	Specifies the start point of the partition

partition_key_to	Specifies the end point of the partition

Privileges

- Table owner
- USAGE privilege on schema that contains the table

Notes

- You cannot run MERGE_PARTITIONS() on a table with data that is not reorganized. You must reorganize the data first using `ALTER_TABLE table REORGANIZE`, or `PARTITION_TABLE(table)`.
- The edge values are included in the range, and `partition_key_from` must be less than or equal to `partition_key_to`.
- Inclusion of partitions in the range is based on the application of less than(<)/greater than(>) operators of the corresponding data type.

  Note: No restrictions are placed on a partition key's data type.
- If `partition_key_from` is the same as `partition_key_to`, all ROS containers of the partition key are merged into one ROS.

Examples

```
=> SELECT MERGE_PARTITIONS('T1', '200', '400');
=> SELECT MERGE_PARTITIONS('T1', '800', '800');
=> SELECT MERGE_PARTITIONS('T1', 'CA', 'MA');
=> SELECT MERGE_PARTITIONS('T1', 'false', 'true');
=> SELECT MERGE_PARTITIONS('T1', '06/06/2008', '06/07/2008');
=> SELECT MERGE_PARTITIONS('T1', '02:01:10', '04:20:40');
=> SELECT MERGE_PARTITIONS('T1', '06/06/2008 02:01:10', '06/07/2008 02:01:10');
=> SELECT MERGE_PARTITIONS('T1', '8 hours', '1 day 4 hours 20 seconds');
```

MOVE_PARTITIONS_TO_TABLE

Moves partitions from a source table to a target table. The target table must have the same projection column definitions, segmentation, and partition expressions as the source table. If the target table does not exist, the function creates a new table based on the source definition. The function requires both minimum and maximum range values, indicating what partition values to move.

Syntax

```
MOVE_PARTITIONS_TO_TABLE ( '[[db-name.]schema.]source_table',
'min_range_value', 'max_range_value', '[[db-name.]schema.]target_table' )
```

Parameters

[[db-name.]schema.]source_table	The source table (optionally qualified), from which you want to move partitions.

min_range_value	The minimum value in the partition to move.
max_range_value	The maximum value of the partition being moved.
target_table	The table to which the partitions are being moved.

Privileges

- Table owner
- If target table is created as part of moving partitions, the new table has the same owner as the target. If the target table exists, user must have own the target table, and have ability to call this function.

Example

If you call `move_partitions_to_table` and the destination table does not exist, the function will create the table automatically:

```
VMART=> select move_partitions_to_table ('prod_trades', '200801', '200801',
'partn_backup.trades_200801');
                          move_partitions_to_table
-------------------------------------------------------------------------------
 1 distinct partition values moved at epoch 15. Effective move epoch: 14.

(1 row)
```

See Also

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

Moving Partitions and Creating a Table Like Another in the Administrator's Guide

PARTITION_PROJECTION

Forces a split of ROS containers of the specified projection.

Syntax

```
PARTITION_PROJECTION ( '[[db-name.]schema.]projection_name' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see

	Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection_name	Specifies the name of the projection.

Privileges

- Table owner
- USAGE privilege on schema

Notes

Partitioning expressions take immutable functions only, in order that the same information be available across all nodes.

PARTITION_PROJECTION() is similar to ***PARTITION_TABLE*** (page 516)(), except that PARTITION_PROJECTION works only on the specified projection, instead of the table.

Users must have USAGE privilege on schema that contains the table.PARTITION_PROJECTION() purges data while partitioning ROS containers if deletes were applied before the AHM epoch.

Example

The following command forces a split of ROS containers on the `states_p_node01` projection:

```
=> SELECT PARTITION_PROJECTION ('states_p_node01');
  partition_projection
-----------------------
 Projection partitioned
(1 row)
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_TABLE (page 516)

Partitioning Tables in the Administrator's Guide

PARTITION_TABLE

Forces the system to break up any ROS containers that contain multiple distinct values of the partitioning expression. Only ROS containers with more than one distinct value participate in the split.

Syntax

```
PARTITION_TABLE ( '[[db-name.]schema.]table_name' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	Specifies the name of the table.

Privileges

- Table owner
- USAGE privilege on schema

Notes

PARTITION_TABLE is similar to ***PARTITION_PROJECTION*** (page 515), except that PARTITION_TABLE works on the specified table.

Users must have USAGE privilege on schema that contains the table.Partitioning functions take immutable functions only, in order that the same information be available across all nodes.

Example

The following example creates a simple table called `states` and partitions data by state.

```
=> CREATE TABLE states (
          year INTEGER NOT NULL,
          state VARCHAR NOT NULL)
   PARTITION BY state;
=> CREATE PROJECTION states_p (state, year) AS
   SELECT * FROM states
   ORDER BY state, year UNSEGMENTED ALL NODES;
```

Now issue the command to partition table `states`:

```
=> SELECT PARTITION_TABLE('states');
                  PARTITION_TABLE
-----------------------------------------------------
```

```
partition operation for projection 'states_p_node0004'
partition operation for projection 'states_p_node0003'
partition operation for projection 'states_p_node0002'
partition operation for projection 'states_p_node0001'
(1 row)
```

See Also

DO_TM_TASK (page 471)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

Partitioning Tables in the Administrator's Guide

PURGE_PARTITION

Purges a table partition of deleted rows. Similar to PURGE and PURGE_PROJECTION, this function removes deleted data from physical storage so that the disk space can be reused. It only removes data from the AHM epoch and earlier.

Syntax

```
PURGE_PARTITION ( '[[db_name.]schema_name.]table_name', partition_key )
```

Parameters

`[[db_name.]schema_name.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	The name of the partitioned table
partition_key	The key of the partition to be purged of deleted rows

Privileges

- Table owner
- USAGE privilege on schema

Example

The following example lists the count of deleted rows for each partition in a table, then calls PURGE_PARTITION to purge the deleted rows from the data.

```
=> SELECT partition_key,table_schema,projection_name,sum(deleted_row_count) AS deleted_row_count
FROM partitions
-> GROUP BY partition_key,table_schema,projection_name  ORDER BY partition_key;

 partition_key | table_schema | projection_name | deleted_row_count
---------------+--------------+-----------------+-------------------
 0             | public       | t_super         |                 2
 1             | public       | t_super         |                 2
 2             | public       | t_super         |                 2
 3             | public       | t_super         |                 2
 4             | public       | t_super         |                 2
 5             | public       | t_super         |                 2
 6             | public       | t_super         |                 2
 7             | public       | t_super         |                 2
 8             | public       | t_super         |                 2
 9             | public       | t_super         |                 1
(10 rows)


=> SELECT PURGE_PARTITION('t',5); -- Purge partition with key 5.
                          purge_partition
-------------------------------------------------------------------------
 Task: merge partitions
(Table: public.t) (Projection: public.t_super)

(1 row)


=> SELECT partition_key,table_schema,projection_name,sum(deleted_row_count) AS deleted_row_count
FROM partitions
-> GROUP BY partition_key,table_schema,projection_name ORDER BY partition_key;

 partition_key | table_schema | projection_name | deleted_row_count
---------------+--------------+-----------------+-------------------
 0             | public       | t_super         |                 2
 1             | public       | t_super         |                 2
 2             | public       | t_super         |                 2
 3             | public       | t_super         |                 2
 4             | public       | t_super         |                 2
 5             | public       | t_super         |                 0
 6             | public       | t_super         |                 2
 7             | public       | t_super         |                 2
 8             | public       | t_super         |                 2
 9             | public       | t_super         |                 1
(10 rows)
```

See Also

PURGE (page 517)

PURGE_PROJECTION (on page 520)

PURGE_TABLE (page 520)

MERGE_PARTITIONS (page 513)

STORAGE_CONTAINERS (page 1098)

Profiling Functions

This section contains profiling functions specific to HP Vertica.

CLEAR_PROFILING

HP Vertica stores profiled data is in memory, so depending on how much data you collect, profiling could be memory intensive. You can use this function to clear profiled data from memory.

Syntax

```
CLEAR_PROFILING( 'profiling-type' )
```

Parameters

profiling-type	The type of profiling data you want to clear. Can be one of: ▪ session—clears profiling for basic session parameters and lock time out data ▪ query—clears profiling for general information about queries that ran, such as the query strings used and the duration of queries ▪ ee—clears profiling for information about the execution run of each query

Example

The following statement clears profiled data for queries:

```
=> SELECT CLEAR_PROFILING('query');
```

See also

DISABLE_PROFILING (page 469)

ENABLE_PROFILING (page 483)

Profiling Database Performance in the Administrator's Guide

DISABLE_PROFILING

Disables profiling for the profiling type you specify.

Syntax

```
DISABLE_PROFILING( 'profiling-type' )
```

Parameters

profiling-type	The type of profiling data you want to disable. Can be one of: ▪ session—disables profiling for basic session parameters and lock time out data ▪ query—disables profiling for general information about queries that ran, such as the query strings used and the duration of queries ▪ ee—disables profiling for information about the execution run of each query

Example

The following statement disables profiling on query execution runs:

```
=> SELECT DISABLE_PROFILING('ee');
   DISABLE_PROFILING
-----------------------
 EE Profiling Disabled
(1 row)
```

See also

CLEAR_PROFILING (page 455)

ENABLE_PROFILING (page 483)

Profiling Database Performance in the Administrator's Guide

ENABLE_PROFILING

Enables profiling for the profiling type you specify.

Note: HP Vertica stores profiled data is in memory, so depending on how much data you collect, profiling could be memory intensive.

Syntax

```
ENABLE_PROFILING( 'profiling-type' )
```

Parameters

profiling-type	The type of profiling data you want to enable. Can be one of: ▪ session—enables profiling for basic session parameters and lock time out data ▪ query—enables profiling for general information about queries that ran, such as the query strings used and the duration of queries ▪ ee—enables profiling for information about the execution run of each query

Example

The following statement enables profiling on query execution runs:

```
=> SELECT ENABLE_PROFILING('ee');
   ENABLE_PROFILING
----------------------
 EE Profiling Enabled
(1 row)
```

See also

CLEAR_PROFILING (page 455)

DISABLE_PROFILING (page 469)

Profiling Database Performance in the Administrator's Guide

Projection Management Functions

This section contains projection management functions specific to HP Vertica.

See also the following SQL system tables:

- ***V_CATALOG.PROJECTIONS*** (page 961)
- ***V_CATALOG.PROJECTION_COLUMNS*** (page 955)
- ***V_MONITOR.PROJECTION_REFRESHES*** (page 1056)
- ***V_MONITOR.PROJECTION_STORAGE*** (page 1059)

EVALUATE_DELETE_PERFORMANCE

Evaluates projections for potential ***DELETE*** (page 807) performance issues. If there are issues found, a warning message is displayed. For steps you can take to resolve delete and update performance issues, see Optimizing Deletes and Updates for Performance in the Administrator's Guide. This function uses data sampling to determine whether there are any issues with a projection. Therefore, it does not generate false-positives warnings, but it can miss some cases where there are performance issues.

Note: Optimizing for delete performance is the same as optimizing for update performance. So, you can use this function to help optimize a projection for updates as well as deletes.

Syntax

```
EVALUATE_DELETE_PERFORMANCE ( 'target' )
```

Parameters

target	The name of a projection or table. If you supply the name of a projection, only that projection is evaluated for DELETE performance issues. If you supply the name of a table, then all of the projections anchored to the table will be evaluated for issues. If you do not provide a projection or table name, EVALUATE_DELETE_PERFORMANCE examines all of the projections that you can access for DELETE performance issues. Depending on the size you your database, this may take a long time.

Privileges

None

Note: When evaluating multiple projections, EVALUATE_DELETE_PERFORMANCE reports up to ten projections that have issues, and refers you to a table that contains the full list of issues it has found.

Example

The following example demonstrates how you can use EVALUATE_DELETE_PERFORMANCE to evaluate your projections for slow DELETE performance.

```
=> create table example (A int, B int,C int);
CREATE TABLE
=> create projection one_sort (A,B,C) as (select A,B,C from example) order by A;
CREATE PROJECTION
=> create projection two_sort (A,B,C) as (select A,B,C from example) order by A,B;
CREATE PROJECTION
=> select evaluate_delete_performance('one_sort');
            evaluate_delete_performance
-----------------------------------------------
 No projection delete performance concerns found.

(1 row)

=> select evaluate_delete_performance('two_sort');
            evaluate_delete_performance
-----------------------------------------------
 No projection delete performance concerns found.

(1 row)
```

The previous example showed that there was no structural issue with the projection that would cause poor DELETE performance. However, the data contained within the projection can create potential delete issues if the sorted columns do not uniquely identify a row or small number of rows. In the following example, Perl is used to populate the table with data using a nested series of loops. The inner loop populates column C, the middle loop populates column B, and the outer loop populates column A. The result is column A contains only three distinct values (0, 1, and 2), while column B slowly varies between 20 and 0 and column C changes in each row. EVALUATE_DELETE_PERFORMANCE is run against the projections again to see if the data within the projections causes any potential DELETE performance issues.

```
=> \! perl -e 'for ($i=0; $i<3; $i++) { for ($j=0; $j<21; $j++) { for ($k=0; $k<19; $k++) { printf
"%d,%d,%d\n", $i,$j,$k;}}}' | /opt/vertica/bin/vsql -c "copy example from stdin delimiter ',' direct;"
```

```
Password:

=> select * from example;

 A | B  | C
---+----+----
 0 | 20 | 18
 0 | 20 | 17
 0 | 20 | 16
 0 | 20 | 15
 0 | 20 | 14
 0 | 20 | 13
 0 | 20 | 12
 0 | 20 | 11
 0 | 20 | 10
 0 | 20 |  9
 0 | 20 |  8
 0 | 20 |  7
 0 | 20 |  6
 0 | 20 |  5
 0 | 20 |  4
 0 | 20 |  3
 0 | 20 |  2
 0 | 20 |  1
 0 | 20 |  0
 0 | 19 | 18
 1157 rows omitted
 2 |  1 |  0
 2 |  0 | 18
 2 |  0 | 17
 2 |  0 | 16
 2 |  0 | 15
 2 |  0 | 14
 2 |  0 | 13
 2 |  0 | 12
 2 |  0 | 11
 2 |  0 | 10
 2 |  0 |  9
 2 |  0 |  8
 2 |  0 |  7
 2 |  0 |  6
 2 |  0 |  5
 2 |  0 |  4
 2 |  0 |  3
 2 |  0 |  2
 2 |  0 |  1
 2 |  0 |  0


=> SELECT COUNT (*) FROM example;
 COUNT
-------
  1197
(1 row)

=> SELECT COUNT (DISTINCT A) FROM example;
 COUNT
-------
     3
(1 row)


=> select evaluate_delete_performance('one_sort');
            evaluate_delete_performance
-------------------------------------------------
 Projection exhibits delete performance concerns.

(1 row)
```

```
release=> select evaluate_delete_performance('two_sort');
            evaluate_delete_performance
-------------------------------------------------
 No projection delete performance concerns found.

(1 row)
```

The one_sort projection has potential delete issues since it only sorts on column A which has few distinct values. This means that each value in the sort column corresponds to many rows in the projection, which negatively impacts DELETE performance. Since the two_sort projection is sorted on columns A and B, each combination of values in the two sort columns identifies just a few rows, allowing deletes to be performed faster.

Not supplying a projection name results in all of the projections you can access being evaluated for DELETE performance issues.

```
=> select evaluate_delete_performance();
                                                    evaluate_delete_performance
-------------------------------------------------------------------------------------------------------
------------------------------
 The following projection exhibits delete performance concerns:
        "public"."one_sort"
See v_catalog.projection_delete_concerns for more details.

(1 row)
```

GET_PROJECTION_STATUS

Returns information relevant to the status of a projection.

Syntax

```
GET_PROJECTION_STATUS ( '[[db-name.]schema-name.]projection' );
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection	Is the name of the projection for which to display status. When using more than one schema, specify the schema that contains the projection, as noted above.

Privileges

None

Description

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 the projection is segmented
- The number and names of buddy projections
- Whether the projection is safe
- Whether the projection is up-to-date
- Whether statistics have been computed for the projection

Notes

- You can use GET_PROJECTION_STATUS to monitor the progress of a projection data refresh. See ***ALTER PROJECTION*** (page 659).
- To view a list of the nodes in a database, use the View Database Command in the Administration Tools.

Examples

```
=> SELECT GET_PROJECTION_STATUS('public.customer_dimension_site01');

                                    GET_PROJECTION_STATUS
-------------------------------------------------------------------------------------------------
 Current system K is 1.
# of Nodes: 4.
public.customer_dimension_site01 [Segmented: No] [Seg Cols: ] [K: 3]
[public.customer_dimension_site04, public.customer_dimension_site03,
public.customer_dimension_site02] [Safe: Yes] [UptoDate: Yes][Stats: Yes]
```

See Also

ALTER PROJECTION (page 659)

GET_PROJECTIONS (page 499)

GET_PROJECTIONS, GET_TABLE_PROJECTIONS

Note: This function was formerly named GET_TABLE_PROJECTIONS(). HP Vertica still supports the former function name.

Returns information relevant to the status of a table:

- The current K-safety status of the database
- The number of sites (nodes) in the database
- The number of projections for which the specified table is the anchor table
- For each projection:
  - The projection's buddy projections
  - Whether the projection is segmented
  - Whether the projection is safe
  - Whether the projection is up-to-date

Syntax

```
GET_PROJECTIONS ( '[[db-name.]schema-name.]table' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table	Is the name of the table for which to list projections. When using more than one schema, specify the schema that contains the table.

Privileges

None

Notes

- You can use GET_PROJECTIONS to monitor the progress of a projection data refresh. See ***ALTER PROJECTION*** (page 659).
- To view a list of the nodes in a database, use the View Database Command in the Administration Tools.

Examples

The following example gets information about the store_dimension table in the VMart schema:

```
=> SELECT GET_PROJECTIONS('store.store_dimension');
-------------------------------------------------------------------------------
Current system K is 1.
# of Nodes: 4.
Table store.store_dimension has 4 projections.

Projection Name: [Segmented] [Seg Cols] [# of Buddies] [Buddy Projections] [Safe] [UptoDate]
-----------------------------------------------------
store.store_dimension_node0004 [Segmented: No] [Seg Cols: ] [K: 3] [store.store_dimension_node0003,
store.store_dimension_node0002, store.store_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
store.store_dimension_node0003 [Segmented: No] [Seg Cols: ] [K: 3] [store.store_dimension_node0004,
store.store_dimension_node0002, store.store_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
store.store_dimension_node0002 [Segmented: No] [Seg Cols: ] [K: 3] [store.store_dimension_node0004,
store.store_dimension_node0003, store.store_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
store.store_dimension_node0001 [Segmented: No] [Seg Cols: ] [K: 3] [store.store_dimension_node0004,
store.store_dimension_node0003, store.store_dimension_node0002] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]

(1 row)
```

See Also

ALTER PROJECTION (page 659)

GET_PROJECTION_STATUS (page 498)

REFRESH

Performs a synchronous, optionally-targeted refresh of a specified table's projections.

Information about a refresh operation—whether successful or unsuccessful—is maintained in the ***PROJECTION_REFRESHES*** (page 1056) system table until either the ***CLEAR_PROJECTION_REFRESHES*** (page 455)() function is executed or the storage quota for the table is exceeded. The `PROJECTION_REFRESHES.IS_EXECUTING` column returns a boolean value that indicates whether the refresh is currently running (t) or occurred in the past (f).

Syntax

```
REFRESH ( '[[db-name.]schema.]table_name [ , ... ]' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	Specifies the name of a specific table containing the projections to be refreshed. The REFRESH() function attempts to refresh all the tables provided as arguments in parallel. Such calls will be part of the Database Designer deployment (and deployment script). When using more than one schema, specify the schema that contains the table, as noted above.

Returns

Column Name	Description
`Projection Name`	The name of the projection that is targeted for refresh.
`Anchor Table`	The name of the projection's associated anchor table.
`Status`	The status of the projection: ▪ Queued — Indicates that a projection is queued for refresh. ▪ Refreshing — Indicates that a refresh for a projection is

	in process. ▪ Refreshed — Indicates that a refresh for a projection has successfully completed. ▪ Failed — Indicates that a refresh for a projection did not successfully complete.
Refresh Method	The method used to refresh the projection: ▪ Buddy – Uses the contents of a buddy to refresh the projection. This method maintains historical data. This enables the projection to be used for historical queries. ▪ Scratch – Refreshes the projection without using a buddy. This method does not generate historical data. This means that the projection cannot participate in historical queries from any point before the projection was refreshed.
Error Count	The number of times a refresh failed for the projection.
Duration (sec)	The length of time that the projection refresh ran in seconds.

Privileges

REFRESH() works only if invoked on tables owned by the calling user.

Notes

- Unlike START_REFRESH(), which runs in the background, REFRESH() runs in the foreground of the caller's session.
- The REFRESH() function refreshes only the projections in the specified table.
- If you run REFRESH() without arguments, it refreshes all non up-to-date projections. If the function returns a header string with no results, then no projections needed refreshing.

Example

The following command refreshes the projections in tables `t1` and `t2`:

```
=> SELECT REFRESH('t1, t2');
refresh
-------------------------------------------------------------------------------
 Refresh completed with the following outcomes:
Projection Name: [Anchor Table] [Status] [Refresh Method] [Error Count] [Duration (sec)]
-------------------------------------------------------------------------------
"public"."t1_p": [t1] [refreshed] [scratch] [0] [0]
"public"."t2_p": [t2] [refreshed] [scratch] [0] [0]
```

This next command shows that only the projection on table `t` was refreshed:

```
=> SELECT REFRESH('allow, public.deny, t');"
refresh
-------------------------------------------------------------------------------
 Refresh completed with the following outcomes:
Projection Name: [Anchor Table] [Status] [Refresh Method] [Error Count] [Duration (sec)]
-------------------------------------------------------------------------------
"n/a"."n/a": [n/a] [failed: insufficient permissions on table "allow"] [] [1] [0]
"n/a"."n/a": [n/a] [failed: insufficient permissions on table "public.deny"] [] [1] [0]
"public"."t_p1": [t] [refreshed] [scratch] [0] [0]
```

See Also

CLEAR_PROJECTION_REFRESHES (page 455)

PROJECTION_REFRESHES (page 1056)

START_REFRESH (page 538)

Clearing PROJECTION_REFRESHES History in the Administrator's Guide

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

- When a design is deployed through the Database Designer, it is automatically refreshed. See Deploying Designs in the Administrator's Guide.
- All nodes must be up in order to start a refresh.
- START_REFRESH() has no effect if a refresh is already running.
- A refresh is run asynchronously.
- Shutting down the database ends the refresh.
- To view the progress of the refresh, see the ***PROJECTION_REFRESHES*** (page 1056) and ***PROJECTIONS*** (page 961) system tables.
- If a projection is updated from scratch, the data stored in the projection represents the table columns as of the epoch in which the refresh commits. As a result, the query optimizer might not choose the new projection for AT EPOCH queries that request historical data at epochs older than the refresh epoch of the projection. Projections refreshed from buddies retain history and can be used to answer historical queries.

Privileges

None

Example

The following command starts the refresh operation:

```
=> SELECT START_REFRESH();
             start_refresh
----------------------------------------
 Starting refresh background process.
```

See Also

CLEAR_PROJECTION_REFRESHES (page 455)

MARK_DESIGN_KSAFE (page 510)

PROJECTION_REFRESHES (page 1056)

PROJECTIONS (page 961)

Clearing PROJECTION_REFRESHES History in the Administrator's Guide

Purge Functions

This section contains purge functions specific to HP Vertica.

PURGE

Purges all projections in the physical schema. Permanently removes deleted data from physical storage so that the disk space can be reused. You can purge historical data up to and including the epoch in which the Ancient History Mark is contained.

Syntax

```
PURGE()
```

Privileges

- Table owner
- USAGE privilege on schema

Note

- PURGE() was formerly named PURGE_ALL_PROJECTIONS. HP Vertica supports both function calls.

Caution: PURGE could temporarily take up significant disk space while the data is being purged.

See Also

MERGE_PARTITIONS (page 513)

PARTITION_TABLE (page 516)

PURGE_PROJECTION (page 520)

PURGE_TABLE (page 520)

STORAGE_CONTAINERS (page 1098)

Purging Deleted Data in the Administrator's Guide

PURGE_PARTITION

Purges a table partition of deleted rows. Similar to PURGE and PURGE_PROJECTION, this function removes deleted data from physical storage so that the disk space can be reused. It only removes data from the AHM epoch and earlier.

Syntax

```
PURGE_PARTITION ( '[[db_name.]schema_name.]table_name', partition_key )
```

Parameters

`[[db_name.]schema_name.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	The name of the partitioned table
partition_key	The key of the partition to be purged of deleted rows

Privileges

- Table owner
- USAGE privilege on schema

Example

The following example lists the count of deleted rows for each partition in a table, then calls PURGE_PARTITION to purge the deleted rows from the data.

```
=> SELECT partition_key,table_schema,projection_name,sum(deleted_row_count) AS deleted_row_count
FROM partitions
-> GROUP BY partition_key,table_schema,projection_name  ORDER BY partition_key;

 partition_key | table_schema | projection_name | deleted_row_count
---------------+--------------+-----------------+-------------------
 0             | public       | t_super         |                 2
 1             | public       | t_super         |                 2
 2             | public       | t_super         |                 2
 3             | public       | t_super         |                 2
 4             | public       | t_super         |                 2
 5             | public       | t_super         |                 2
 6             | public       | t_super         |                 2
 7             | public       | t_super         |                 2
 8             | public       | t_super         |                 2
 9             | public       | t_super         |                 1
(10 rows)


=> SELECT PURGE_PARTITION('t',5); -- Purge partition with key 5.
```

```
                          purge_partition
-------------------------------------------------------------------------
 Task: merge partitions
(Table: public.t) (Projection: public.t_super)

(1 row)


=> SELECT partition_key,table_schema,projection_name,sum(deleted_row_count) AS deleted_row_count
FROM partitions
-> GROUP BY partition_key,table_schema,projection_name ORDER BY partition_key;

 partition_key | table_schema | projection_name | deleted_row_count
---------------+--------------+-----------------+-------------------
 0             | public       | t_super         |                 2
 1             | public       | t_super         |                 2
 2             | public       | t_super         |                 2
 3             | public       | t_super         |                 2
 4             | public       | t_super         |                 2
 5             | public       | t_super         |                 0
 6             | public       | t_super         |                 2
 7             | public       | t_super         |                 2
 8             | public       | t_super         |                 2
 9             | public       | t_super         |                 1
(10 rows)
```

See Also

PURGE (page 517)

PURGE_PROJECTION (on page 520)

PURGE_TABLE (page 520)

MERGE_PARTITIONS (page 513)

STORAGE_CONTAINERS (page 1098)

PURGE_PROJECTION

Purges the specified projection. Permanently removes deleted data from physical storage so that the disk space can be reused. You can purge historical data up to and including the epoch in which the Ancient History Mark is contained.

Caution: PURGE_PROJECTION could temporarily take up significant disk space while purging the data.

Syntax

```
PURGE_PROJECTION ( '[[db-name.]schema.]projection_name' )
```

Parameters

`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database

	objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection_name	Identifies the projection name. When using more than one schema, specify the schema that contains the projection, as noted above.

Privileges

- Table owner
- USAGE privilege on schema

Notes

See ***PURGE*** (page 517) for notes about the outcome of purge operations.

See Also

MERGE_PARTITIONS (page 513)

PURGE_TABLE (page 520)

STORAGE_CONTAINERS (page 1098)

Purging Deleted Data in the Administrator's Guide

PURGE_TABLE

Note: This function was formerly named PURGE_TABLE_PROJECTIONS(). HP Vertica still supports the former function name.

Purges all projections of the specified table. You cannot use this function to purge temporary tables. Permanently removes deleted data from physical storage so that the disk space can be reused. You can purge historical data up to and including the epoch in which the Ancient History Mark is contained.

Syntax

```
PURGE_TABLE ( '[[db-name.]schema.]table_name' )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	Specifies the table to purge.

Privileges

- Table owner
- USAGE privilege on schema

Caution: PURGE_TABLE could temporarily take up significant disk space while the data is being purged.

Example

The following example purges all projections for the store sales fact table located in the Vmart schema:

```
=> SELECT PURGE_TABLE('store.store_sales_fact');
```

See Also

PURGE (page 517) for notes about the outcome of purge operations.

MERGE_PARTITIONS (page 513)

PURGE_TABLE (page 520)

STORAGE_CONTAINERS (page 1098)

Purging Deleted Data in the Administrator's Guide

Session Management Functions

This section contains session management functions specific to HP Vertica.

See also the SQL system table ***V_MONITOR.SESSIONS*** (page 1095)

CANCEL_REFRESH

Cancels refresh-related internal operations initiated by START_REFRESH().

Syntax

```
CANCEL_REFRESH()
```

Privileges

None

Notes

- Refresh tasks run in a background thread in an internal session, so you cannot use ***INTERRUPT_STATEMENT*** (page 503) to cancel those statements. Instead, use CANCEL_REFRESH to cancel statements that are run by refresh-related internal sessions.
- Run CANCEL_REFRESH() on the same node on which START_REFRESH() was initiated.
- CANCEL_REFRESH() cancels the refresh operation running on a node, waits for the cancelation to complete, and returns SUCCESS.

- Only one set of refresh operations runs on a node at any time.

See Also

INTERRUPT_STATEMENT (page 503)

SESSIONS (page 1095)

START_REFRESH (page 538)

PROJECTION_REFRESHES (page 1056)

CLOSE_ALL_SESSIONS

Closes all external sessions except the one issuing the CLOSE_ALL_SESSIONS functions.

Syntax

```
CLOSE_ALL_SESSIONS()
```

Privileges

None; however, a non-superuser can only close his or her own session.

Notes

Closing of the sessions is processed asynchronously. It might take some time for the session to be closed. Check the ***SESSIONS*** (page 1095) table for the status.

Database shutdown is prevented if new sessions connect after the CLOSE_SESSION or CLOSE_ALL_SESSIONS() command is invoked (and before the database is actually shut down). See **Controlling Sessions** below.

Message

```
close_all_sessions | Close all sessions command sent.
Check SESSIONS for progress.
```

Examples

Two user sessions opened, each on a different node:

```
vmartdb=> SELECT * FROM sessions;
-[ RECORD 1
]--------------+-----------------------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
client_pid                 | 4554
login_timestamp            | 2011-01-03 14:05:40.252625-05
session_id                 | stress04-4325:0x14
client_label               |
transaction_start          | 2011-01-03 14:05:44.325781
transaction_id             | 45035996273728326
transaction_description    | user dbadmin (select * from sessions;)
statement_start            | 2011-01-03 15:36:13.896288
statement_id               | 10
```

```
last_statement_duration_us | 14978
current_statement          | select * from sessions;
ssl_state                  | None
authentication_method      | Trust
-[ RECORD 2
]--------------+--------------------------------------------------
node_name                  | v_vmartdb_node0002
user_name                  | dbadmin
client_hostname            | 127.0.0.1:57174
client_pid                 | 30117
login_timestamp            | 2011-01-03 15:33:00.842021-05
session_id                 | stress05-27944:0xc1a
client_label               |
transaction_start          | 2011-01-03 15:34:46.538102
transaction_id             | -1
transaction_description    | user dbadmin (COPY Mart_Fact FROM
'/data/mart_Fact.tbl'
                             DELIMITER '|' NULL '\\n';)
statement_start            | 2011-01-03 15:34:46.538862
statement_id               |
last_statement_duration_us | 26250
current_statement          | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER
'|'
                             NULL '\\n';
ssl_state                  | None
authentication_method      | Trust
-[ RECORD 3
]--------------+--------------------------------------------------
node_name                  | v_vmartdb_node0003
user_name                  | dbadmin
client_hostname            | 127.0.0.1:56367
client_pid                 | 1191
login_timestamp            | 2011-01-03 15:31:44.939302-05
session_id                 | stress06-25663:0xbec
client_label               |
transaction_start          | 2011-01-03 15:34:51.05939
transaction_id             | 54043195528458775
transaction_description    | user dbadmin (COPY Mart_Fact FROM
'/data/Mart_Fact.tbl'
                             DELIMITER '|' NULL '\\n' DIRECT;)
statement_start            | 2011-01-03 15:35:46.436748
statement_id               |
last_statement_duration_us | 1591403
current_statement          | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER
'|'
                             NULL '\\n' DIRECT;
ssl_state                  | None
authentication_method      | Trust
```

Close all sessions:

```
vmartdb=> \x
Expanded display is off.
vmartdb=> SELECT CLOSE_ALL_SESSIONS();
```

```
                            CLOSE_ALL_SESSIONS
-------------------------------------------------------------------------
 Close all sessions command sent. Check v_monitor.sessions for progress.
(1 row)
```

Sessions contents after issuing the CLOSE_ALL_SESSIONS() command:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1 ]--------------+---------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
client_pid                 | 4554
login_timestamp            | 2011-01-03 14:05:40.252625-05
session_id                 | stress04-4325:0x14
client_label               |
transaction_start          | 2011-01-03 14:05:44.325781
transaction_id             | 45035996273728326
transaction_description    | user dbadmin (SELECT * FROM sessions;)
statement_start            | 2011-01-03 16:19:56.720071
statement_id               | 25
last_statement_duration_us | 15605
current_statement          | SELECT * FROM SESSIONS;
ssl_state                  | None
authentication_method      | Trust
```

Controlling Sessions

The database administrator must be able to disallow new incoming connections in order to shut down the database. On a busy system, database shutdown is prevented if new sessions connect after the CLOSE_SESSION or CLOSE_ALL_SESSIONS() command is invoked — and before the database actually shuts down.

One option is for the administrator to issue the `SHUTDOWN('true')` command, which forces the database to shut down and disallow new connections. See ***SHUTDOWN*** (page 535) in the SQL Reference Manual.

Another option is to modify the `MaxClientSessions` parameter from its original value to 0, in order to prevent new non-dbadmin users from connecting to the database.

1. Determine the original value for the `MaxClientSessions` parameter by querying the V_MONITOR.***CONFIGURATIONS_PARAMETERS*** (page 996) system table:

   ```
   => SELECT CURRENT_VALUE FROM CONFIGURATION_PARAMETERS WHERE
   parameter_name='MaxClientSessions';
    CURRENT_VALUE
   ---------------
    50
   (1 row)
   ```

2. Set the `MaxClientSessions` parameter to 0 to prevent new non-dbadmin connections:

   ```
   => SELECT SET_CONFIG_PARAMETER('MaxClientSessions', 0);
   ```

   Note: The previous command allows up to five administrators to log in.

3. Issue the `CLOSE_ALL_SESSIONS()` command to remove existing sessions:

```
=> SELECT CLOSE_ALL_SESSIONS();
```

4 Query the SESSIONS table:

```
=> SELECT * FROM SESSIONS;
```

When the session no longer appears in the SESSIONS table, disconnect and run the Stop Database command.

5 Restart the database.

6 Restore the MaxClientSessions parameter to its original value:

```
=> SELECT SET_CONFIG_PARAMETER('MaxClientSessions', 50);
```

See Also

CLOSE_SESSION (page 458)

CONFIGURATION_PARAMETERS (page 996)

SHUTDOWN (page 535)

V_MONITOR.SESSIONS (page 1095)

Managing Sessions and Configuration Parameters in the Administrator's Guide

CLOSE_SESSION

Interrupts the specified external session, rolls back the current transaction, if any, and closes the socket.

Syntax

```
CLOSE_SESSION ( 'sessionid' )
```

Parameters

sessionid	A string that specifies the session to close. This identifier is unique within the cluster at any point in time but can be reused when the session closes.

Privileges

None; however, a non-superuser can only close his or her own session.

Notes

- Closing of the session is processed asynchronously. It could take some time for the session to be closed. Check the ***SESSIONS*** (page 1095) table for the status.
- Database shutdown is prevented if new sessions connect after the CLOSE_SESSION() command is invoked (and before the database is actually shut down. See **Controlling Sessions** below.

Messages

The following are the messages you could encounter:

- For a badly formatted sessionID

```
close_session | Session close command sent. Check SESSIONS for progress.
Error: invalid Session ID format
```

- For an incorrect sessionID parameter

```
Error: Invalid session ID or statement key
```

Examples

User session opened. RECORD 2 shows the user session running COPY DIRECT statement.

```
vmartdb=> SELECT * FROM sessions;
-[ RECORD 1 ]--------------+-----------------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
client_pid                 | 4554
login_timestamp            | 2011-01-03 14:05:40.252625-05
session_id                 | stress04-4325:0x14
client_label               |
transaction_start          | 2011-01-03 14:05:44.325781
transaction_id             | 45035996273728326
transaction_description    | user dbadmin (SELECT * FROM sessions;)
statement_start            | 2011-01-03 15:36:13.896288
statement_id               | 10
last_statement_duration_us | 14978
current_statement          | select * from sessions;
ssl_state                  | None
authentication_method      | Trust
-[ RECORD 2 ]--------------+-----------------------------------------------
node_name                  | v_vmartdb_node0002
user_name                  | dbadmin
client_hostname            | 127.0.0.1:57174
client_pid                 | 30117
login_timestamp            | 2011-01-03 15:33:00.842021-05
session_id                 | stress05-27944:0xc1a
client_label               |
transaction_start          | 2011-01-03 15:34:46.538102
transaction_id             | -1
transaction_description    | user dbadmin (COPY ClickStream_Fact FROM
                           |  '/data/clickstream/1g/ClickStream_Fact.tbl'
                           |  DELIMITER '|' NULL '\\n' DIRECT;)
statement_start            | 2011-01-03 15:34:46.538862
statement_id               |
last_statement_duration_us | 26250
current_statement          | COPY ClickStream_Fact FROM '/data/clickstream
                           |  /1g/ClickStream_Fact.tbl' DELIMITER '|' NULL
                           |  '\\n' DIRECT;
ssl_state                  | None
authentication_method      | Trust
```

Close user session stress05-27944:0xc1a

```
vmartdb=> \x
Expanded display is off.
vmartdb=> SELECT CLOSE_SESSION('stress05-27944:0xc1a');
```

```
                          CLOSE_SESSION
-----------------------------------------------------------------
 Session close command sent. Check v_monitor.sessions for progress.
(1 row)
```

Query the sessions table again for current status, and you can see that the second session has been closed:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1 ]--------------+-----------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
client_pid                 | 4554
login_timestamp            | 2011-01-03 14:05:40.252625-05
session_id                 | stress04-4325:0x14
client_label               |
transaction_start          | 2011-01-03 14:05:44.325781
transaction_id             | 45035996273728326
transaction_description    | user dbadmin (select * from SESSIONS;)
statement_start            | 2011-01-03 16:12:07.841298
statement_id               | 20
last_statement_duration_us | 2099
current_statement          | SELECT * FROM SESSIONS;
ssl_state                  | None
authentication_method      | Trust
```

Controlling Sessions

The database administrator must be able to disallow new incoming connections in order to shut down the database. On a busy system, database shutdown is prevented if new sessions connect after the CLOSE_SESSION or CLOSE_ALL_SESSIONS() command is invoked — and before the database actually shuts down.

One option is for the administrator to issue the `SHUTDOWN('true')` command, which forces the database to shut down and disallow new connections. See ***SHUTDOWN*** (page 535) in the SQL Reference Manual.

Another option is to modify the `MaxClientSessions` parameter from its original value to 0, in order to prevent new non-dbadmin users from connecting to the database.

1. Determine the original value for the `MaxClientSessions` parameter by querying the V_MONITOR.***CONFIGURATIONS_PARAMETERS*** (page 996) system table:

   ```
   => SELECT CURRENT_VALUE FROM CONFIGURATION_PARAMETERS WHERE
   parameter_name='MaxClientSessions';
    CURRENT_VALUE
   ---------------
    50
   (1 row)
   ```

2. Set the `MaxClientSessions` parameter to 0 to prevent new non-dbadmin connections:

   ```
   => SELECT SET_CONFIG_PARAMETER('MaxClientSessions', 0);
   ```

Note: The previous command allows up to five administrators to log in.

3 Issue the CLOSE_ALL_SESSIONS() command to remove existing sessions:

```
=> SELECT CLOSE_ALL_SESSIONS();
```

4 Query the SESSIONS table:

```
=> SELECT * FROM SESSIONS;
```

When the session no longer appears in the SESSIONS table, disconnect and run the Stop Database command.

5 Restart the database.

6 Restore the MaxClientSessions parameter to its original value:

```
=> SELECT SET_CONFIG_PARAMETER('MaxClientSessions', 50);
```

See Also

CLOSE_ALL_SESSIONS (page 461), ***CONFIGURATION_PARAMETERS*** (page 996), ***SESSIONS*** (page 1095), ***SHUTDOWN*** (page 535)

Managing Sessions and Configuration Parameters in the Administrator's Guide

GET_NUM_ACCEPTED_ROWS

Returns the number of rows loaded into the database for the last completed load for the current session. GET_NUM_ACCEPTED_ROWS is a meta-function. Do not use it as a value in an INSERT query.

The number of accepted rows is not available for a load that is currently in process. Check the ***LOAD_STREAMS*** (page 1031) system table for its status.

Also, this meta-function supports only loads from STDIN or a single file on the initiator. You cannot use GET_NUM_ACCEPTED_ROWS for multi-node loads.

Syntax

```
GET_NUM_ACCEPTED_ROWS();
```

Privileges

None

NOTE: The data regarding accepted rows from the last load during the current session does not persist, and is lost when you initiate a new load.

See Also

GET_NUM_REJECTED_ROWS (page 498)

GET_NUM_REJECTED_ROWS

Returns the number of rows that were rejected during the last completed load for the current session. GET_NUM_REJECTED_ROWS is a meta-function. Do not use it as a value in an INSERT query.

Rejected row information is unavailable for a load that is currently running. The number of rejected rows is not available for a load that is currently in process. Check the ***LOAD_STREAMS*** (page 1031) system table for its status.

Also, this meta-function supports only loads from STDIN or a single file on the initiator. You cannot use GET_NUM_REJECTED_ROWS for multi-node loads.

Syntax

```
GET_NUM_REJECTED_ROWS();
```

Privileges

None

Note: The data regarding rejected rows from the last load during the current session does not persist, and is dropped when you initiate a new load.

See Also

GET_NUM_ACCEPTED_ROWS (page 497)

INTERRUPT_STATEMENT

Interrupts the specified statement (within an external session), rolls back the current transaction, and writes a success or failure message to the log file.

Syntax

```
INTERRUPT_STATEMENT( 'session_id ', statement_id )
```

Parameters

session_id	Specifies the session to interrupt. This identifier is unique within the cluster at any point in time.
statement_id	Specifies the statement to interrupt

Privileges

Must be a superuser

Notes

- Only statements run by external sessions can be interrupted.
- Sessions can be interrupted during statement execution.

- If the *statement_id* is valid, the statement is interruptible. The command is successfully sent and returns a success message. Otherwise the system returns an error.

Messages

The following list describes messages you might encounter:

Message	Meaning
`Statement interrupt sent. Check SESSIONS for progress.`	This message indicates success.
`Session <id> could not be successfully interrupted: session not found.`	The session ID argument to the interrupt command does not match a running session.
`Session <id> could not be successfully interrupted: statement not found.`	The statement ID does not match (or no longer matches) the ID of a running statement (if any).
`No interruptible statement running`	The statement is DDL or otherwise non-interruptible.
`Internal (system) sessions cannot be interrupted.`	The session is internal, and only statements run by external sessions can be interrupted.

Examples

Two user sessions are open. RECORD 1 shows user session running `SELECT FROM SESSION`, and RECORD 2 shows user session running `COPY DIRECT`:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1
]--------------+-----------------------------------------------------
node_name                   | v_vmartdb_node0001
user_name                   | dbadmin
client_hostname             | 127.0.0.1:52110
client_pid                  | 4554
login_timestamp             | 2011-01-03 14:05:40.252625-05
session_id                  | stress04-4325:0x14
client_label                |
transaction_start           | 2011-01-03 14:05:44.325781
transaction_id              | 45035996273728326
transaction_description     | user dbadmin (select * from sessions;)
statement_start             | 2011-01-03 15:36:13.896288
statement_id                | 10
last_statement_duration_us | 14978
current_statement           | select * from sessions;
ssl_state                   | None
authentication_method       | Trust
-[ RECORD 2
]--------------+-----------------------------------------------------
node_name                   | v_vmartdb_node0003
user_name                   | dbadmin
```

```
client_hostname            | 127.0.0.1:56367
client_pid                 | 1191
login_timestamp            | 2011-01-03 15:31:44.939302-05
session_id                 | stress06-25663:0xbec
client_label               |
transaction_start          | 2011-01-03 15:34:51.05939
transaction_id             | 54043195528458775
transaction_description    | user dbadmin (COPY Mart_Fact FROM
'/data/Mart_Fact.tbl'
                             DELIMITER '|' NULL '\\n' DIRECT;)
statement_start            | 2011-01-03 15:35:46.436748
statement_id               | 5
last_statement_duration_us | 1591403
current_statement          | COPY Mart_Fact FROM '/data/Mart_Fact.tbl' DELIMITER
'|'
                             NULL '\\n' DIRECT;
ssl_state                  | None
authentication_method      | Trust
```

Interrupt the COPY DIRECT statement running in stress06-25663:0xbec:

```
=> \x
Expanded display is off.
=> SELECT INTERRUPT_STATEMENT('stress06-25663:0x1537', 5);
                       interrupt_statement
-----------------------------------------------------------------
 Statement interrupt sent. Check v_monitor.sessions for progress.
(1 row)
```

Verify that the interrupted statement is no longer active by looking at the current_statement column in the SESSIONS system table. This column becomes blank when the statement has been interrupted:

```
=> SELECT * FROM SESSIONS;
-[ RECORD 1
]--------------+-----------------------------------------------------
node_name                  | v_vmartdb_node0001
user_name                  | dbadmin
client_hostname            | 127.0.0.1:52110
client_pid                 | 4554
login_timestamp            | 2011-01-03 14:05:40.252625-05
session_id                 | stress04-4325:0x14
client_label               |
transaction_start          | 2011-01-03 14:05:44.325781
transaction_id             | 45035996273728326
transaction_description    | user dbadmin (select * from sessions;)
statement_start            | 2011-01-03 15:36:13.896288
statement_id               | 10
last_statement_duration_us | 14978
current_statement          | select * from sessions;
ssl_state                  | None
authentication_method      | Trust
-[ RECORD 2
]--------------+-----------------------------------------------------
node_name                  | v_vmartdb_node0003
```

```
user_name                   | dbadmin
client_hostname             | 127.0.0.1:56367
client_pid                  | 1191
login_timestamp             | 2011-01-03 15:31:44.939302-05
session_id                  | stress06-25663:0xbec
client_label                |
transaction_start           | 2011-01-03 15:34:51.05939
transaction_id              | 54043195528458775
transaction_description     | user dbadmin (COPY Mart_Fact FROM
'/data/Mart_Fact.tbl'
                              DELIMITER '|' NULL '\\n' DIRECT;)
statement_start             | 2011-01-03 15:35:46.436748
statement_id                | 5
last_statement_duration_us  | 1591403
current_statement           |
ssl_state                   | None
authentication_method       | Trust
```

See Also

SESSIONS (page 1095)

Managing Sessions and Configuration Parameters in the Administrator's Guide

Statistic Management Functions

This section contains statistic management functions specific to HP Vertica.

ANALYZE_HISTOGRAM

Collects and aggregates data samples and storage information from all nodes that store projections associated with the specified table or column.

If the function returns successfully (0), HP Vertica writes the returned statistics to the catalog. The query optimizer uses this collected data to recommend the best possible plan to execute a query. Without analyzing table statistics, the query optimizer would assume uniform distribution of data values and equal storage usage for all projections.

ANALYZE_HISTOGRAM is a DDL operation that auto-commits the current transaction, if any. The ANALYZE_HISTOGRAM function reads a variable amount of disk contents to aggregate sample data for statistical analysis. Use the function's *percent* float parameter to specify the total disk space from which HP Vertica collects sample data. The ***ANALYZE_STATISTICS*** (page 440) function returns similar data, but uses a fixed disk space amount (10 percent). Analyzing more than 10 percent disk space takes proportionally longer to process, but produces a higher level of sampling accuracy. ANALYZE_HISTOGRAM is supported on local temporary tables, but not on global temporary tables.

Syntax

```
ANALYZE_HISTOGRAM ('')
... | ( '[ [ db-name.]schema.]table [.column-name ]' [, percent ] )
```

Return value

0 - For success. If an error occurs, refer to `vertica.log` for details.

Parameters

`''`	Empty string. Collects statistics for all tables.
`[[`*db-name*`.]`*schema*`.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the name of the table and collects statistics for all projections of that table. If you are using more than one schema, specify the schema that contains the projection, as noted in the [[*db-name*.]*schema*.] entry.
`[.`*column-name*`]`	[Optional] Specifies the name of a single column, typically a predicate column. Using this option with a table specification lets you collect statistics for only that column. **Note:** If you alter a table to add or drop a column, or add a new column to a table and populate its contents with either default or other values, HP Vertica recommends calling this function on the new table column to get the most current statistics.
percent	[Optional] Specifies what percentage of data to read from disk (not the amount of data to analyze). Specify a float from 1 – 100, such as 33.3. By default, the function reads 10% of the table data from disk. For more information, see Collecting Statistics in the Administrator's Guide.

Privileges

- Any INSERT/UPDATE/DELETE privilege on table
- USAGE privilege on schema that contains the table

Use the HP Vertica statistics functions as follows:

Use this function...	To obtain...
ANALYZE_STATISTICS (page 440)	A fixed-size statistical data sampling (10 percent per disk). This function returns results quickly, but is less

	accurate than using ANALYZE_HISTOGRAM to get a larger sampling of disk data.
ANALYZE_HISTOGRAM (page 437)	A specified percentage of disk data sampling (from 1 - 100). If you analyze more than 10 percent data per disk, this function is more accurate than ANALYZE_STATISTICS, but requires proportionately longer to return statistics.

Analyzing Results

To retrieve hints about under-performing queries and the associated root causes, use the ***ANALYZE_WORKLOAD*** (page 443) function. This function runs the Workload Analyzer and returns tuning recommendations, such as "`run analyze_statistics on schema.table.column`". You or your database administrator should act upon the tuning recommendations.

You can also find database tuning recommendations on the Management Console.

Canceling ANALYZE_HISTOGRAM

You can cancel this function mid-analysis by issuing CTRL-C in a vsql shell or by invoking the ***INTERRUPT_STATEMENT()*** (page 503) function.

Notes

By default, HP Vertica analyzes more than one column (subject to resource limits) in a single-query execution plan to:

- Reduce plan execution latency
- Help speed up analysis of relatively small tables that have a large number of columns

Examples

In this example, the ANALYZE_STATISTICS() function reads 10 percent of the disk data. This is the static default value for this function. The function returns 0 for success:

```
=> SELECT ANALYZE_STATISTICS('shipping_dimension.shipping_key');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

This example uses ANALYZE_HISTOGRAM () without specifying a percentage value. Since this function has a default value of 10 percent, it returns the identical data as the ANALYZE_STATISTICS() function, and returns 0 for success:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key');
 ANALYZE_HISTOGRAM
-------------------
                 0
(1 row)
```

This example uses ANALYZE_HISTOGRAM (), specifying its percent parameter as 100, indicating it will read the entire disk to gather data. After the function performs a full column scan, it returns 0 for success:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key', 100);
 ANALYZE_HISTOGRAM
-------------------
                 0
(1 row)
```

In this command, only 0.1% (1/1000) of the disk is read:

```
=> SELECT ANALYZE_HISTOGRAM('shipping_dimension.shipping_key', 0.1);
 ANALYZE_HISTOGRAM
-------------------
                 0
(1 row)
```

See Also

ANALYZE_STATISTICS (page 440)

ANALYZE_WORKLOAD (page 443)

DROP_STATISTICS (page 476)

EXPORT_STATISTICS (page 490)

IMPORT_STATISTICS (page 502)

INTERRUPT_STATEMENT (page 503)

Collecting Statistics in the Administrator's Guide

ANALYZE_STATISTICS

Collects and aggregates data samples and storage information from all nodes that store projections associated with the specified table or column.

If the function returns successfully (0), HP Vertica writes the returned statistics to the catalog. The query optimizer uses this collected data to recommend the best possible plan to execute a query. Without analyzing table statistics, the query optimizer would assume uniform distribution of data values and equal storage usage for all projections.

ANALYZE_STATISTICS is a DDL operation that auto-commits the current transaction, if any. The ANALYZE_STATISTICS function reads a fixed, 10 percent of disk contents to aggregate sample data for statistical analysis. To obtain a larger (or smaller) data sampling, use the ***ANALYZE_HISTOGRAM*** (page 437) function, which lets you specify the percent of disk to read. Analyzing more that 10 percent disk space takes proportionally longer to process, but results in a higher level of sampling accuracy. ***ANALYZE_STATISTICS*** (page 440) is supported on local temporary tables, but not on global temporary tables.

Syntax

```
ANALYZE_STATISTICS [ ('')
... | ( '[ [ db-name.]schema.]table [.column-name ]' ) ]
```

Return Value

0 - For success.

If an error occurs, refer to `vertica.log` for details.

Parameters

`''`	Empty string. Collects statistics for all tables.
`[[`*db-name*`.]`*schema*`.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the name of the table and collects statistics for all projections of that table. **Note:** If you are using more than one schema, specify the schema that contains the projection, as noted as noted in the [[*db-name*.]*schema*.] entry.
`[.`*column-name*`]`	[Optional] Specifies the name of a single column, typically a predicate column. Using this option with a table specification lets you collect statistics for only that column. **Note:** If you alter a table to add or drop a column, or add a new column to a table and populate its contents with either default or other values, HP Vertica recommends calling this function on the new table column to get the most current statistics.

Privileges

- Any INSERT/UPDATE/DELETE privilege on table
- USAGE privilege on schema that contains the table

Use the HP Vertica statistics functions as follows:

Use this function...	To obtain...
ANALYZE_STATISTICS (page 440)	A fixed-size statistical data sampling (10 percent per disk). This function returns results quickly, but is less accurate than using ANALYZE_HISTOGRAM to get a larger sampling of disk data.
ANALYZE_HISTOGRA	A specified percentage of disk data sampling (from 1 -

M (page 437)	100). If you analyze more than 10 percent data per disk, this function is more accurate than ANALYZE_STATISTICS, but requires proportionately longer to return statistics.

Analyzing results

To retrieve hints about under-performing queries and the associated root causes, use the ***ANALYZE_WORKLOAD*** (page 443) function. This function runs the Workload Analyzer and returns tuning recommendations, such as "`run analyze_statistics on schema.table.column`". You or your database administrator should act upon the tuning recommendations.

You can also find database tuning recommendations on the Management Console.

Canceling this function

You can cancel statistics analysis by issuing CTRL-C in a vsql shell or by invoking the ***INTERRUPT_STATEMENT()*** (page 503) function.

Notes

- Always run ANALYZE_STATISTICS on a table or column rather than a projection.
- By default, HP Vertica analyzes more than one column (subject to resource limits) in a single-query execution plan to:
  - reduce plan execution latency
  - help speed up analysis of relatively small tables that have a large number of columns
- Pre-join projection statistics are updated on any pre-joined tables.

Examples

Computes statistics on all projections in the Vmart database and returns 0 (success):

```
=> SELECT ANALYZE_STATISTICS ('');
 analyze_statistics
--------------------
                  0
(1 row)
```

Computes statistics on a single table (`shipping_dimension`) and returns 0 (success):

```
=> SELECT ANALYZE_STATISTICS ('shipping_dimension');
 analyze_statistics
--------------------
                  0
(1 row)
```

Computes statistics on a single column (`shipping_key`) across all projections for the `shipping_dimension` table and returns 0 (success):

```
=> SELECT ANALYZE_STATISTICS('shipping_dimension.shipping_key');
 analyze_statistics
--------------------
                  0
(1 row)
```

For use cases, see Collecting Statistics in the Administrator's Guide

See Also

ANALYZE_HISTOGRAM (page 437)

ANALYZE_WORKLOAD (page 443)

DROP_STATISTICS (page 476)

EXPORT_STATISTICS (page 490)

IMPORT_STATISTICS (page 502)

INTERRUPT_STATEMENT (page 503)

DROP_STATISTICS

Removes statistics for the specified table and lets you optionally specify the category of statistics to drop.

Syntax

```
DROP_STATISTICS { ('') | ('[[db-name.]schema-name.]table' [, {'BASE' |
'HISTOGRAMS' | 'ALL'} ])};
```

Return Value

0 - If successful, DROP_STATISTICS always returns 0. If the command fails, DROP_STATISTICS displays an error message. See `vertica.log` for message details.

Parameters

`''`	Empty string. Drops statistics for all projections.
`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
`table`	Drops statistics for all projections within the specified table. When using more than one schema, specify the schema that contains the table with the projections you want to delete, as noted in the syntax.

CATEGORY	Specifies the category of statistics to drop for the named [db-name.]schema-name.]table: ▪ 'BASE' (default) drops histograms and row counts (min/max column values, histogram. ▪ 'HISTOGRAMS' drops only the histograms. Row counts statistics remain. ▪ 'ALL' drops all statistics.

Privileges

- INSERT/UPDATE/DELETE privilege on table
- USAGE privilege on schema that contains the table

Notes

Once dropped, statistics can be time consuming to regenerate.

Example

The following command analyzes all statistics on the VMart schema database:

```
=> SELECT ANALYZE_STATISTICS('');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)
```

This command drops base statistics for table store_sales_fact in the store schema:

```
=> SELECT DROP_STATISTICS('store.store_sales_fact', 'BASE');
 drop_statistics
-----------------
               0
(1 row)
```

Note that this command works the same as the previous command:

```
=> SELECT DROP_STATISTICS('store.store_sales_fact');
 DROP_STATISTICS
-----------------
               0
(1 row)
```

This command also drops statistics for all table projections:

```
=> SELECT DROP_STATISTICS ('');
 DROP_STATISTICS
-----------------
               0
(1 row)
```

For use cases, see Collecting Statistics in the Administrator's Guide

See Also

ANALYZE_STATISTICS (page 440)

EXPORT_STATISTICS (page 490)

IMPORT_STATISTICS (page 502)

EXPORT_STATISTICS

Generates an XML file that contains statistics for the database. You can optionally export statistics on a single database object (table, projection, or table column).

Before you export statistics for the database, run ***ANALYZE_STATISTICS()*** (page 440) to automatically collect the most up to date statistics information.

Note: Use the second argument only if statistics in the database do not match the statistics of data.

Syntax

```
EXPORT_STATISTICS
[ ( 'destination' )
... | ( '[ [ db-name.]schema.]table [.column-name ]' ) ]
```

Parameters

destination	Specifies the path and name of the XML output file. An empty string returns the script to the screen.
[[*db-name*.]*schema*.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the name of the table and exports statistics for all projections of that table. **Note:** If you are using more than one schema, specify the schema that contains the projection, as noted as noted in the [[*db-name*.]*schema*.] entry.
[.*column-name*]	[Optional] Specifies the name of a single column, typically a predicate column. Using this option with a table specification lets you export statistics for only that column.

Privileges

Must be a superuser

Examples

The following command exports statistics on the VMart example database to a file:

```
vmart=> SELECT EXPORT_STATISTICS('/vmart/statistics/vmart_stats.xml');
         export_statistics
-----------------------------------
 Statistics exported successfully

(1 row)
```

The next statement exports statistics on a single column (price) from a table called food:

```
=> SELECT EXPORT_STATISTICS('/vmart/statistics/price.xml', 'food.price');
```

See Also

ANALYZE_STATISTICS (page 440)

DROP_STATISTICS (page 476)

IMPORT_STATISTICS (page 502)

Collecting Statistics in the Administrator's Guide

IMPORT_STATISTICS

Imports statistics from the XML file generated by the EXPORT_STATISTICS command.

Syntax

```
IMPORT_STATISTICS ( 'destination' )
```

Parameters

destination	Specifies the path and name of the XML input file (which is the output of EXPORT_STATISTICS function).

Privileges

Must be a superuser

Notes

- Imported statistics override existing statistics for all projections on the specified table.
- For use cases, see Collecting Statistics in the Administrator's Guide

See Also

ANALYZE_STATISTICS (page 440)

DROP_STATISTICS (page 476)

EXPORT_STATISTICS (page 490)

Storage Management Functions

This section contains storage management functions specific to HP Vertica.

ADD_LOCATION

Adds a storage location to the cluster. Use this function to add a new location, optionally with a location label. You can also add a location specifically for user access, and then grant one or more users access to the location.

Syntax

```
ADD_LOCATION ( 'path' [, 'node' , 'usage', 'location_label' ] )
```

Parameters

path	[Required] Specifies where the storage location is mounted. Path must be an empty directory with write permissions for user, group, or all.
node	[Optional] Indicates the cluster node on which a storage location resides. If you omit this parameter, the function adds the location to only the initiator node. Specifying the *node* parameter as an empty string (`''`) adds a storage location to all cluster nodes in a single transaction. **NOTE:** If you specify a node, you must also add a usage parameter.
usage	[Optional] Specifies what the storage location will be used for: ▪ DATA: Stores only data files. Use this option for labeled storage locations. ▪ TEMP: Stores only temporary files, created during loads or queries. ▪ DATA,TEMP: Stores both types of files in the location. ▪ USER: Allows non-dbadmin users access to the storage location for data files (not temp files), once they are granted privileges. DO NOT create a storage location for later use in a storage policy. Storage locations with policies must be for DATA usage. Also, note that this keyword is orthogonal to DATA and TEMP, and does not specify a particular usage, other than being accessible to non-dbadmin users with assigned privileges. You cannot alter a storage location to or from USER usage. **NOTE:** You can use this parameter only in conjunction with the node option. If you omit the usage parameter, the default is DATA,TEMP.

location_label	[Optional] Specifies a location label as a string, for example, *SSD*. Labeling a storage location lets you use the location label to create storage policies and as part of a multi-tenanted storage scheme.

Privileges

Must be a superuser

Storage Location Subdirectories

You cannot create a storage location in a subdirectory of an existing location. For example, if you create a storage location at one location, you cannot add a second storage location in a subdirectory of the first:

```
dbt=> select add_location ('/myvertica/Test/KMM','','DATA','SSD');
               add_location
-----------------------------------------
 /myvertica/Test/KMM added.
(1 row)
dbt=> select add_location ('/myvertica/Test/KMM/SSD','','DATA','SSD');
ERROR 5615:  Location [/myvertica/Test/KMM/SSD] conflicts with existing location
[/myvertica/Test/KMM] on node v_node0001
ERROR 5615:  Location [/myvertica/Test/KMM/SSD] conflicts with existing location
[/myvertica/Test/KMM] on node v_node0002
ERROR 5615:  Location [/myvertica/Test/KMM/SSD] conflicts with existing location
[/myvertica/Test/KMM] on node v_node0003
```

Example

This example adds a location that stores data and temporary files on the initiator node:

```
=> SELECT ADD_LOCATION('/secondverticaStorageLocation/');
```

This example adds a location to store data on `v_vmartdb_node0004`:

```
=> SELECT ADD_LOCATION('/secondverticaStorageLocation/' , 'v_vmartdb_node0004' ,
'DATA');
```

This example adds a new `DATA` storage location with a label, *SSD*. The label identifies the location when you create storage policies. Specifying the *node* parameter as an empty string adds the storage location to all cluster nodes in a single transaction:

```
VMART=> select add_location ('home/dbadmin/SSD/schemas', '', 'DATA', 'SSD');
          add_location
--------------------------------
 home/dbadmin/SSD/schemas added.
(1 row)
```

See Also

Adding Storage Locations in the Administrator's Guide

ALTER_LOCATION_USE (page 429)

DROP_LOCATION (page 472)

RESTORE_LOCATION (page 525)

RETIRE_LOCATION (page 526)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

ALTER_LOCATION_USE

Alters the type of files that can be stored at the specified storage location.

Syntax

```
ALTER_LOCATION_USE ( 'path' , [ 'node' ] , 'usage' )
```

Parameters

path	Specifies where the storage location is mounted.
node	[Optional] The HP Vertica node with the storage location. Specifying the *node* parameter as an empty string (`''`) alters the location across all cluster nodes in a single transaction. If you omit this parameter, *node* defaults to the initiator.
usage	Is one of the following: ▪ DATA: The storage location stores only data files. This is the supported use for both a USER storage location, and a labeled storage location. ▪ TEMP: The location stores only temporary files that are created during loads or queries. ▪ DATA,TEMP: The location can store both types of files.

Privileges

Must be a superuser

USER Storage Location Restrictions

You cannot change a storage location from a USER usage type if you created the location that way, or to a USER type if you did not. You can change a USER storage location to specify DATA (storing TEMP files is not supported). However, doing so does not affect the primary objective of a USER storage location, to be accessible by non-dbadmin users with assigned privileges.

Monitoring Storage Locations

Disk storage information that the database uses on each node is available in the ***V_MONITOR.DISK_STORAGE*** (page 1014) system table.

Example

The following example alters the storage location across all cluster nodes to store only data:

```
=> SELECT ALTER_LOCATION_USE ('/thirdVerticaStorageLocation/' , '' , 'DATA');
```

See Also

Altering Storage Locations in the Administrator's Guide

ADD_LOCATION (page 426)

DROP_LOCATION (page 472)

RESTORE_LOCATION (page 525)

RETIRE_LOCATION (page 526)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

ALTER_LOCATION_LABEL

Alters the location label. Use this function to add, change, or remove a location label. You change a location label only if it is not currently in use as part of a storage policy.

You can use this function to remove a location label. However, you cannot remove a location label if the name being removed is used in a storage policy, *and* the location from which you are removing the label is the last available storage for its associated objects.

NOTE: If you label an existing storage location that already contains data, and then include the labeled location in one or more storage policies, existing data could be moved. If the ATM determines data stored on a labeled location does not comply with a storage policy, the ATM moves the data elsewhere.

Syntax

```
ALTER_LOCATION_LABEL ( 'path' , 'node' , 'location_label' )
```

Parameters

path	Specifies the path of the storage location.
node	The HP Vertica node for the storage location. If you enter node as an empty string (''), the function performs a cluster-wide label change to all nodes. Any node that is unavailable generates an error.
location_label	Specifies a storage label as a string, for instance *SSD*. You can change an existing label assigned to a storage location, or add a new label. Specifying an empty string (') removes an existing label.

Privileges

Must be a superuser

Example

The following example alters (or adds) the label `SSD` to the storage location at the given path on all cluster nodes:

```
VMART=> select alter_location_label('/home/dbadmin/SSD/tables','', 'SSD');
          alter_location_label
--------------------------------------
 /home/dbadmin/SSD/tables label changed.
(1 row)
```

See Also

Altering Location Labels in the Administrator's Guide

CLEAR_OBJECT_STORAGE_POLICY (page 457)

SET_OBJECT_STORAGE_POLICY (page 534)

CLEAR_OBJECT_STORAGE_POLICY

Removes an existing storage policy. The specified object will no longer use a default storage location. Any existing data stored currently at the labeled location in the object's storage policy is moved to default storage during the next TM moveout operation.

Syntax

```
CLEAR_OBJECT_STORAGE_POLICY ( 'object_name' , [', key_min, key_max '])
```

Parameters

object_name	Specifies the database object with a storage policy to clear.
key_min, key_max	Specifies the table partition key value ranges stored at the labeled location. These parameters are applicable only when *object_name* is a table.

Privileges

Must be a superuser

Example

This example clears the storage policy for the object `lineorder`:

```
release=> select clear_object_storage_policy('lineorder');
      clear_object_storage_policy
-----------------------------------
 Default storage policy cleared.
(1 row)
```

See Also

Clearing a Storage Policy in the Administrator's Guide

ALTER_LOCATION_LABEL (page 430)

SET_OBJECT_STORAGE_POLICY (page 534)

DROP_LOCATION

Removes the specified storage location.

Syntax

```
DROP_LOCATION ( 'path' , 'node' )
```

Parameters

path	Specifies where the storage location to drop is mounted.
node	Is the HP Vertica node where the location is available.

Privileges

Must be a superuser

Retiring or Dropping a Storage Location

Dropping a storage location is a permanent operation and cannot be undone. Therefore, HP recommends that you retire a storage location before dropping it. Retiring a storage location lets you verify that you do not need the storage before dropping it. Additionally, you can easily restore a retired storage location if you determine it is still in use.

Storage Locations with Temp and Data Files

Dropping storage locations is limited to storage locations that contain only temp files.

If you use a storage location to store data and then alter it to store only temp files, the location can still contain data files. HP Vertica does not let you drop a storage location containing data files. You can manually merge out the data files from the storage location, and then wait for the ATM to mergeout the data files automatically, or, you can drop partitions. Deleting data files does not work.

Example

The following example drops a storage location on node3 that was used to store temp files:

```
=> SELECT DROP_LOCATION('/secondHP VerticaStorageLocation/' , 'node3');
```

See Also

Dropping Storage Locations and Retiring Storage Locations in the Administrator's Guide

ADD_LOCATION (page 426)

ALTER_LOCATION_USE (page 429)

RESTORE_LOCATION (page 525)

RETIRE_LOCATION (page 526)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

MEASURE_LOCATION_PERFORMANCE

Measures disk performance for the location specified.

Syntax

```
MEASURE_LOCATION_PERFORMANCE ( 'path' , 'node' )
```

Parameters

path	Specifies where the storage location to measure is mounted.
node	Is the HP Vertica node where the location to be measured is available.

Privileges

Must be a superuser

Notes

- To get a list of all node names on your cluster, query the ***V_MONITOR.DISK_STORAGE*** (page 1014) system table:

```
=> SELECT node_name from DISK_STORAGE;
   node_name
------------------
 v_vmartdb_node0004
 v_vmartdb_node0004
 v_vmartdb_node0005
 v_vmartdb_node0005
 v_vmartdb_node0006
 v_vmartdb_node0006
(6 rows)
```

- If you intend to create a tiered disk architecture in which projections, columns, and partitions are stored on different disks based on predicted or measured access patterns, you need to measure storage location performance for each location in which data is stored. You do not need to measure storage location performance for temp data storage locations because temporary files are stored based on available space.
- The method of measuring storage location performance applies only to configured clusters. If you want to measure a disk before configuring a cluster see Measuring Location Performance.

- Storage location performance equates to the amount of time it takes to read and write 1MB of data from the disk. This time equates to:

  IO time = Time to read/write 1MB + Time to seek = 1/Throughput + 1/Latency

  Throughput is the average throughput of sequential reads/writes (units in MB per second)

  Latency is for random reads only in seeks (units in seeks per second)

  Note: The IO time of a faster storage location is less than a slower storage location.

Example

The following example measures the performance of a storage location on v_vmartdb_node0004:

```
=> SELECT MEASURE_LOCATION_PERFORMANCE('/secondVerticaStorageLocation/' ,
'v_vmartdb_node0004');

WARNING:  measure_location_performance can take a long time. Please check logs for
progress
           measure_location_performance
-------------------------------------------------
 Throughput : 122 MB/sec. Latency : 140 seeks/sec
```

See Also

ADD_LOCATION (page 426)

ALTER_LOCATION_USE (page 429)

RESTORE_LOCATION (page 525)

RETIRE_LOCATION (page 526)

Measuring Location Performance in the Administrator's Guide

RESTORE_LOCATION

Restores a storage location that was previously retired with ***RETIRE_LOCATION*** (page 526).

Syntax

```
RESTORE_LOCATION ( 'path'  , 'node' )
```

Parameters

path	Specifies where the retired storage location is mounted.
node	Is the HP Vertica node where the retired location is available.

Privileges

Must be a superuser

Effects of Restoring a Previously Retired Location

After restoring a storage location, HP Vertica re-ranks all of the cluster storage locations and uses the newly-restored location to process queries as determined by its rank.

Monitoring Storage Locations

Disk storage information that the database uses on each node is available in the ***V_MONITOR.DISK_STORAGE*** (page 1014) system table.

Example

The following example restores the retired storage location on node3:

```
=> SELECT RESTORE_LOCATION ('/thirdHP VerticaStorageLocation/' ,
'v_vmartdb_node0004');
```

See Also

Modifying Storage Locations in the Administrator's Guide

ADD_LOCATION (page 426)

ALTER_LOCATION_USE (page 429)

DROP_LOCATION (page 472)

RETIRE_LOCATION (page 526)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

RETIRE_LOCATION

Makes the specified storage location inactive.

Syntax

```
RETIRE_LOCATION ( 'path'  , 'node' )
```

Parameters

path	Specifies where the storage location to retire is mounted.
node	Is the HP Vertica node where the location is available.

Privileges

Must be a superuser

Effects of Retiring a Storage Location

When you use this function, HP Vertica checks that the location is not the only storage for data and temp files. At least one location must exist on each node to store data and temp files, though you can store both sorts of files in either the same location, or separate locations.

NOTE: You cannot retire a location if it is used in a storage policy, *and* is the last available storage for its associated objects.

When you retire a storage location:

- No new data is stored at the retired location, unless you first restore it with the ***RESTORE_LOCATION()*** (page 525) function.
- If the storage location being retired contains stored data, the data is not moved, so you cannot drop the storage location. Instead, HP Vertica removes the stored data through one or more mergeouts.
- If the storage location being retired was used only for temp files, you can drop the location. See Dropping Storage Locations in the Administrators Guide and the ***DROP_LOCATION()*** (page 472) function.

Monitoring Storage Locations

Disk storage information that the database uses on each node is available in the ***V_MONITOR.DISK_STORAGE*** (page 1014) system table.

Example

The following example retires a storage location:

```
=> SELECT RETIRE_LOCATION ('/secondVerticaStorageLocation/' ,
'v_vmartdb_node0004');
```

See Also

Retiring Storage Locations in the Administrator's Guide

ADD_LOCATION (page 426)

ALTER_LOCATION_USE (page 429)

DROP_LOCATION (page 472)

RESTORE_LOCATION (page 525)

GRANT (Storage Location) (page 839)

REVOKE (Storage Location) (page 861)

SET_LOCATION_PERFORMANCE

Sets disk performance for the location specified.

Syntax

```
SET_LOCATION_PERFORMANCE ( 'path' , 'node' , 'throughput' , 'average_latency' )
```

Parameters

path	Specifies where the storage location to set is mounted.
node	Is the HP Vertica node where the location to be set is

	available. If this parameter is omitted, *node* defaults to the initiator.
throughput	Specifies the throughput for the location, which must be 1 or more.
average_latency	Specifies the average latency for the location. The *average_latency* must be 1 or more.

Privileges

Must be a superuser

Notes

To obtain the throughput and average latency for the location, run the ***MEASURE_LOCATION_PERFORMANCE()*** (page 511) function before you attempt to set the location's performance.

Example

The following example sets the performance of a storage location on node2 to a throughput of 122 megabytes per second and a latency of 140 seeks per second.

```
=> SELECT SET_LOCATION_PERFORMANCE('/secondVerticaStorageLocation/','node2','122','140');
```

See Also

ADD_LOCATION (page 426)

MEASURE_LOCATION_PERFORMANCE (page 511)

Measuring Location Performance and Setting Location Performance in the Administrator's Guide

SET_OBJECT_STORAGE_POLICY

Creates or changes an object storage policy by associating a database object with a labeled storage location.

NOTE: You cannot create a storage policy on a USER type storage location.

Syntax

```
SET_OBJECT_STORAGE_POLICY ( 'object_name', 'location_label' [, 'key_min,
key_max'] [, 'enforce_storage_move' ] )
```

Parameters

object_name	Identifies the database object assigned to a labeled storage location. The *object_name* can resolve to a database, schema, or table.
location_label	The label of the storage location with which *object_name* is being associated.

key_min, key_max	Applicable only when *object_name* is a table, key_min and key_max specify the table partition key value range to be stored at the location.
enforce_storage_move={true | false}	[Optional] Applicable only when setting a storage policy for an object that has data stored at another labeled location. Specify this parameter as `true` to move all existing storage data to the target location within this function's transaction.

Privileges

Must be the object owner to set the storage policy, and have access to the storage location.

New Storage Policy

If an object does not have a storage policy, this function creates a new policy. The labeled location is then used as the default storage location during TM operations, such as moveout and mergeout.

Existing Storage Policy

If the object already has an active storage policy, calling this function changes the default storage for the object to the new labeled location. Any existing data stored on the previous storage location is marked to move to the new location during the next TM moveout operations, unless you use the *enforce_storage_move* option.

Forcing Existing Data Storage to a New Storage Location

You can optionally use this function to move existing data storage to a new location as part of completing the current transaction, by specifying the last parameter as `true`.

To move existing data as part of the next TM moveout, either omit the parameter, or specify its value as f`alse`.

NOTE: Specifying the parameter as `true` performs a cluster-wide operation. If an error occurs on any node, the function displays a warning message, skips the offending node, and continues execution on the remaining nodes.

Example

This example sets a storage policy for the table `states` to use the storage labeled `SSD` as its default location:

```
VMART=> select set_object_storage_policy ('states', 'SSD');
     set_object_storage_policy
---------------------------------
 Default storage policy set.
(1 row)
```

See Also

ALTER_LOCATION_LABEL (page 430)

CLEAR_OBJECT_STORAGE_POLICY (page 457)

Creating Storage Policies in the Administrator's Guide

Moving Data Storage Locations in the Administrator's Guide

Tuple Mover Functions

This section contains tuple mover functions specific to HP Vertica.

DO_TM_TASK

Runs a Tuple Mover operation on one or more projections defined on the specified table.

Tip: You do not need to stop the Tuple Mover to run this function.

Syntax

```
DO_TM_TASK ( 'task' [ , '[[db-name.]schema.]table' |
'[[db-name.]schema.]projection' ] )
```

Parameters

task	Is one of the following tuple mover operations: ▪ `'moveout'` — Moves out all projections on the specified table (if a particular projection is not specified) from WOS to ROS. ▪ `'mergeout'` — Consolidates ROS containers and purges deleted records. ▪ `'analyze_row_count'` — Automatically collects the number of rows in a projection every 60 seconds and aggregates row counts calculated during loads.
[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Runs a tuple mover operation for all projections within the specified table. When using more than one schema, specify

	the schema that contains the table with the projections you want to affect, as noted above.
projection	If *projection* is not passed as an argument, all projections in the system are used. If *projection* is specified, DO_TM_TASK looks for a projection of that name and, if found, uses it; if a named projection is not found, the function looks for a table with that name and, if found, moves out all projections on that table.

Privileges

- Any INSERT/UPDATE/DELETE privilege on table
- USAGE privileges on schema

Notes

`DO_TM_TASK()` is useful for moving out all projections from a table or database without having to name each projection individually.

Examples

The following example performs a moveout of all projections for table `t1`:

```
=> SELECT DO_TM_TASK('moveout', 't1');
```

The following example performs a moveout for projection t1_proj:

```
=> SELECT DO_TM_TASK('moveout', 't1_proj')
```

See Also

COLUMN_STORAGE (page 992)

DROP_PARTITION (page 473)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

Partitioning Tables in the Administrator's Guide

Collecting Statistics in the Administrator's Guide

Workload Management Functions

This section contains workload management functions specific to HP Vertica.

ANALYZE_WORKLOAD

Runs the Workload Analyzer (WLA), a utility that analyzes system information held in ***system tables*** (page 933).

The Workload Analyzer intelligently monitors the performance of SQL queries and workload history, resources, and configurations to identify the root causes for poor query performance. Calling the ANALYZE_WORKLOAD function returns tuning recommendations for all events within the scope and time that you specify.

Tuning recommendations are based on a combination of statistics, system and data collector events, and database-table-projection design. WLA's recommendations let database administrators quickly and easily tune query performance without needing sophisticated skills.

See Understanding WLA Triggering Conditions in the Administrator's Guide for the most common triggering conditions and recommendations.

Syntax 1

```
ANALYZE_WORKLOAD ( 'scope' , 'since_time' );
```

Syntax 2

```
ANALYZE_WORKLOAD ( 'scope' , [ true ] );
```

Parameters

scope	Specifies which HP Vertica catalog objects to analyze. Can be one of: ▪ An empty string (`''`) returns recommendations for all database objects ▪ 'table_name' returns all recommendations related to the specified table ▪ 'schema_name' returns recommendations on all database objects in the specified schema

since_time	Limits the recommendations from all events that you specified in 'scope' since the specified time in this argument, up to the current system status. If you omit the since_time parameter, ANALYZE_WORKLOAD returns recommendations on events since the last recorded time that you called this function.
	Note: You must explicitly cast strings that you use for the since_time parameter to TIMESTAMP or TIMESTAMPTZ. For example:
	`SELECT ANALYZE_WORKLOAD('T1', '2010-10-04 11:18:15'::TIMESTAMPTZ);`
	`SELECT ANALYZE_WORKLOAD('T1', TIMESTAMPTZ '2010-10-04 11:18:15');`

true	[Optional] Tells HP Vertica to record this particular call of WORKLOAD_ANALYZER() in the system. The default value is false (do not record). If recorded, subsequent calls to ANALYZE_WORKLOAD analyze only the events that have occurred since this recorded time, ignoring all prior events.

Return value

ANALYZE_WORKLOAD() returns aggregated tuning recommendations, as described in the following table.

Column	Data type	Description
observation_count	INTEGER	Integer for the total number of events observed for this tuning recommendation. For example, if you see a return value of 1, WLA is making its first tuning recommendation for the event in 'scope'.
first_observation_time	TIMESTAMPTZ	Timestamp when the event first occurred. If this column returns a null value, the tuning recommendation is from the current status of the system instead of from any prior event.
last_observation_time	TIMESTAMPTZ	Timestamp when the event last occurred. If this column returns a null value, the tuning recommendation is from the current status of the system instead of from any prior event.
tuning_parameter	VARCHAR	Objects on which you should perform a tuning action. For example, a return value of: ▪ **`public.t`** informs the DBA to run Database Designer on table t in the public schema ▪ **`bsmith`** notifies a DBA to set a password for user bsmith
tuning_description	VARCHAR	Textual description of the tuning recommendation from the Workload Analyzer to perform on the tuning_parameter object. Examples of some of the returned values include, but are not limited to: ▪ Run database designer on table `schema.table` ▪ Create replicated projection for table `schema.table` ▪ Consider query-specific design on query ▪ Reset configuration parameter with `SELECT set_config_parameter('parameter', 'new_value')` ▪ Re-segment projection `projection-name` on high-cardinality column(s) ▪ Drop the projection `projection-name` ▪ Alter a table's partition expression ▪ Reorganize data in partitioned table ▪ Decrease the MoveOutInterval configuration parameter setting

tuning_command	VARCHAR	Command string if tuning action is a SQL command. For example, the following example statements recommend that the DBA:
		Update statistics on a particular schema's table.column:
		`SELECT ANALYZE_STATISTICS('public.table.column');`
		Resolve mismatched configuration parameter 'LockTimeout':
		`SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'LockTimeout';`
		Set the password for user bsmith:
		`ALTER USER (user) IDENTIFIED BY ('new_password');`

tuning_cost	VARCHAR	Cost is based on the type of tuning recommendation and is one of: ▪ LOW—minimal impact on resources from running the tuning command ▪ MEDIUM—moderate impact on resources from running the tuning command ▪ HIGH—maximum impact on resources from running the tuning command Depending on the size of your database or table, consider running high-cost operations after hours instead of during peak load times.

Privileges

Must be a superuser

Examples

See Analyzing Workloads through an API in the Administrator's Guide for examples.

See also

V_MONITOR.TUNING_RECOMMENDATIONS (page 1120) in this guide

Analyzing Workloads in the Administrator's Guide

Understanding WLA Triggering Conditions in the Administrator's Guide

CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY

Changes the run-time priority of a query that is actively running.

Syntax

`CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY(TRANSACTION_ID, 'value')`

Parameters

TRANSACTION_ID	An identifier for the transaction within the session. TRANSACTION_ID cannot be NULL. You can find the transaction ID in the Sessions table.
'value'	The RUNTIMEPRIORITY value. Can be HIGH, MEDIUM, or LOW.

Privileges

No special privileges required. However, non-super users can change the run-time priority of their own queries only. In addition, non-superusers can never raise the run-time priority of a query to a level higher than that of the resource pool.

Example

```
VMart => SELECT CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY(45035996273705748,
'low')
```

CHANGE_RUNTIME_PRIORITY

Changes the run-time priority of a query that is actively running. Note that, while this function is still valid, you should instead use CHANGE_CURRENT_STATEMENT_RUNTIME_PRIORITY to change run-time priority. CHANGE_RUNTIME_PRIORITY will be deprecated in a future release of Vertica.

Syntax

```
CHANGE_RUNTIME_PRIORITY(TRANSACTION_ID,STATEMENT_ID, 'value')
```

Parameters

TRANSACTION_ID	An identifier for the transaction within the session. TRANSACTION_ID cannot be NULL. You can find the transaction ID in the Sessions table.
STATEMENT_ID	A unique numeric ID assigned by the HP Vertica catalog, which identifies the currently executing statement. You can find the statement ID in the Sessions table. You can specify NULL to change the run-time priority of the currently running query within the transaction.
'value'	The RUNTIMEPRIORITY value. Can be HIGH, MEDIUM, or LOW.

Privileges

No special privileges required. However, non-super users can change the run-time priority of their own queries only. In addition, non-superusers can never raise the run-time priority of a query to a level higher than that of the resource pool.

Example

```
VMart => SELECT CHANGE_RUNTIME_PRIORITY(45035996273705748, NULL, 'low')
```

CLEAR_CACHES

Clears the HP Vertica internal cache files.

Syntax

```
CLEAR_CACHES ( )
```

Privileges

Must be a superuser

Notes

If you want to run benchmark tests for your queries, in addition to clearing the internal HP Vertica cache files, clear the Linux file system cache. The kernel uses unallocated memory as a cache to hold clean disk blocks. If you are running version 2.6.16 or later of Linux and you have root access, you can clear the kernel filesystem cache as follows:

1. Make sure that all data is the cache is written to disk:
   ```
   # sync
   ```
2. Writing to the `drop_caches` file causes the kernel to drop clean caches, dentries, and inodes from memory, causing that memory to become free, as follows:
   - To clear the page cache:
     ```
     # echo 1 > /proc/sys/vm/drop_caches
     ```
   - To clear the dentries and inodes:
     ```
     # echo 2 > /proc/sys/vm/drop_caches
     ```
   - To clear the page cache, dentries, and inodes:
     ```
     # echo 3 > /proc/sys/vm/drop_caches
     ```

Example

The following example clears the HP Vertica internal cache files:

```
=> CLEAR_CACHES();
 CLEAR_CACHES
--------------
 Cleared
(1 row)
```

SLEEP

Waits a specified number of seconds before executing another statement or command.

Syntax

```
SLEEP( seconds )
```

Parameters

seconds	The wait time, specified in one or more seconds (0 or higher) expressed as a positive integer. Single quotes are optional; for example, `SLEEP(3)` is the same as `SLEEP('3')`.

Notes

- This function returns value 0 when successful; otherwise it returns an error message due to syntax errors.
- You cannot cancel a sleep operation.
- Be cautious when using SLEEP() in an environment with shared resources, such as in combination with transactions that take exclusive locks.

Example

The following command suspends execution for 100 seconds:

```
=> SELECT SLEEP(100);
 sleep
-------
     0
(1 row)
```

SQL Statements

The primary structure of a SQL query is its statement. Multiple statements are separated by semicolons; for example:

```
CREATE TABLE fact ( ..., date_col date NOT NULL, ...);
CREATE TABLE fact(..., state VARCHAR NOT NULL, ...);
```

ALTER FUNCTION

Alters a user-defined SQL function or user defined function (UDF) by providing a new function or different schema name or my modifying its fenced mode setting.

Syntax 1

```
ALTER FUNCTION
... [[db-name.]schema.]function-name ( [ [ argname ] argtype  [, ...] ] )
... RENAME TO new_name
... SET FENCED bool_val
```

Syntax 2

```
ALTER FUNCTION
... [[db-name.]schema.]function-name ( [ [ argname ] argtype  [, ...] ] )
... SET SCHEMA new_schema
... SET FENCED bool_val
```

Syntax 3

```
ALTER FUNCTION
... [[db-name.]schema.]function-name ( [ [ argname ] argtype  [, ...] ] )
... SET FENCED bool_val
```

Parameters

`[[db-name.]schema-name.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
function-name	The name of the user-defined SQL Function (function body) to alter. If the function name is schema-qualified (as described above), the function is altered in the specified schema.
argname	Specifies the name of the argument.

argtype	Specifies the data type for argument that is passed to the function. Argument types must match HP Vertica type names. See ***SQL Data Types*** (page 71).
RENAME TO *new_name*	Specifies the new name of the function
SET SCHEMA *new_schema*	Specifies the new schema name where the function resides.
SET FENCED *bool_val*	A boolean value that specifies if Fenced Mode is enabled for this function. Fenced Mode is not available for User Defined Aggregates or User Defined Load.

Permissions

Only a superuser or owner can alter a function.

To rename a function (ALTER FUNCTION RENAME TO) the user must have USAGE and CREATE privilege on schema that contains the function.

To specify a new schema (ALTER FUNCTION SET SCHEMA), the user must have USAGE privilege on schema that currently contains the function (old schema) and CREATE privilege on the schema to which the function will be moved (new schema).

Notes

When you alter a function you must specify the argument type, because there could be several functions that share the same name with different argument types.

Example

This example creates a SQL function called `zerowhennull` that accepts an `INTEGER` argument and returns an `INTEGER` result.

```
=> CREATE FUNCTION zerowhennull(x INT) RETURN INT
   AS BEGIN
     RETURN (CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END);
   END;
```

This next command renames the `zerowhennull` function to `zeronull`:

```
=> ALTER FUNCTION zerowhennull(x INT) RENAME TO zeronull;
ALTER FUNCTION
```

This command moves the renamed function to a new schema called `macros`:

```
=> ALTER FUNCTION zeronull(x INT) SET SCHEMA macros;
ALTER FUNCTION
```

This command disables Fenced Mode for the `Add2Ints` function:

```
=> ALTER FUNCTION Add2Ints(INT, INT) SET FENCED false;
ALTER FUNCTION
```

See Also

CREATE FUNCTION (page 722)

DROP FUNCTION (page 811)

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Using SQL Macros in the Programmer's Guide

ALTER LIBRARY

Replaces the Linux shared object library file (.so) or R file for an already-defined library with a new file. The new file is automatically distributed throughout the HP Vertica cluster. See Developing and Using User Defined Functions in the Programmer's Guide for details. All of the functions that reference the library automatically begin using the new library file after it is loaded.

Note: The new library must be developed in the same language as the library file being replaced. For example, you cannot use this statement to replace a C++ library file with an R library file.

Syntax

```
ALTER LIBRARY [[db-name.]schema.]library_name AS 'library_path';
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
library_name	The name of the library being altered. This library must have already been created using ***CREATE LIBRARY*** (page 735).
library_path	The absolute path to the replacement library file. The file must be the same type as the library file used by the current library definition.

Permissions

Must be a superuser to alter a library.

Notes

- All of the UDFs that reference the library begin calling the code in the updated library file once it has been distributed to all of the nodes in the HP Vertica cluster.
- Any nodes that are down or that are added to the cluster later automatically receive a copy of the updated library file when they join the cluster.
- HP Vertica does not compare the functions defined in the new library to ensure they match any currently-defined functions in the catalog. If you change the signature of a function in the library (for example, if you change the number and data types accepted by a UDSF defined in the library), calls to that function will likely generate errors. If your new library file changes the definition of a function, you must remove the function using ***DROP FUNCTION*** (page 811) before using ALTER LIBRARY to load the new library. You can then recreate the function using its new signature.

ALTER PROJECTION RENAME

Initiates a rename operation on the specified projection.

Syntax

```
ALTER PROJECTION [ [db-name.]schema.]projection-name RENAME TO
new-projection-name
```

Parameters

`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection-name	Specifies the projection to change. You must include the base table name prefix that is added automatically when you create a projection.
new-projection-name	Specifies the new projection name.

Permissions

To rename a projection, the user must own the anchor table for which the projection was created and have USAGE and CREATE privileges on the schema that contains the projection.

Notes

The projection must exist before it can be renamed.

See Also

CREATE PROJECTION (page 742)

ALTER NETWORK INTERFACE

Lets you rename a network interface.

Syntax

```
ALTER NETWORK INTERFACE network-interface-name RENAME TO
new-network-interface-name
```

The parameters are defined as follows:

network-interface-name	The name of the existing network interface.
new-network-interface-name	The new name for the network interface.

Permissions

Must be a superuser to alter a network interface.

ALTER PROFILE

Changes a profile. Only a database superuser can alter a profile.

Syntax

```
ALTER PROFILE name LIMIT
... [PASSWORD_LIFE_TIME {life-limit | DEFAULT | UNLIMITED}]
... [PASSWORD_GRACE_TIME {grace_period | DEFAULT | UNLIMITED}]
... [FAILED_LOGIN_ATTEMPTS {login-limit | DEFAULT | UNLIMITED}]
... [PASSWORD_LOCK_TIME {lock-period | DEFAULT | UNLIMITED}]
... [PASSWORD_REUSE_MAX {reuse-limit | DEFAULT | UNLIMITED}]
... [PASSWORD_REUSE_TIME {reuse-period | DEFAULT | UNLIMITED}]
... [PASSWORD_MAX_LENGTH {max-length | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_LENGTH {min-length | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_LETTERS {min-letters | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_UPPERCASE_LETTERS {min-cap-letters | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_LOWERCASE_LETTERS {min-lower-letters | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_DIGITS {min-digits | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_SYMBOLS {min-symbols | DEFAULT | UNLIMITED}]
```

Note: For all parameters, the special value DEFAULT means the parameter is inherited from the DEFAULT profile.

Parameters

Name	Description	Meaning of UNLIMITED value
name	The name of the profile to create	N/A

PASSWORD_LIFE_TIME *life-limit*	Integer number of days a password remains valid. After the time elapses, the user must change the password (or will be warned that their password has expired if PASSWORD_GRACE_TIME is set to a value other than zero or UNLIMITED).	Passwords never expire.
PASSWORD_GRACE_TIME *grace-period*	Integer number of days the users are allowed to login (while being issued a warning message) after their passwords are older than the PASSWORD_LIFE_TIME. After this period expires, users are forced to change their passwords on login if they have not done so after their password expired.	No grace period (the same as zero)
FAILED_LOGIN_ATTEMPTS *login-limit*	The number of consecutive failed login attempts that result in a user's account being locked.	Accounts are never locked, no matter how many failed login attempts are made.
PASSWORD_LOCK_TIME *lock-period*	Integer value setting the number of days an account is locked after the user's account was locked by having too many failed login attempts. After the PASSWORD_LOCK_TIME has expired, the account is automatically unlocked.	Accounts locked because of too many failed login attempts are never automatically unlocked. They must be manually unlocked by the database superuser.
PASSWORD_REUSE_MAX *reuse-limit*	The number of password changes that need to occur before the current password can be reused.	Users are not required to change passwords a certain number of times before reusing an old password.
PASSWORD_REUSE_TIME *reuse-period*	The integer number of days that must pass after a password has been set before the before it can be reused.	Password reuse is not limited by time.
PASSWORD_MAX_LENGTH *max-length*	The maximum number of characters allowed in a password. Value must be in the range of 8 to 100.	Passwords are limited to 100 characters.
PASSWORD_MIN_LENGTH *min-length*	The minimum number of characters required in a password. Valid range is 0 to *max-length*.	Equal to *max-length*.

PASSWORD_MIN_LETTERS *min-of-letters*	Minimum number of letters (a-z and A-Z) that must be in a password. Valid ranged is 0 to *max-length*.	0 (no minimum).
PASSWORD_MIN_UPPERCASE_LETTERS *min-cap-letters*	Minimum number of capital letters (A-Z) that must be in a password. Valid range is is 0 to *max-length*.	0 (no minimum).
PASSWORD_MIN_LOWERCASE_LETTERS *min-lower-letters*	Minimum number of lowercase letters (a-z) that must be in a password. Valid range is is 0 to *max-length*.	0 (no minimum).
PASSWORD_MIN_DIGITS *min-digits*	Minimum number of digits (0-9) that must be in a password. Valid range is is 0 to *max-length*.	0 (no minimum).
PASSWORD_MIN_SYMBOLS *min-symbols*	Minimum number of symbols (any printable non-letter and non-digit character, such as $, #, @, and so on) that must be in a password. Valid range is is 0 to *max-length*.	0 (no minimum).

Permissions

Must be a superuser to alter a profile.

Note: Only the profile settings for how many failed login attempts trigger account locking and how long accounts are locked have an effect on external password authentication methods such as LDAP or Kerberos. All password complexity, reuse, and lifetime settings have an effect on passwords managed by HP Vertica only.

See Also

CREATE PROFILE (page 739)

DROP PROFILE (page 817)

ALTER PROFILE RENAME

Rename an existing profile.

Syntax

```
ALTER PROFILE name RENAME TO newname;
```

Parameters

name	The current name of the profile.
newname	The new name for the profile.

Permissions

Must be a superuser to alter a profile.

See Also

ALTER PROFILE (page 660)

CREATE PROFILE (page 739)

DROP PROFILE (page 817)

ALTER RESOURCE POOL

Modifies a resource pool. The resource pool must exist before you can issue the `ALTER RESOURCE POOL` command.

Syntax

```
ALTER RESOURCE POOL pool-name MEMORYSIZE 'sizeUnits'
... [ MAXMEMORYSIZE 'sizeUnits' ]
... [ PRIORITY {integer | DEFAULT }]
... [ EXECUTIONPARALLELISM {integer | AUTO | DEFAULT} ]
... [ RUNTIMEPRIORITY (HIGH | MEDIUM | LOW | DEFAULT) ]
... [ RUNTIMEPRIORITYTHRESHOLD {integer |DEFAULT }]
... [ QUEUETIMEOUT {integer | NONE |DEFAULT }]
... [ PLANNEDCONCURRENCY {integer | DEFAULT | AUTO} ]
... [ RUNTIMECAP {interval | NONE |DEFAULT} ]
... [ MAXCONCURRENCY {integer | NONE | DEFAULT }]
... [ SINGLEINITIATOR { bool | DEFAULT}]
```

Parameters

Note: If you set any of these parameters to `DEFAULT`, HP Vertica sets the parameter to the value stored in `RESOURCE_POOL_DEFAULTS`.

pool-name	Specifies the name of the resource pool to alter. Resource pool names are subject to the same rules as HP Vertica ***identifiers*** (page 22). ***Built-in pool*** (page 757) names cannot be used for user-defined pools.

MEMORYSIZE *'sizeUnits'*	[Default 0%] The amount of memory allocated to this pool per node and not across the whole cluster. The default of 0% means that the pool has no memory allocated to it and must exclusively borrow from the GENERAL ***pool*** (page 757). Units can be one of the following: ▪ Percentage (%) of total memory available to the Resource Manager. (In this case size must be 0-100). ▪ K Kilobytes ▪ M Megabytes ▪ G Gigabytes ▪ T Terabytes See also MAXMEMORYSIZE parameter.
MAXMEMORYSIZE *'sizeUnits'* | NONE	[Default unlimited] Maximum size the resource pool could grow by borrowing memory from the GENERAL pool. See ***Built-in Pools*** (page 757) for a discussion on how resource pools interact with the GENERAL pool. Units can be one of the following: ▪ % percentage of total memory available to the Resource Manager. (In this case, size must be 0-100). This notation has special meaning for the GENERAL pool, described in Notes below. ▪ K Kilobytes ▪ M Megabytes ▪ G Gigabytes ▪ T Terabytes If MAXMEMORYSIZE NONE is specified, there is no upper limit. **Notes:** The MAXMEMORYSIZE parameter refers to the maximum memory borrowed by this pool per node and not across the whole cluster. The default of unlimited means that the pool can borrow as much memory from GENERAL pool as is available. The MAXMEMORYSIZE of the WOSDATA and SYSDATA pools cannot be changed as long as any of their memory is in use. For example, in order to change the MAXMEMORYSIZE of the WOSDATA pool, you need to disable any trickle loading jobs and wait until the WOS is empty before you can change the MAXMEMORYSIZE.
EXECUTIONPARALLELISM	[Default: AUTO] Limits the number of threads used to process any single query issued in this resource pool. When set to AUTO, HP Vertica sets this value based on the number of cores, available memory, and amount of data in the system. Unless data is limited, or the amount of data is very small, HP Vertica sets this value to the number of cores on the node. Reducing this value increases the throughput of short queries issued in the pool, especially if the queries are executed concurrently. If you choose to set this parameter manually, set it to a value between

	1 and the number of cores.
RUNTIMEPRORITY	[Default: MEDIUM] Determines the amount of run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to queries already running in the resource pool. Valid values are: ▪ HIGH ▪ MEDIUM ▪ LOW Queries with a HIGH run-time priority are given more CPU and I/O resources than those with a MEDIUM or LOW run-time priority.
RUNTIMEPRIORITYTHRESHOLD	[Default 2] Specifies a time limit (in seconds) by which a query must finish before the Resource Manager assigns to it the RUNTIMEPRIORITY of the resource pool. All queries begin runnng at a HIGH priority. When a query's duration exceeds this threshold, it is assigned the RUNTIMEPRIORITY of the resource pool.
PRIORITY	[Default 0] An integer that represents priority of queries in this pool, when they compete for resources in the GENERAL pool. Higher numbers denote higher priority. Administrator-created resource pools can have a priority of -100 to 100. The built-in resource pools SYSQUERY, RECOVERY, and TM can have a range of -110 to 110.
QUEUETIMEOUT	[Default 300 seconds] An integer, in seconds, that represents the maximum amount of time the request is allowed to wait for resources to become available before being rejected. If set to NONE, the request can be queued for an unlimited amount of time.
RUNTIMECAP	[Default: NONE] Sets the maximum amount of time any query on the pool can execute. Set RUNTIMECAP using interval, such as '1 minute' or '100 seconds' (see ***Interval Values*** (page 37) for details). This value cannot exceed one year. Setting this value to NONE specifies that there is no time limit on queries running on the pool. If the user or session also has a RUNTIMECAP, the shorter limit applies.
PLANNEDCONCURRENCY	[Default: AUTO] When set to AUTO, this value is calculated automatically at query runtime. HP Vertica sets this parameter to the lower of these two calculations: • Number of cores • Memory/2GB When this parameter is set to AUTO, HP Vertica will not choose a value lower than 4. HP Vertica advises changing this value only after evaluating performance over a period of time. **Notes:** ▪ The PLANNEDCONCURRENCY setting for the GENERAL pool defaults to a too-small value for machines with large numbers of cores. To adjust to a more appropriate value:

	▪ => ALTER RESOURCE POOL general PLANNEDCONCURRENCY <#cores>; ▪ This is a cluster-wide maximum and not a per-node limit. ▪ For clusters where the number of cores differs on different nodes, AUTO can apply differently on each node. Distributed queries run like the minimal effective planned concurrency. Single node queries run with the planned concurrency of the initiator. ▪ If you created or upgraded your database in 4.0 or 4.1, the PLANNEDCONCURRENCY setting on the GENERAL pool defaults to a too-small value for machines with large numbers of cores. To adjust to a more appropriate value: ▪ => ALTER RESOURCE POOL general PLANNEDCONCURRENCY ▪ <#cores>; ▪ You need to set this parameter only if you created a database before 4.1, patchset 1. See Guidelines for Setting Pool Parameters in the Administrator's Guide
SINGLEINITIATOR	[Default false] This parameter is included for backwards compatibility only. Do not change the value.
MAXCONCURRENCY	[Default unlimited] An integer that represents the maximum number of concurrent execution slots available to the resource pool. If MAXCONCURRENCY NONE is specified, there is no limit. **Note:** This is a cluster wide maximum and NOT a per-node limit.

Permissions

Must be a superuser on the resource pool for the following parameters:

- MAXMEMORYSIZE
- PRIORITY
- QUEUETIMEOUT

The following parameters require UPDATE privileges:

- PLANNEDCONCURRENCY
- SINGLEINITIATOR
- MAXCONCURRENCY

Notes

- New resource pools can be created or altered without shutting down the system. The only exception is that changes to GENERAL.MAXMEMORYSIZE take effect only on a node restart. When a new pool is created (or its size altered), MEMORYSIZE amount of memory is taken out of the GENERAL pool. If the GENERAL pool does not currently have sufficient memory to create the pool due to existing queries being processed, a request is made to the system to create a pool as soon as resources become available. The pool is in operation as soon as the specified amount of memory becomes available. You can monitor whether the ALTER has been completed in the V_MONITOR.RESOURCE_POOL_STATUS (page 965) system table.

- If the GENERAL.MAXMEMORYSIZE parameter is modified while a node is down, and that node is restarted, the restarted node sees the new setting whereas other nodes continue to see the old setting until they are restarted. HP Vertica recommends that you do not change this parameter unless absolutely necessary.
- Under normal operation, MEMORYSIZE is required to be less than MAXMEMORYSIZE and an error is returned during CREATE/ALTER operations if this size limit is violated. However, under some circumstances where the node specification changes by addition/removal of memory, or if the database is moved to a different cluster, this invariant could be violated. In this case, MAXMEMORYSIZE is reduced to MEMORYSIZE.
- If two pools have the same PRIORITY, their requests are allowed to borrow from the GENERAL pool in order of arrival.

See Guidelines for Setting Pool Parameters in the Administrator's Guide for details about setting these parameters.

See Also

CREATE RESOURCE POOL (page 753)

CREATE USER (page 801)

DROP RESOURCE POOL (page 819)

RESOURCE_POOL_STATUS (page 1083)

SET SESSION RESOURCE POOL (page 916)

SET SESSION MEMORYCAP (page 915)

Managing Workloads in the Administrator's Guide

ALTER ROLE RENAME

Rename an existing role.

Syntax

```
ALTER ROLE name RENAME [TO] new_name;
```

Parameters

name	The current name of the role that you want to rename.
new_name	The new name for the role.

Permissions

Must be a superuser to rename a role.

Example

```
=> ALTER ROLE applicationadministrator RENAME TO appadmin;
```

```
ALTER ROLE
```

See Also

CREATE ROLE (page 764)

DROP ROLE (page 820)

ALTER SCHEMA

Renames one or more existing schemas.

Syntax

```
ALTER SCHEMA [db-name.]schema-name [ , ... ]
... RENAME TO new-schema-name [ , ... ]
```

Parameters

[db-name.]	[Optional] Specifies the current database name. Using a database name prefix is optional, and does not affect the command in any way. You must be connected to the specified database.
schema-name	Specifies the name of one or more schemas to rename.
RENAME TO	Specifies one or more new schema names. The lists of schemas to rename and the new schema names are parsed from left to right and matched accordingly using one-to-one correspondence. When renaming schemas, be sure to follow these standards: ▪ The number of schemas to rename must match the number of new schema names supplied. ▪ The new schema names must not already exist. The RENAME TO parameter is applied atomically. Either all the schemas are renamed or none of the schemas are renamed. If, for example, the number of schemas to rename does not match the number of new names supplied, none of the schemas are renamed. **Note:** Renaming a schema that is referenced by a view will cause the view to fail unless another schema is created to replace it.

Privileges

Schema owner or user requires CREATE privilege on the database

Notes

Renaming schemas does not affect existing pre-join projections because pre-join projections refer to schemas by the schemas' unique numeric IDs (OIDs), and the OIDs for schemas are not changed by ALTER SCHEMA.

Tip

Renaming schemas is useful for swapping schemas without actually moving data. To facilitate the swap, enter a non-existent, temporary placeholder schema. The following example uses the temporary schema *temps* to facilitate swapping schema S1 with schema S2. In this example, *S1* is renamed to *temps*. Then *S2* is renamed to *S1*. Finally, *temps* is renamed to *S2*.

```
ALTER SCHEMA S1, S2, temps
    RENAME TO temps, S1, S2;
```

Examples

The following example renames schema S1 to S3 and schema S2 to S4:

```
ALTER SCHEMA S1, S2
    RENAME TO S3, S4;
```

See Also

CREATE SCHEMA (page 764) and ***DROP SCHEMA*** (page 821)

ALTER SEQUENCE

Changes the attributes of an existing sequence. All changes take effect in the next database session. Any parameters not set during an ALTER SEQUENCE statement retain their prior settings. You must be a sequence owner or a superuser to use this statement.

Note: You can rename an existing sequence, or the schema of a sequence, but neither of these changes can be combined with any other optional parameters.

Syntax

```
ALTER SEQUENCE [[db-name.]schema.]sequence-name
... [ RENAME TO new-name | SET SCHEMA new-schema-name ]
... [ OWNER TO new-owner-name ]
... |
... [ INCREMENT [ BY ] increment-value ]
... [ MINVALUE minvalue | NO MINVALUE ]
... [ MAXVALUE maxvalue | NO MAXVALUE ]
... [ RESTART [ WITH ] restart ]
... [ CACHE cache ]
... [ CYCLE | NO CYCLE ]
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database

	objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
sequence-name	The name of the sequence to alter. The name must be unique among sequences, tables, projections, and views.
RENAME TO *new-name*	Renames a sequence within the same schema. To move a sequence, see SET SCHEMA below.
OWNER TO *new-owner-name*	Reassigns the current sequence owner to the specified owner. Only the sequence owner or a superuser can change ownership, and reassignment does not transfer grants from the original owner to the new owner (grants made by the original owner are dropped).
SET SCHEMA *new-schema-name*	Moves a sequence between schemas.
INCREMENT [BY] *increment-value*	Modifies how much to increment or decrement the current sequence to create a new value. A positive value increments an ascending sequence, and a negative value decrements the sequence.
MINVALUE *minvalue* | NO MINVALUE	Modifies the minimum value a sequence can generate. If you change this value and the current value exceeds the range, the current value is changed to the minimum value if increment is greater than zero, or to the maximum value if increment is less than zero.
MAXVALUE *maxvalue* | NO MAXVALUE	Modifies the maximum value for the sequence. If you change this value and the current value exceeds the range, the current value is changed to the minimum value if increment is greater than zero, or to the maximum value if increment is less than zero.
RESTART [WITH] *restart*	Changes the current value of the sequence to *restart*. The subsequent call to NEXTVAL will return the *restart* value.
CACHE [*value* | NO CACHE]	Modifies how many sequence numbers are preallocated and stored in memory for faster access. The default is 250,000 with a minimum value of 1. Specifying a value of 1 indicates that only one value can be generated at a time, since no cache is assigned. Alternatively, you can specify NO CACHE.
CYCLE | NO CYCLE	Allows you you to switch between CYCLE and NO CYCLE. The CYCLE option allows the sequence to wrap around when the maxvalue or minvalue is reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated is the minvalue or maxvalue, respectively. If NO CYCLE is specified, any calls to NEXTVAL after the sequence has reached its maximum/minimum value, return an error. The default is NO CYCLE.

Permissions

- To rename a schema, the user must be the sequence owner and have USAGE and CREATE privileges on the schema.
- To move a sequence between schemas, the user must be the sequence owner and have USAGE privilege on the schema that currently contains the sequence (old schema) and CREATE privilege on new schema to contain the sequence.

Examples

The following example modifies an ascending sequence called sequential to restart at 105:

```
ALTER SEQUENCE sequential RESTART WITH 105;
```

The following example moves a sequence from one schema to another:

```
ALTER SEQUENCE public.sequence SET SCHEMA vmart;
```

The following example renames a sequence in the Vmart schema:

```
ALTER SEQUENCE vmart.sequence RENAME TO serial;
```

The following example reassigns sequence ownership from the current owner to user Bob:

```
ALTER SEQUENCE sequential OWNER TO Bob;
```

See Also

CREATE SEQUENCE (page 765)

CURRVAL (page 353)

DROP SEQUENCE (page 822)

GRANT (Sequence) (page 838)

NEXTVAL (page 351)

Using Named Sequences, Sequence Privileges, and Changing a sequence owner in the Administrator's Guide

ALTER SUBNET

Renames an existing subnet.

Syntax

```
ALTER SUBNET subnet-name RENAME TO 'new-subnet-name'
```

Parameters

The parameters are defined as follows:

subnet-name	The name of the existing subnet.

new-subnet-name	The new name for the subnet.

Permissions

Must be a superuser to alter a subnet.

ALTER TABLE

Modifies an existing table with a new table definition.

Syntax1

```
ALTER TABLE [[db-name.]schema.]table-name {
... ADD COLUMN column-definition ( table ) (page 779)
... | ADD table-constraint (page 678)
... | ALTER COLUMN column-name
        | [ SET DEFAULT default-expression ]
        | [ DROP DEFAULT ]
        | [ { SET | DROP } NOT NULL]
        | [ SET DATA TYPE data-type ]
... | DROP CONSTRAINT constraint-name [ CASCADE | RESTRICT ]
... | [ DROP [ COLUMN ] column-name [ CASCADE | RESTRICT ] ]
... | RENAME [ COLUMN ] column TO new-column
... | SET SCHEMA new-schema-name [ CASCADE | RESTRICT ]
... | PARTITION BY partition-clause [ REORGANIZE ]
... | REORGANIZE
... | REMOVE PARTITIONING
... | OWNER TO new-owner-name  }
```

Syntax 2

```
ALTER TABLE [[db-name.]schema.]table-name [ , ... ]
... RENAME [TO] new-table-name [ , ... ]
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (mytable.column1), a schema, table, and column (myschema.mytable.column1), and as full qualification, a database, schema, table, and column (mydb.myschema.mytable.column1).

table-name	Specifies the name of the table to alter. When using more than one schema, specify the schema that contains the table. You can use ALTER TABLE in conjunction with SET SCHEMA to move only one table between schemas at a time. When using ALTER TABLE to rename one or more tables, you can specify a comma-delimited list of table names to rename.
ADD COLUMN *column-definition*	Adds a new column to table as defined by column-definition (page 779) and automatically adds the new column with a unique projection column name to all superprojections of the table. column-definition is any valid SQL function that does not contain volatile functions. For example, a constant or a function of other columns in the same table. ADD COLUMN operations take an ***O lock*** (page 1037) on the table until the operation completes, in order to prevent DELETE, UPDATE, INSERT, and COPY statements from affecting the table. One consequence of the O lock is that SELECT statements issued at SERIALIZABLE isolation level are blocked until the operation completes. You can add a column when nodes are down. For more information, see Altering Tables in the Administrator's Guide.
ADD *table-constraint*	Adds a ***table-constraint*** (page 678) to a table that does not have any associated projections. Adding a table constraint has no effect on views that reference the table. See About Constraints in the Administrator's Guide.
ALTER COLUMN *column-name* [SET DEFAULT *default-exp*] [DROP DEFAULT] [{SET | DROP} NOT NULL]	Alters an existing table column to change, drop, or establish a DEFAULT expression for the column or set or drop a NOT NULL constraint. (You can also use DROP DEFAULT to remove the default expression.) You can specify a volatile function as the default expression for a column as part of an ALTER COLUMN SET DEFAULT statement.
SET DATA TYPE *data-type*	Changes the column's data type to any type whose conversion does not require storage reorganization. The following types are the conversions that HP Vertica supports: ▪ Binary types–expansion and contraction (*cannot* convert between BINARY and VARBINARY types). ▪ Character types–all conversions allowed, even between CHAR and VARCHAR ▪ Exact numeric types–INTEGER, INT, BIGINT, TINYINT, INT8, SMALLINT, and all NUMERIC values of scale <=18 and precision 0 are interchangeable. For NUMERIC data types, you cannot alter precision, but you can change the scale in the ranges (0-18), (19-37), and so on. **Restrictions** You also cannot alter a column that is used in the CREATE PROJECTION .. SEGMENTED BY clause. To resize a segmented column, you must either create new superprojections and omit the column in the segmentation clause, or you can create a new table and

	projections with the column size that specifies the new size. The following type conversions are not allowed: ▪ Boolean to other types ▪ DATE/TIME type conversion ▪ Approximate numeric type conversions ▪ Conversions between BINARY and VARBINARY
DROP CONSTRAINT *name* [CASCADE | RESTRICT]	Drops the specified table-constraint from the table. Use the CASCADE keyword to drop a constraint upon which something else depends. For example, a FOREIGN KEY constraint depends on a UNIQUE or PRIMARY KEY constraint on the referenced columns. Use the RESTRICT keyword to drop the constraint only from the given table. **Note:** Dropping a table constraint has no effect on views that reference the table.
DROP COLUMN *column-name* [CASCADE | RESTRICT]	Drops both the specified column from the table and the ROS containers that correspond to the dropped column. Because drop operations physically purge object storage and catalog definitions (table history) from the table, AT EPOCH (historical) queries return nothing for the dropped column. **Restrictions** ▪ At the table level, you cannot drop or alter a primary key column or a column participating in the table's partitioning clause. ▪ At the projection level, you cannot drop the first column in a projection's sort order or columns that participate in the segmentation expression of a projection. ▪ All nodes must be up for the drop operation to succeed. **Using CASCADE to force a drop** You can use the CASCADE keyword to drop a column if that column: ▪ Has a constraint of any kind on it. ▪ Participates in the projection's sort order. ▪ Participates in a pre-join projection or participates in the projection's segmentation expression. Note that when a pre-join projection contains a column to be dropped with CASCADE, HP Vertica tries to drop the projection. In all cases, CASCADE tries to drop the projection(s) and will roll back if K-safety is compromised. See the Dropping a table column in the Administrator's Guide for additional details about CASCADE behavior and examples. Use the RESTRICT keyword to drop the column only from the given table.
RENAME [TO]	Renames one or more tables. In either case, the keyword changes the name of the table or tables to the specified name or names. For more information, see Altering Tables in the Administrator's Guide. Renaming a table requires USAGE and CREATE privilege on the schema that contains the table.

RENAME [COLUMN]	Renames the specified column within the table. **Note:** If a column that is referenced by a view is renamed, the column does not appear in the result set of the view even if the view uses the wild card (*) to represent all columns in the table. Recreate the view to incorporate the column's new name.
SET SCHEMA *new-schema-name* [RESTRICT | CASCADE]	Moves a table to the specified schema. You must have USAGE privilege on the old schema and CREATE privilege on new schema. SET SCHEMA supports moving only one table between schemas at a time. You cannot move temporary tables between schemas. For more information, see Altering Tables in the Administrator's Guide.
PARTITION BY *partition-clause* [REORGANIZE]	Partitions or re-partitions a table according to the *partition-clause* that you define. Existing partition keys are immediately dropped when you run the command. You can use the PARTITION BY and REORGANIZE keywords separately or together. However, you cannot use these keywords with any other clauses. Partition-clause expressions are limited in the following ways: ▪ Your partition-clause must calculate a single non-null value for each row. You can reference multiple columns, but each row must return a single value. ▪ You can specify leaf expressions, functions, and operators in the partition clause expression. ▪ All leaf expressions in the partition clause must be either constants or columns of the table. ▪ Aggregate functions and queries are not permitted in the partition-clause expression. ▪ SQL functions used in the partition-clause expression must be immutable. Partitioning or re-partitioning tables requires USAGE privilege on the schema that contains the table. See Partitioning, repartitioning, and reorganizing tables in the Administrator's Guide for details and best practices on repartitioning and reorganizing data, as well as how to monitor REORGANIZE operations. **Note:** It is not recommended to ALTER table partitioning when nodes are down because doing so prevents those nodes down from assisting in database recovery.
REMOVE PARTITIONING	Immediately removes partitioning on a table. The ROS containers are not immediately altered, but are later cleaned by the Tuple Mover.

OWNER TO *new-owner-name*	Changes the table owner. Only the table owner or a superuser can change ownership, and reassignment does not transfer grants from the original owner to the new owner (grants made by the original owner are dropped). **Note:** Changing the table owner transfers ownership of the associated IDENTITY/AUTO_INCREMENT sequences (defined in ***CREATE TABLE column-constraint*** (page 783) syntax) but not other REFERENCES sequences. See How to change a table owner and How to change a sequence owner in the Administrator's Guide.

Permissions

You must be a table owner or a superuser and have USAGE privileges on schema that contains the table in order to:

- Add, drop, rename, or alter column
- Add or drop a constraint
- Partition or re-partition the table

To rename a table, you must have USAGE and CREATE privilege on the schema that contains the table.

Moving a table to a new schema requires:

- USAGE privilege on the old schema
- CREATE privilege on new schema

Table Behavior After Alteration

After you modify a column, any new data that you load will conform to the modified table definition.

If you restore the database to an epoch other than the current epoch, the restore operation will overwrite the changes with the prior table schema. For example, if you change a column's data type from CHAR(8) to CHAR(16) in epoch 10 and you restore the database from epoch 5, the column will be CHAR(8) again.

Changing a Data Type for a Column Specified in a SEGMENTED BY Clause

If you create a table and do not create a superprojection for it, HP Vertica automatically creates a superprojection when you first load data into the table. By default, superprojections are segmented by all columns to ensure that all of the data is available for queries. If you try to alter a column used in the superprojection's segmentation clause, HP Vertica returns an error message like in the following example:

```
=> CREATE TABLE colmod (c1 VARCHAR(13), c2 VARCHAR (8), c3 INT);
CREATE TABLE
=> CREATE PROJECTION colmod_c1seg AS SELECT c1 FROM colmod
     SEGMENTED BY HASH(c1) ALL NODES;
WARNING 4116:  No super projections created for table public.colmod.
HINT:  Default super projections will be automatically created with the next DML
CREATE PROJECTION
=> ALTER TABLE colmod ALTER COLUMN c1 SET DATA TYPE VARCHAR(30);
```

```
ROLLBACK 2353:  Cannot alter type of column "c1" since it is referenced in the
segmentation expression of projection "colmod_c1seg"
```

To resize a segmented column, you must either create new superprojections and omit the column in the segmentation clause or create a new table (with new column size) and projections.

Locked tables

If the operation cannot obtain an ***O Lock*** (page 1037) on the table(s), HP Vertica attempts to close any internal Tuple Mover (TM) sessions running on the same table(s) so that the operation can proceed. Explicit TM operations that are running in user sessions are not closed. If an explicit TM operation is running on the table, then the operation cannot proceed until the explicit TM operation completes.

Examples

The following example drops the default expression specified for the `Discontinued_flag` column:

```
=> ALTER TABLE Retail.Product_Dimension
   ALTER COLUMN Discontinued_flag DROP DEFAULT;
```

The following example renames a column in the `Retail.Product_Dimension` table from `Product_description` to `Item_description`:

```
=> ALTER TABLE Retail.Product_Dimension
   RENAME COLUMN Product_description TO Item_description;
```

The following example moves table `T1` from schema `S1` to schema `S2`. SET SCHEMA defaults to CASCADE, so all the projections that are anchored on table `T1` are automatically moved to schema `S2` regardless of the schema in which they reside:

```
=> ALTER TABLE S1.T1 SET SCHEMA S2;
```

The following example adds partitioning to the `Sales` table based on `state` and reorganizes the data into partitions:

```
=> ALTER TABLE Sales PARTITION BY state REORGANIZE;
```

Adding and Changing Constraints on Columns Using ALTER TABLE

The following example uses ALTER TABLE to add a column (`b`) with not NULL and default 5 constraints to a table (`test6`):

```
CREATE TABLE test6 (a INT);
ALTER TABLE test6 ADD COLUMN b INT DEFAULT 5 NOT NULL;
```

Use ALTER TABLE with the ALTER COLUMN and SET NOT NULL clauses to add the constraint on column `a` in table `test6`:

```
ALTER TABLE test6 ALTER COLUMN a SET NOT NULL;
```

Adding and Dropping NOT NULL Column Constraints

Use the SET NOT NULL or DROP NOT NULL clause to add or remove a not NULL column constraint. Use these clauses to ensure that the column has the proper constraints when you have added or removed a primary key constraint on a column, or any time you want to add or remove the not NULL constraint.

Note: A PRIMARY KEY constraint includes a not NULL constraint, but if you drop the PRIMARY KEY constraint on a column, the not NULL constraint remains on that column.

Examples

```
ALTER TABLE T1 ALTER COLUMN x SET NOT NULL;
ALTER TABLE T1 ALTER COLUMN x DROP NOT NULL;
```

For more information, see Altering Table Definitions.

See Also

For additional examples, see Working with Tables in the Administrator's Guide

table-constraint

Adds a constraint to the metadata of a table. See Adding Constraints in the Administrator's Guide.

Syntax

```
[ CONSTRAINT constraint_name ]
... { PRIMARY KEY ( column [ , ... ] )
... | FOREIGN KEY ( column [ , ... ] ) REFERENCES table [( column [ , ... ] )]
... | UNIQUE ( column [ , ... ] )
... }
```

Parameters

CONSTRAINT *constraint-name*	Assigns a name to the constraint. HP recommends that you name all constraints.
PRIMARY KEY (*column* [, ...])	Adds a referential integrity constraint defining one or more NOT NULL columns as the primary key.
FOREIGN KEY (*column* [, ...])	Adds a referential integrity constraint defining one or more columns as a foreign key.
REFERENCES *table* [(*column* [, ...])]	Specifies the table to which the FOREIGN KEY constraint applies. If you omit the optional *column* definition of the referenced table, the default is the primary key of *table*.
UNIQUE (*column* [, ...])	Specifies that the data contained in a column or a group of columns is unique with respect to all the rows in the table.

Permissions

Table owner or user WITH GRANT OPTION is grantor.

- REFERENCES privilege on table to create foreign key constraints that reference this table
- USAGE privilege on schema that contains the table

Specifying Primary and Foreign Keys

You must define PRIMARY KEY and FOREIGN KEY constraints in all tables that participate in inner joins.

You can specify a foreign key table constraint either explicitly (with the FOREIGN KEY parameter), or implicitly using the REFERENCES parameter to reference the table with the primary key. You do not have to explicitly specify the columns in the referenced table, for example:

```
CREATE TABLE fact(c1 INTEGER PRIMARY KEY);
CREATE TABLE dim (c1 INTEGER REFERENCES fact);
```

Adding Constraints to Views

Adding a constraint to a table that is referenced in a view does not affect the view.

Examples

The Retail Sales Example Database described in the Getting Started Guide 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

Changes a database user account. Only a superuser can alter another user's database account. Making changes to a database user account with the ALTER USER function does not affect current sessions.

Database Account Changes Users Can Make

Users can change their own user accounts with these options:

- IDENTIFIED BY. . .
- RESOURCE POOL . . .
- SEARCH_PATH . . .

Users can change their own passwords using the IDENTIFIED BY option and supplying the current password with the REPLACE clause. Users can set the default RESOURCE POOL to any pool to which they have been granted usage privileges.

Syntax

```
ALTER USER name
... [ ACCOUNT { LOCK | UNLOCK } ]
... [ DEFAULT ROLE {role [, ...] | NONE} ]
... [ IDENTIFIED BY 'password' [ REPLACE 'old-password' ] ]
... [ MEMORYCAP { 'memory-limit' | NONE } ]
... [ PASSWORD EXPIRE ]
... [ PROFILE { profile-name | DEFAULT } ]
... [ RESOURCE POOL pool-name ]
... [ RUNTIMECAP { 'time-limit' | NONE } ]
... [ TEMPSPACECAP { 'space-limit' | NONE } ]
... [ SEARCH_PATH { schema[,schema2,...] | DEFAULT } ]
```

Parameters

name	Specifies the name of the user to alter. You must double quote names that contain special characters.
ACCOUNT LOCK | UNLOCK	Locks or unlocks the named user's access to the database. Users cannot log in if their account is locked. A superuser can manually lock and unlock accounts using ALTER USER syntax or automate account locking by setting a maximum number of failed login attempts through the CREATE PROFILE (page 739) statement. See also Profiles in the Administrator's Guide.
DEFAULT ROLE {role [, ...] | NONE}	One or more roles that should be active when the user's session starts. The user must have already been granted access to the roles (see ***GRANT (Role)*** (page 835)). The role or roles specified in this command replace any existing default roles. Use the NONE keyword to eliminate all default roles for the user.
IDENTIFIED BY 'password' [REPLACE 'old_password']	Sets a user's password to *password*. Supplying an empty string for *password* removes the user's password. The use of this clause differs between superusers and non-superusers. A non-superuser can alter only his or her own password, and must supply the existing password using the REPLACE parameter. Superusers can change any user's password and do not need to supply the REPLACE parameter. See Password Guidelines and Creating a Database Name and Password for password policies.
PASSWORD EXPIRE	Expires the user's password. HP Vertica will force the user to change passwords during his or her next login. **Note:** PASSWORD EXPIRE has no effect when using external password authentication methods such as LDAP or Kerberos.
PROFILE *profile-name* | DEFAULT	Sets the user's profile to *profile-name*. Using the value DEFAULT sets the user's profile to the default profile.
MEMORYCAP '*memory-limit*' | NONE	Limits the amount of memory that the user's requests can use. This value is a number representing the amount of space, followed by a unit (for example, '10G'). The unit can be one of the following: ▪ % percentage of total memory available to the Resource Manager. (In this case value of the size size must be 0-100) ▪ K Kilobytes ▪ M Megabytes ▪ G Gigabytes ▪ T Terabytes Setting this value to NONE means the user has no limits on memory use.
RESOURCE POOL *pool-name*	Sets the name of the default resource pool for the user. Attempting to alter a database user account to associate the

	account with a particular resource pool will result in an error if the user has not already been granted access to the resource pool. particular resource pool on which they have not been granted access results in an error (even for a superuser).
RUNTIMECAP '*time-limit*' | NONE	Sets the maximum amount of time any of the user's queries can execute. time-limit is an interval, such as '1 minute' or '100 seconds' (see ***Interval Values*** (page 37) for details). This value cannot exceed one year. Setting this value to NONE means there is no time limit on the user's queries. If RUNTIMECAP is also set for the resource pool or the session, HP Vertica always uses the shortest limit.
TEMPSPACECAP '*space-limit*' | NONE	Limits the amount of temporary file storage the user's requests can use. This parameter's value has the same format as the MEMORYCAP value.
SEARCH_PATH *schema*[,*schema2*,...] | DEFAULT	Sets the user's default search path that tells HP Vertica which schemas to search for unqualified references to tables and UDFs. See Setting Search Paths in the Administrator's Guide for an explanation of the schema search path. The DEFAULT keyword sets the search path to: "$user", public, v_catalog, v_monitor, v_internal

Permissions

Must be a superuser to alter a user.

See Also

CREATE USER (page 801)

DROP USER (page 826)

Managing Workloads in the Administrator's Guide

Setting a Run-Time Limit for Queries

ALTER VIEW

Renames a view.

Syntax

```
ALTER VIEW [[db-name.]schema.] current-view-name
... RENAME TO new-view-name
```

Parameters

viewname	Specifies the name of the view you want to rename.
RENAME TO *new-view-name*	Specifies the new name of the view. The view name must be unique. Do not use the same name as any table, view, or projection within the database.

Notes

Views are read only. You cannot perform insert, update, delete, or copy operations on a view.

Permissions

To create a view, the user must be a superuser or have CREATE privileges on the schema in which the view is renamed.

Example

The following command renames view1 to view2:

```
=> CREATE VIEW view1 AS SELECT * FROM t;
CREATE VIEW

=> ALTER VIEW view1 RENAME TO view2;
ALTER VIEW
```

BEGIN

Starts a transaction block.

Syntax

```
BEGIN [ WORK | TRANSACTION ] [ isolation_level ] [ transaction_mode ]
```

where *isolation_level* is one of:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
}
```

and where *transaction_mode* is one of:

```
READ { ONLY | WRITE }
```

Parameters

`WORK | TRANSACTION`	Have no effect; they are optional keywords for readability.
`ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }`	Isolation level determines what data the transaction can access when other transactions are running concurrently. The isolation level cannot be changed after the first query (SELECT) or DML statement (INSERT, DELETE, UPDATE) has run. A transaction retains its isolation level until it completes, even if the session's transaction isolation level changes mid-transaction. HP Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations are always run at `SERIALIZABLE` isolation level to ensure consistency.

	isolation_level can one of the following values: ▪ `SERIALIZABLE`—Sets the strictest level of SQL transaction isolation. This level emulates transactions serially, rather than concurrently. It holds locks and blocks write operations until the transaction completes. Not recommended for normal query operations. ▪ `REPEATABLE READ`—Automatically converted to SERIALIZABLE by HP Vertica. ▪ `READ COMMITTED` (Default)—Allows concurrent transactions. Use `READ COMMITTED` isolation or Snapshot Isolation for normal query operations, but be aware that there is a subtle difference between them. (See section below this table.) ▪ `READ UNCOMMITTED`—Automatically converted to `READ COMMITTED by` HP Vertica.
`READ { ONLY | WRITE }`	Transaction mode can be one of the following: ▪ `READ WRITE`—(default)The transaction is read/write. ▪ `READ ONLY`—The transaction is read-only. Setting the transaction session mode to read-only disallows the following SQL commands, but does not prevent all disk write operations: ▪ 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 run is among those listed.

Permissions

No special permissions required.

Notes

START TRANSACTION (page 926) performs the same function as BEGIN.

See Also

- Transactions
- Creating and Rolling Back Transactions
- ***COMMIT*** (page 697)
- ***END*** (page 827)
- ***ROLLBACK*** (page 867)

COMMENT ON Statements

COMMENT ON COLUMN

Adds, revises, or removes a projection column comment. You can only add comments to projection columns, not to table columns. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON COLUMN [[db-name.]schema.]proj_name.column_name IS {'comment' | NULL}
```

Parameters

`[[db-name.]schema.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
proj_name.column_name	Specifies the name of the projection and column with which to associate the comment.
comment	Specifies the comment text to add. Enclose the text of the comment within single-quotes. If a comment already exists for this column, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
`NULL`	Removes an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to the `customer_name` column in the customer_dimension projection:

```
=> COMMENT ON COLUMN customer_dimension_vmart_node01.customer_name IS 'Last name
only';
```

The following examples remove a comment from the `customer_name` column in the customer_dimension projection in two ways, using the `NULL` option, or specifying a blank string:

```
=> COMMENT ON COLUMN customer_dimension_vmart_node01.customer_name IS NULL;
=> COMMENT ON COLUMN customer_dimension_vmart_node01.customer_name IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON CONSTRAINT

Adds, revises, or removes a comment on a constraint. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON CONSTRAINT constraint_name ON [ [db-name.]schema.]table_name IS
... {'comment' | NULL };
```

Parameters

constraint_name	The name of the constraint associated with the comment.
[[*db-name*.]*schema*.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table_name	Specifies the name of the table constraint with which to associate a comment.

comment	Specifies the comment text to add. Enclose the text of the comment within single-quotes. If a comment already exists for this constraint, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
NULL	Removes an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to the `constraint_x` constraint on the promotion_dimension table:

```
=> COMMENT ON CONSTRAINT constraint_x ON promotion_dimension IS 'Primary key';
```

The following examples remove a comment from the `constraint_x` constraint on the promotion_dimension table:

```
=> COMMENT ON CONSTRAINT constraint_x ON promotion_dimension IS NULL;
=> COMMENT ON CONSTRAINT constraint_x ON promotion_dimension IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON FUNCTION

Adds, revises, or removes a comment on a function. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON FUNCTION [[db-name.]schema.]function_name function_arg IS { 'comment'
| NULL };
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths).

	You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
function_name	Specifies the name of the function with which to associate the comment.
function_arg	Indicates the function arguments.
comment	Specifies the comment text to add. Enclose the comment text within single-quotes. If a comment already exists for this function, the comment you enter overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
`NULL`	Removes an existing comment.

Notes

- A superuser can view and add comments to all objects.
- A user must own an object to be able to add or edit comments for the object.
- A user must have viewing privileges on an object to view its comments.
- If you drop an object, all comments associated with the object are dropped as well.

Example

The following example adds a comment to the `macros.zerowhennull (x INT)` function:

```
=> COMMENT ON FUNCTION macros.zerowhennull(x INT) IS 'Returns a 0 if not NULL';
```

The following examples remove a comment from the `macros.zerowhennull (x INT)` function in two ways by using the `NULL` option, or specifying a blank string:

```
=> COMMENT ON FUNCTION macros.zerowhennull(x INT) IS NULL;
=> COMMENT ON FUNCTION macros.zerowhennull(x INT) IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON LIBRARY

Adds, revises, or removes a comment on a library . Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON LIBRARY [ [db-name.]schema.]library_name IS {'comment' | NULL}
```

Parameters

`[[db-name.]schema.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
library_name	The name of the library associated with the comment.
comment	Specifies the comment text to add. Enclose the text of the comment within single-quotes. If a comment already exists for this library, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
`NULL`	Removes an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to the library `MyFunctions`:

```
=> COMMENT ON LIBRARY MyFunctions IS 'In development';
```

The following examples remove a comment from the library `MyFunctions`:

```
=> COMMENT ON LIBRARY MyFunctions IS NULL;

=> COMMENT ON LIBRARY MyFunctions IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON NODE

Adds, revises, or removes a comment on a node. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON NODE node_name IS  { 'comment' | NULL }
```

Parameters

node_name	The name of the node associated with the comment.
comment	Specifies the comment text to add. Enclose the text of the comment within single-quotes. If a comment already exists for this node, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
NULL	Removes an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment for the `initiator node`:

```
=> COMMENT ON NODE initiator IS 'Initiator node';
```

The following examples removes a comment from the initiator node.

```
=> COMMENT ON NODE initiator IS NULL;
```

```
=> COMMENT ON NODE initiator IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON PROJECTION

Adds, revises, or removes a comment on a projection. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON PROJECTION [ [db-name.]schema.]proj_name IS { 'comment' | NULL }
```

Parameters

[[*db-name*.]*schema*.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection_name	The name of the projection associated with the comment.
comment	Specifies the text of the comment to add. Enclose the text of the comment within single-quotes. If a comment already exists for this projection, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
Null	Removes an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to the `customer_dimension_vmart_node01` projection:

```
=> COMMENT ON PROJECTION customer_dimension_vmart_node01 IS 'Test data';
```

The following examples remove a comment from the `customer_dimension_vmart_node01` projection:

```
=> COMMENT ON PROJECTION customer_dimension_vmart_node01 IS NULL;
=> COMMENT ON PROJECTION customer_dimension_vmart_node01 IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON SCHEMA

Adds, revises, or removes a comment on a schema. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON SCHEMA [db-name.]schema_name IS {'comment' | NULL}
```

Parameters

[db-name.]	[Optional] Specifies the database name. You must be connected to the database you specify. You cannot make changes to objects in other databases.
schema_name	Indicates the schema associated with the comment.
comment	Text of the comment you want to add. Enclose the text of the comment in single-quotes. If a comment already exists for this schema, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
NULL	Allows you to remove an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to the `public` schema:

```
=> COMMENT ON SCHEMA public  IS 'All users can access this schema';
```

The following examples remove a comment from the `public` schema.

```
=> COMMENT ON SCHEMA public IS NULL;
=> COMMENT ON SCHEMA public IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON SEQUENCE

Adds, revises, or removes a comment on a sequence. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON SEQUENCE [[db-name.]schema.]sequence_name IS { 'comment' | NULL }
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
sequence_name	The name of the sequence associated with the comment.
comment	Specifies the text of the comment to add. Enclose the text of the comment within single-quotes. If a comment already exists for this sequence, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove

	an existing comment.
NULL	Removes an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to the sequence called prom_seq.

```
=> COMMENT ON SEQUENCE prom_seq IS 'Promotion codes';
```

The following examples remove a comment from the prom_seq sequence.

```
=> COMMENT ON SEQUENCE prom_seq IS NULL;
=> COMMENT ON SEQUENCE prom_seq IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON TABLE

Adds, revises, or removes a comment on a table. Each object can have a maximum of one comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON TABLE [ [db-name.]schema.]table_name IS { 'comment' | NULL }
```

Parameters

[[*db-name*.]*schema*.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table_name	Specifies the name of the table with which to associate the comment.
comment	Specifies the text of the comment to add. Enclose the text of the comment within single-quotes. If a comment already exists

	for this table, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
`Null`	Removes a previously added comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to the promotion_dimension table:

```
=> COMMENT ON TABLE promotion_dimension IS '2011 Promotions';
```

The following examples remove a comment from the promotion_dimension table:

```
=> COMMENT ON TABLE promotion_dimension IS NULL;
=> COMMENT ON TABLE promotion_dimension IS '';
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMENT ON TRANSFORM FUNCTION

Adds, revises, or removes a comment on a user-defined transform function. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the `v_catalog.comments` system table.

Syntax

```
COMMENT ON TRANSFORM FUNCTION [[db-name.]schema.]t_function_name
...([t_function_arg_name t_function_arg_type] [,...]) IS {'comment' | NULL}
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database

	objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
t_function_name	Specifies name of the transform function with which to associate the comment.
t_function_arg_name *t_function_arg_type*	[Optional] Indicates the names and data types of one or more transform function arguments. If you supply argument names and types, each type must match the type specified in the library used to create the original transform function.
comment	Specifies the comment text to add. Enclose the text of the comment within single-quotes. If a comment already exists for this transform function, the comment you enter overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
`NULL`	Removes an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to the `macros.zerowhennull (x INT)` UTF function:

```
=> COMMENT ON TRANSFORM FUNCTION macros.zerowhennull(x INT) IS 'Returns a 0 if not
NULL';
```

The following example removes a comment from the `acros.zerowhennull (x INT)` function by using the `NULL` option:

```
=> COMMENT ON TRANSFORM FUNCTION macros.zerowhennull(x INT) IS NULL;
```

COMMENT ON VIEW

Adds, revises, or removes a comment on a view. Each object can have a maximum of 1 comment (1 or 0). Comments are stored in the V_CATALOG.COMMENTS system table.

Syntax

```
COMMENT ON VIEW [ [db-name.]schema.]view_name IS { 'comment' | NULL }
```

Parameters

[[*db-name*.]*schema*.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
view_name	The name of the view with which to associate the comment.
comment	Specifies the text of the comment to add. If a comment already exists for this view, the comment you enter here overwrites the previous comment. Comments can be up to 8192 characters in length. If a comment exceeds that limitation, HP Vertica truncates the comment and alerts the user with a message. You can enclose a blank value within single quotes to remove an existing comment.
NULL	Removes an existing comment.

Permissions

- A superuser can view and add comments to all objects.
- The object owner can add or edit comments for the object.
- A user must have VIEW privileges on an object to view its comments.

Notes

Dropping an object drops all comments associated with the object.

Example

The following example adds a comment to a view called `curr_month_ship`:

```
=> COMMENT ON VIEW curr_month_ship IS 'Shipping data for the current month';
```

The following example removes a comment from the `curr_month_ship` view:

```
=> COMMENT ON VIEW curr_month_ship IS NULL;
```

See Also

V_CATALOG.COMMENTS (page 937)

COMMIT

Ends the current transaction and makes all changes that occurred during the transaction permanent and visible to other users.

Syntax

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

Parameters

WORK | TRANSACTION	Have no effect; they are optional keywords for readability.

Permissions

No special permissions required.

Notes

END (page 827) is a synonym for COMMIT.

See Also

- Transactions
- Creating and Rolling Back Transactions
- ***BEGIN*** (page 682)
- ***ROLLBACK*** (page 867)
- ***START TRANSACTION*** (page 926)

CONNECT

Connects to another HP Vertica database to enable data import (using the ***COPY FROM VERTICA*** (page 711) statement) or export (using the ***EXPORT*** (page 829) statement). By default, invoking CONNECT occurs over the HP Vertica private network. Creating a connection over a public network requires some configuration. For information about using CONNECT to export data to or import data over a public network, see Export/Import from a Public Network.

Note: When importing from or exporting to an HP Vertica database, you can connect only to a database that uses trusted- (username-only) or password-based authentication, as described in Implementing Security. Neither LDAP nor SSL authentication is supported.

Syntax

```
CONNECT TO VERTICA database USER username PASSWORD 'password' ON 'host',port
```

Parameters

database	The connection target database name.
username	The username to use when connecting to the other database.
password	A string containing the password to use to connect to the other database.
host	A string containing the hostname of one of the nodes in the other database.
port	The port number of the other database as an integer.

Permissions

No special permissions required.

Connection Details

Once you successfully establish a connection to another database, the connection remains open for the current session. To disconnect a connection, use the ***DISCONNECT*** (page 809) statement.

You can have only one connection to another database at a time, though you can create connections to multiple different databases in the same session.

If the target database does not have a password, and you specify a password in the CONNECT statement, the connection succeeds, but does not give any indication that you supplied an incorrect password.

Example

```
=> CONNECT TO VERTICA ExampleDB USER dbadmin PASSWORD 'Password123' ON 'VerticaHost01',5433;
CONNECT
```

See Also

COPY FROM VERTICA (page 711)

DISCONNECT (page 809)

EXPORT TO VERTICA (page 829)

COPY

Bulk loads data into an HP Vertica database. You can initiate loading one or more files or pipes on a cluster host, or on a client system (using the COPY LOCAL option).

Permissions

You must connect to the HP Vertica database as a superuser, or, as a non-superuser, have a USER-accessible storage location, and applicable `READ` or `WRITE` privileges granted to the storage location from which files are read or written to. COPY LOCAL users must have `INSERT` privileges to copy data from the `STDIN` pipe, as well as `USAGE` privileges on the schema.

To COPY from STDIN:

- INSERT privilege on table
- USAGE privilege on schema

Syntax

```
COPY [ [db-name.]schema-name.]table
... [ ( { column-as-expression | column }
...... [ FILLER datatype ]
...... [ FORMAT 'format' ]
...... [ ENCLOSED BY 'char' ]
...... [ ESCAPE AS 'char' | NO ESCAPE ]
...... [ NULL [ AS ] 'string' ]
...... [ TRIM 'byte' ]
...... [ DELIMITER [ AS ] 'char' ]
... [, ... ] ) ]
... [ COLUMN OPTION ( column
...... [ FORMAT 'format' ]
...... [ ENCLOSED BY 'char' ]
...... [ ESCAPE AS 'char' | NO ESCAPE ]
...... [ NULL [ AS ] 'string' ]
...... [ DELIMITER [ AS ] 'char' ]
... [, ... ] ) ]
FROM { STDIN
...... [ BZIP | GZIP | UNCOMPRESSED ]
...| 'pathToData' [ ON nodename | ON ANY NODE ]
...... [ BZIP | GZIP | UNCOMPRESSED ] [, ...]
...| LOCAL STDIN | 'pathToData'
...... [ BZIP | GZIP | UNCOMPRESSED ] [, ...]
}
...[ NATIVE | NATIVE VARCHAR | FIXEDWIDTH COLSIZES (integer [, ....]) ]
...[ WITH ]
...[ WITH  [ SOURCE source(arg='value')] [ FILTER filter(arg='value') ] [ PARSER
parser([arg='value']) ]]
...[ DELIMITER [ AS ] 'char' ]
...[ TRAILING NULLCOLS ]
...[ NULL [ AS ] 'string' ]
...[ ESCAPE AS 'char' | NO ESCAPE ]
...[ ENCLOSED BY 'char' ]
...[ RECORD TERMINATOR 'string' ]
```

```
...[ SKIP records ]
...[ SKIP BYTES integer ]
...[ TRIM 'byte' ]
...[ REJECTMAX integer ]
...[ EXCEPTIONS 'path' [ ON nodename ] [, ...] ]
...[ REJECTED DATA 'path' [ ON nodename ] [, ...] ]
...[ ENFORCELENGTH ]
...[ ABORT ON ERROR ]
...[ AUTO | DIRECT | TRICKLE ]
...[ STREAM NAME  'streamName']
...[ NO COMMIT ]
```

Parameters

table	The table containing the data to load into the HP Vertica database.
[[*db-name.*]*schema-name.*]*table*	[Optional] Specifies the name of a schema table (not a projection), optionally preceded by a database name. HP Vertica loads the data into all projections that include columns from the schema table. When using more than one schema, specify the schema that contains the table. **Note:** COPY ignores db-name or schema-name options when used as part of a CREATE EXTERNAL TABLE statement.

column-as-expression	Specifies the expression used to compute values for the target column. For example:
	`COPY t(year AS TO_CHAR(k, 'YYYY'),`
	`.`
	`.`
	Use this option to transform data when it is loaded into the target database. For more information about using expressions with COPY, see Transforming Data During Loads in the Administrator's Guide. See Ignoring Columns and Fields in the Load File in the Administrator's Guide for information about using fillers.

column	Restricts the load to one or more specified columns in the table. If you do not specify any columns, COPY loads all columns 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. If you leave the column parameter blank to load all columns in the table, you can use the optional parameter COLUMN OPTION to specify parsing options for specific columns. **Note:** 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), the following command loads the three columns of data in each record to columns C1, C6, and C9, respectively: `=> COPY T1 (C1, C6, C9);`
FILLER	Specifies not to load the column and its fields into the destination table. Use this option to omit columns that you do not want to transfer into a table. This parameter also transforms data from a source column and loads the transformed data to the destination table, rather than loading the original, untransformed source column (parsed column). (See Ignoring Columns and Fields in the Load File in the Administrator's Guide.)
FORMAT	Specifies the input formats to use when loading ***date/time*** (page 78) and ***binary*** (page 72) columns. These are the valid input formats when loading binary columns: ▪ 'octal' ▪ 'hex' ▪ 'bitstream' See Loading Binary Data to learn more about using these formats. When loading ***date/time*** (page 78) columns, using FORMAT significantly improves load performance. COPY supports the same formats as the ***TO_DATE*** (page 259) function. See the following topics for additional information: ▪ ***Template Patterns for Date/Time Formatting*** (page 265) ▪ ***Template Pattern Modifiers for Date/Time Formatting*** (page 267) If you specify invalid format strings, the COPY operation returns an error.
pathToData	Specifies the absolute path of the file containing the data, which can be from multiple input sources. If *path* resolves to a storage location, and the user invoking COPY is not a superuser, these are the required privileges:

	▪ The storage location must have been created with the USER option (see ***ADD_LOCATION*** (page 426)) ▪ The user must already have been granted READ access to the storage location where the file(s) exist, as described in ***GRANT (Storage Location)*** (page 839) Further, if a non-superuser invokes COPY from a storage location to which she has privileges, HP Vertica also checks any symbolic links (symlinks) the user has to ensure no symlink can access an area to which the user has not been granted privileges. The *pathToData* can optionally contain wildcards to match more than one file. The file or files must be accessible to the local client or the host on which the COPY statement runs. You can use variables to construct the pathname as described in Using Load Scripts. The supported patterns for wildcards are specified in the ***Linux Manual Page for Glob (7)*** ***http://linux.die.net/man/7/glob***, and for ADO.net platforms, through the .NET ***Directory.getFiles Method*** ***http://msdn.microsoft.com/en-us/library/wz42302f.aspx***.
nodename	[Optional] Specifies the node on which the data to copy resides and the node that should parse the load file. You can use *nodename* to COPY and parse a load file from a node other than the initiator node of the COPY statement. If you omit *nodename*, the location of the input file defaults to the initiator node for the COPY statement. **Note:** You cannot specify *nodename* with either `STDIN` or `LOCAL`, because `STDIN` is read on the initiator node only and `LOCAL` indicates a client node.
`ON ANY NODE`	[Optional] Specifies that the source file to load is on all of the nodes, so COPY opens the file and parses it from any node in the cluster. Make sure that the source file is available and accessible on each cluster node. You can use a wildcard or glob (such as *.dat) to load multiple input files, combined with the ON ANY NODE clause. Using a glob results in COPY distributing the list of files to all cluster nodes and spreading the workload. **Note:** You cannot specify `ON ANY NODE` with either `STDIN` or `LOCAL`, because `STDIN` is read on the initiator node only and `LOCAL` indicates a client node.
`STDIN`	Reads from the client a standard input instead of a file. `STDIN` takes one input source only and is read on the initiator node. To load multiple input sources, use *pathToData*. User must have INSERT privilege on table and USAGE privilege on schema/
`LOCAL`	Specifies that all paths for the COPY statement are on the client system and that all COPY variants are initiated from a

	client. You can use the LOCAL and STDIN parameters together. See Using the COPY and LCOPY Statements in the Administrator's Guide.
BZIP|GZIP|UNCOMPRESSED	Specifies the input file format. The default value is UNCOMPRESSED, and input files can be of any format. If using wildcards, all qualifying input files must be in the same format. **Notes**: ▪ When using concatenated BZIP or GZIP files, be sure that each source file is terminated with a record terminator before concatenating them. ▪ Concatenated BZIP and GZIP files are not supported for NATIVE (binary) and NATIVE VARCHAR formats.
WITH, AS	Improve readability of the statement. These parameters have no effect on the actions performed by the statement.
[WITH [SOURCE *source*(arg='value')] [FILTER *filter*(arg='value')] [PARSER *parser*(arg='value')]]	Specifies COPY to optionally use one or more User Defined Load functions. You can specify up to one source, zero or more filters, and up to one parser.
NATIVE | NATIVE VARCHAR | FIXEDWIDTH	Specifies the parser to use when bulk loading data. By default, COPY uses the DELIMITER parser for UTF-8 format, delimited text input data. Do not specify DELIMITER. COPY always uses the default parser unless you specify another. For more information about using these options, see Loading Different Formats in the Administrator's Guide. **NOTE:** COPY LOCAL does not support the NATIVE and NATIVE VARCHAR parsers
COLUMN OPTION	Specifies load metadata for one or more columns declared in the table column list. For example, you can specify that a column has its own DELIMITER, ENCLOSED BY, NULL as 'NULL' expression, and so on. You do not have to specify every column name explicitly in the COLUMN OPTION list, but each column you specify must correspond to a column in the table column list.
COLSIZES (*integer* [,...])	Required specification when loading fixed-width data using the FIXEDWIDTH parser. COLSIZES and the list of integers must correspond to the columns listed in the table column list. For more information, see Loading Fixed-Width Format Data in the Administrator's Guide.
DELIMITER	A single ASCII character that separates columns within each record of a file. The default in HP Vertica is a vertical bar (|). You can use any ASCII value in the range E'\000' to E'\177' inclusive. You cannot use the same character for both the DELIMITER and NULL options. For more information, see Loading UTF-8 Format Data in the Administrator's Guide.

TRAILING NULLCOLS	Specifies that if HP Vertica encounters a record with insufficient data to match the columns in the table column list, COPY inserts the missing columns with NULLs. For other information and examples, see Loading Fixed-Width Format Data in the Administrator's Guide.
ESCAPE AS	Sets the escape character to indicate that the following character should be interpreted literally, rather than as a special character. You can define an escape character using any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000'). **Note:** The COPY statement does not interpret the data it reads in as ***string literals*** (page 26), and does not follow the same escape rules as other SQL statements (including the COPY parameters). When reading in data, COPY interprets only characters defined by these options as special values: ▪ ESCAPE AS ▪ DELIMITER ▪ ENCLOSED BY ▪ RECORD TERMINATOR
NO ESCAPE	Eliminates escape character handling. Use this option if you do not need any escape character and you want to prevent characters in your data from being interpreted as escape sequences.
ENCLOSED BY	Sets the quote character within which to enclose data, allowing delimiter characters to be embedded in string values. You can choose any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000'). By default, ENCLOSED BY has no value, meaning data is not enclosed by any sort of quote character.
NULL	The string representing a null value. The default is an empty string (''). You can specify a null value as any ASCII value in the range E'\001' to E'\177' inclusive (any ASCII character except NULL: E'\000'). You cannot use the same character for both the DELIMITER and NULL options. For more information, see Loading UTF-8 Format Data.
RECORD TERMINATOR	Specifies the literal character string that indicates the end of a data file record. For more information about using this parameter, see Loading UTF-8 Format Data.
SKIP *records*	Skips a number (integer) of records in a load file. For example, you can use the SKIP option to omit table header information.
SKIP BYTES *total*	Skips the *total* number (integer) of bytes in a record. This option is only available when loading fixed-width data.
TRIM	Trims the number of bytes you specify from a column. This option is only available when loading fixed-width data.

REJECTMAX	Specifies a maximum number of logical records to be rejected before a load fails. For more details about using this option, see Tracking Load Exceptions and Rejections Status in the Administrator's Guide.
EXCEPTIONS *'path'* [ON *nodename*] [, ...]	Specifies the filename or absolute path for the file containing load exceptions. If *path* resolves to a storage location, and the user invoking COPY is not a superuser, these are the required privileges: ▪ The storage location must have been created with the USER option (see ***ADD_LOCATION*** (page 426)) ▪ The user must already have been granted READ access to the storage location where the file(s) exist, as described in ***GRANT (Storage Location)*** (page 839) The optional ON *nodename* clause moves any existing exceptions files on *nodename* to the indicated *path* on the same node. For more details about using this option, see Tracking Load Exceptions and Rejections Status in the Administrator's Guide.
REJECTED DATA *'path'* [ON *nodename*] [, ...]	Specifies the filename or absolute path in which to write rejected rows. If *path* resolves to a storage location, and the user invoking COPY is not a superuser, these are the required privileges: ▪ The storage location must have been created with the USER option (see ***ADD_LOCATION*** (page 426)) ▪ The user must already have been granted READ access to the storage location where the file(s) exist, as described in ***GRANT (Storage Location)*** (page 839) The optional ON *nodename* clause moves any existing rejected data files on *nodename* to *path* on the same node. For more details, see Tracking Load Exceptions and Rejections Status in the Administrator's Guide.
ENFORCELENGTH	Determines whether COPY truncates or rejects data rows of type char, varchar, binary, and varbinary if they do not fit the target table. By default, COPY truncates offending rows of these data types, but does not reject them. For more details, see Tracking Load Exceptions and Rejections Status in the Administrator's Guide.
ABORT ON ERROR	Stops the COPY command if a row is rejected and rolls back the command. No data is loaded.
AUTO | DIRECT | TRICKLE	Specifies the method COPY uses to load data into the database. The default load method is AUTO, in which COPY loads data into the WOS (Write Optimized Store) in memory. When the WOS is full, the load continues directly into ROS (Read Optimized Store) on disk. For more information, see Choosing a Load Method in the Administrator's Guide. **Note:** COPY ignores these options when used as part of a

	CREATE EXTERNAL TABLE statement.
STREAM NAME	[Optional] Supplies a COPY load stream identifier. Using a stream name helps to quickly identify a particular load. The STREAM NAME value that you supply in the load statement appears in the stream column of the LOAD_STREAMS (page 1031) system table. By default, HP Vertica names streams by table and file name. For example, if you have two files (f1, f2) in Table A, their stream names are A-f1, A-f2, respectively. To name a stream: `=> COPY mytable FROM myfile DELIMITER '|' DIRECT STREAM NAME 'My stream name';`
NO COMMIT	Prevents the COPY statement from committing its transaction automatically when it finishes copying data. For more information about using this parameter, see Choosing a Load Method in the Administrator's Guide. **Note:** COPY ignores this option when used as part of a CREATE EXTERNAL TABLE statement.

NOTE: Always use the COPY statement EXCEPTIONS and REJECTED DATA parameters to save load exceptions. Using the RETURNREJECTED parameter is supported only for internal use by the JDBC and ODBC drivers. HP Vertica's internal-use options can change without notice.

COPY Option Summary

The following table summarizes which COPY options are available when loading from delimited text, NATIVE (binary), and NATIVE VARCHAR, and FIXEDWIDTH data:

COPY Option	Delimited Text	NATIVE (BINARY)	NATIVE (VARCHAR)	FIXEDWIDTH
COLUMN OPTION	x	x	x	x
AUTO	x	x	x	x
DIRECT	x	x	x	x
TRICKLE	x	x	x	x
ENFORCELENGTH	x	x	x	x
EXCEPTIONS	x	x	x	x
FILLER	x	x	x	x
REJECTED DATA	x	x	x	x

ABORT ON ERROR	x	x	x	x
STREAM NAME	x	x	x	x
SKIP	x	x	x	x
SKIP BYTES				x
REJECTMAX	x	x	x	x
STDIN	x	x	x	x
UNCOMPRESSED	x	x	x	x
BZIP | GZIP	x	x	x	x
CONCATENATED BZIP or GZIP	x			x
NO COMMIT	x	x	x	x
FORMAT	x	x	x	x
NULL	x	x	x	x
DELIMITED	x			
ENCLOSED BY	x			
ESCAPE AS	x			
TRAILING NULLCOLS	x			
RECORD TERMINATOR	x			x
TRIM				x

Notes

When loading data with the COPY statement, COPY considers the following data invalid:

- 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*** (page 770) command, unless you do not supply a column option as part of the COPY statement.
- Incorrect representation of data type. For example, non-numeric data in an INTEGER column is invalid.

When COPY encounters an empty line during load, it is neither inserted nor rejected, but COPY increments the record number. Keep this fact in mind when you evaluate rejected records lists. If you return a list of rejected records and one empty row was encountered during load, the position of rejected records is incremented by one.

Examples

The following examples load data with the COPY statement using the FORMAT, DELIMITER, NULL and ENCLOSED BY string options, as well as a DIRECT option.

```
 => COPY public.customer_dimension (customer_since FORMAT 'YYYY')
    FROM STDIN
    DELIMITER ','
    NULL AS 'null'
    ENCLOSED BY '"'
=> COPY a
    FROM STDIN
    DELIMITER ','
    NULL E'\\N'
    DIRECT;
=> COPY store.store_dimension
    FROM :input_file
    DELIMITER '|'
    NULL ''
    RECORD TERMINATOR E'\f'
```

Setting vsql Variables

The first two examples load data from STDIN. The last example uses a vsql variable (input_file) . You can set a vsql variable as follows:

```
\set input_file ../myCopyFromLocal/large_table.gzip
```

Using Compressed Data and Named Pipes

COPY supports named pipes that follow the same naming conventions as file names on the given file system. Permissions are open, write, and close.

This command creates the named pipe, *pipe1*, and sets two vsql variables, dir and file:

```
\! mkfifo  pipe1
\set dir `pwd`/
\set file '''':dir'pipe1'''
```

The following sequence copies an uncompressed file from the named pipe:

```
\! cat pf1.dat > pipe1 &
COPY large_tbl FROM :file delimiter '|';
SELECT * FROM large_tbl;
COMMIT;
```

The following statement copies a GZIP file from named pipe and uncompresses it:

```
\! gzip pf1.dat
\! cat pf1.dat.gz > pipe1 &
COPY large_tbl FROM :file ON site01 GZIP delimiter '|';
SELECT * FROM large_tbl;
COMMIT;
\!gunzip pf1.dat.gz
```

The following COPY command copies a BZIP file from named pipe and then uncompresses it:

```
\!bzip2 pf1.dat
\! cat pf1.dat.bz2 > pipe1 &
COPY large_tbl FROM :file ON site01 BZIP delimiter '|';
SELECT * FROM large_tbl;
COMMIT;
bunzip2 pf1.dat.bz2
```

See Also

SQL Data Types (page 71)

ANALYZE_CONSTRAINTS (page 432)

Choosing a Load Method in the Administrator's Guide

CREATE EXTERNAL TABLE AS COPY (page 714)

Directory.getFiles Method http://msdn.microsoft.com/en-us/library/wz42302f.aspx

Bulk Loading Data in the Administrator's Guide

Loading Fixed-Width Format Data in the Administrator's Guide

Loading Binary (Native) Data in the Administrator's Guide

Ignoring Columns and Fields in the Load File in the Administrator's Guide

Linux Manual Page for Glob (7) http://linux.die.net/man/7/glob

Tracking Load Exceptions and Rejections Status in the Administrator's Guide

Transforming Data During Loads in the Administrator's Guide

COPY LOCAL

Using the COPY statement with its `LOCAL` option lets you load a data file on a client system, rather than on a cluster host. COPY LOCAL supports the `STDIN` and `'pathToData'` parameters, but not the `[ON nodename]` clause. COPY LOCAL does not support NATIVE or NATIVE VARCHAR formats.

The COPY LOCAL option is platform independent. The statement works in the same way across all supported HP Vertica platforms and drivers. For more details about using COPY LOCAL with supported drivers, see the Programmer's Guide section for your platform.

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

Invoking COPY LOCAL does not automatically create exceptions and rejections files, even if one or both occur. For information about saving such files, see Capturing Load Exceptions and Rejections in the Administrator's Guide.

Permissions

User must have INSERT privilege on the table and USAGE privilege on the schema.

How Copy Local Works

COPY LOCAL loads data in a platform-neutral way. The COPY LOCAL statement loads all files from a local client system to the HP Vertica host, where the server processes the files. You can copy files in various formats: uncompressed, compressed, fixed-width format, in bzip or gzip format, or specified as a bash glob. Files of a single format (such as all bzip, or gzip) can be comma-separated in the list of input files. You can also use any of the applicable COPY statement options (as long as the data format supports the option). For instance, you can define a specific delimiter character, or how to handle NULLs, and so forth.

NOTE: The Linux `glob` command returns files that match the pattern you enter, as specified in the ***Linux Manual Page for Glob (7) http://linux.die.net/man/7/glob***. For ADO.net platforms, specify patterns and wildcards as described in the .NET ***Directory.getFiles Method http://msdn.microsoft.com/en-us/library/wz42302f.aspx***.

For examples of using the `COPY LOCAL` option to load data, see ***COPY*** (page 699) for syntactical descriptions, and the Bulk Loading Data section in the Administrator's Guide.

The HP Vertica host uncompresses and processes the files as necessary, regardless of file format or the client platform from which you load the files. Once the server has the copied files, HP Vertica maintains performance by distributing file parsing tasks, such as encoding, compressing, uncompressing, across nodes.

Viewing Copy Local Operations in a Query Plan

When you use the `COPY LOCAL` option, the GraphViz Explain plan includes a label for `Load-Client-File`, rather than `Load-File`. Following is a section from a sample Explain plan:

```
-----------------------------------------------
 PLAN:  BASE BULKLOAD PLAN  (GraphViz Format)
-----------------------------------------------
 digraph G {
 graph [rankdir=BT, label = " BASE BULKLOAD PLAN \nAll Nodes Vector: \n\n
node[0]=initiator (initiator) Up\n", labelloc=t, labeljust=l ordering=out]
.
.
.
10[label = "Load-Client-File(/tmp/diff) \nOutBlk=[UncTuple]", color = "green",
shape = "ellipse"];
```

COPY FROM VERTICA

Copies data from another HP Vertica database once you have established a connection to the other HP Vertica database with the ***CONNECT*** (page 697) statement. See Importing Data for more setup information. The COPY FROM VERTICA statement works similarly to the ***COPY*** (page 699) statement, but accepts only a subset of COPY parameters. You can import data from an earlier HP Vertica release, as long as the earlier release is a version of the last major release. For instance, for Version 6.x, you can import data from any version of 5.x, but not from 4.x.

By default, using COPY FROM VERTICA to copy or import data from another database occurs over the HP Vertica private network. Connecting to a public network requires some configuration. For information about using this statement to copy data across a public network, see Importing/Exporting From Public Networks.

Syntax

```
COPY [target_schema.]target_table
... [( target_column_name[, target_column_name2,...])]
... FROM VERTICA database.[source_schema.]source_table
... [(source_column_name[, source_column_name2,...])]
... [AUTO | DIRECT | TRICKLE]
... [STREAM NAME 'stream name']
... [NO COMMIT]
```

Parameters

`[target_schema.]target_table`	The table to store the copied data. This table must be in your local database, and must already exist.
`(target_column_name[, target_column_name2,...])`	A list of columns in the target table to store the copied data. **NOTE:** You cannot use column fillers as part of the column definition.
database	The name of the database that is the source of the copied data. You must have already created a connection to this database in the current session.
`[source_schema.]source_table`	The table in the source database that is the source of the copied data.
`(source_column_name[, source_column_name2,...])`	A list of columns in the source table to be copied. If this list is supplied, only these columns are copied from the source table.
`AUTO | DIRECT | TRICKLE`	Specifies the method COPY uses to load data into the database. The default load method is AUTO, in which COPY loads data into the WOS (Write Optimized Store) in memory. When the WOS is full, the load continues directly into ROS (Read Optimized Store) on disk. For more information, see Choosing a Load Method in the Administrator's Guide. **Note:** COPY ignores these options when used as part of a CREATE EXTERNAL TABLE statement.

STREAM NAME	[Optional] Supplies a COPY load stream identifier. Using a stream name helps to quickly identify a particular load. The STREAM NAME value that you supply in the load statement appears in the stream column of the LOAD_STREAMS (page 1031) system table. By default, HP Vertica names streams by table and file name. For example, if you have two files (f1, f2) in Table A, their stream names are A-f1, A-f2, respectively. To name a stream: => COPY *mytable* FROM *myfile* DELIMITER '|' DIRECT STREAM NAME 'My stream name';
NO COMMIT	Prevents the COPY statement from committing its transaction automatically when it finishes copying data. For more information about using this parameter, see Choosing a Load Method in the Administrator's Guide.

Permissions

- SELECT privileges on the source table
- USAGE privilege on source table schema
- INSERT privileges for the destination table in target database
- USAGE privilege on destination table schema

Notes

- Importing and exporting data fails if either side of the connection is a single-node cluster installed to localhost, or you do not specify a host name or IP address.
- If you do not supply a list of source and destination columns, COPY FROM VERTICA attempts to match columns in the source table with corresponding columns in the destination table. See the following section for details.

Source and Destination Column Mapping

The COPY FROM VERTICA statement needs to map columns in the source table to columns in the destination table. You can optionally supply lists of either source columns to be copied, columns in the destination table where data should be stored, or both. Specifying the lists lets you select a subset of source table columns to copy to the destination table. Since source and destination lists are not required, results differ depending on which list is present. The following table presents the results of supplying one or more lists:

	Omit Source Column List	Supply Source Column List

Omit Destination Column List	Matches all columns in the source table to columns in the destination table. The number of columns in the two tables need not match, but the destination table must not have fewer columns than the source.	Copies content only from the supplied list of source table columns. Matches columns in the destination table to columns in the source list. The number of columns in the two tables need not match, but the destination table must not have fewer columns than the source.
Supply Destination Column List	Matches columns in the destination column list to columns in the source. The number of columns in the destination list must match the number of columns in the source table.	Matches columns from the source table column lists to those in the destination table. The lists must have the same number of columns.

Example

This example demonstrates connecting to another database, copying the contents of an entire table from the source database to an identically-defined table in the current database directly into ROS, and then closing the connection.

```
=> CONNECT TO VERTICA vmart USER dbadmin PASSWORD '' ON 'VertTest01',5433;
CONNECT
=> COPY customer_dimension FROM  VERTICA vmart.customer_dimension DIRECT;
 Rows Loaded
-------------
      500000
(1 row)
=> DISCONNECT vmart;
DISCONNECT
```

This example demonstrates copying several columns from a table in the source database into a table in the local database.

```
=> CONNECT TO VERTICA vmart USER dbadmin PASSWORD '' ON 'VertTest01',5433;
CONNECT
=> COPY people (name, gender, age) FROM VERTICA
-> vmart.customer_dimension (customer_name, customer_gender,
-> customer_age);
 Rows Loaded
-------------
      500000
(1 row)
=> DISCONNECT vmart;
DISCONNECT
```

You can copy tables (or columns) containing Identity and Auto-increment values, but the sequence values are not incremented automatically at their destination.

See Also

CONNECT (page 697)

DISCONNECT (page 809)

EXPORT TO VERTICA (page 829)

CREATE EXTERNAL TABLE AS COPY

Creates an external table. This statement is a combination of the ***CREATE TABLE*** (page 770) and ***COPY*** (page 699) statements, supporting a subset of each statement's parameters, as noted below. You can also use user-defined load extension functions (UDLs) to create external tables. For more information about UDL syntax, see User Defined Load (UDL) and ***COPY*** (page 699).

Note: HP Vertica does not create a superprojection for an external table when you create it.

Permissions

Must be a database superuser to create external tables, unless the superuser has created a user-accessible storage location to which the COPY refers, as described in ***ADD_LOCATION*** (page 426). Once external tables exist, you must also be a database superuser to access them through a select statement.

NOTE: Permission requirements for external tables differ from other tables. To gain full access (including SELECT) to an external table that a user has privileges to create, the database superuser must also grant READ access to the USER-accessible storage location, see ***GRANT (Storage Location)*** (page 839).

Syntax

```
CREATE EXTERNAL TABLE [ IF NOT EXISTS ] [schema.]table-name
{
... ( column-definition (table) (page 779)  [ , ... ] )
... | [ column-name-list  (create table)  (page 780) ]
} AS COPY [ [db-name.]schema-name.]table
... [ ( { column-as-expression | column }
...... [ FILLER datatype ]
...... [ FORMAT 'format' ]
...... [ ENCLOSED BY 'char' ]
...... [ ESCAPE AS 'char' | NO ESCAPE ]
...... [ NULL [ AS ] 'string' ]
...... [ TRIM 'byte' ]
...... [ DELIMITER [ AS ] 'char' ]
... [, ... ] ) ]
... [ COLUMN OPTION ( column
...... [ FORMAT 'format' ]
...... [ ENCLOSED BY 'char' ]
...... [ ESCAPE AS 'char' | NO ESCAPE ]
...... [ NULL [ AS ] 'string' ]
...... [ DELIMITER [ AS ] 'char' ]
... [, ... ] ) ]
FROM {
...| 'pathToData' [ ON nodename ]
...... [ BZIP | GZIP | UNCOMPRESSED ] [, ...]
}
...[ NATIVE
...| NATIVE VARCHAR
...| FIXEDWIDTH { COLSIZES (integer [, ....]) }
...]
```

```
...[ WITH ]
...[ DELIMITER [ AS ] 'char' ]
...[ TRAILING NULLCOLS ]
...[ NULL [ AS ] 'string' ]
...[ ESCAPE AS 'char' | NO ESCAPE ]
...[ ENCLOSED BY 'char' [ AND 'char' ] ]
...[ RECORD TERMINATOR 'string' ]
...[ SKIP integer ]
...[ SKIP BYTES integer ]
...[ TRIM 'byte' ]
...[ REJECTMAX integer ]
...[ EXCEPTIONS 'path' [ ON nodename ] [, ...] ]
...[ REJECTED DATA 'path' [ ON nodename ] [, ...] ]
...[ ENFORCELENGTH ]
...[ ABORT ON ERROR ]
```

Parameters

The following parameters from the parent statements are not supported in the `CREATE EXTERNAL TABLE AS COPY` statement:

CREATE TABLE	COPY
AS AT EPOCH LAST	FROM STDIN
AT TIME 'timestamp'	FROM LOCAL
ORDER BY table-column [,...]	DIRECT
ENCODED BY	TRICKLE
hash-segmentation-clause	NO COMMIT
range-segmentation-clause	STREAM NAME
UNSEGMENTED {node | node all}	
KSAFE [k_num]	
PARTITION BY partition-clause	

For all supported parameters, see the ***CREATE TABLE*** (page 770) and ***COPY*** (page 699) statements.

Notes

Canceling a CREATE EXTERNAL TABLE AS COPY statement can cause unpredictable results. HP recommends that you allow the statement to finish, then use ***DROP TABLE*** (page 823) once the table exists.

Examples

Examples of external table definitions:

```
CREATE EXTERNAL TABLE ext1 (x integer) AS COPY FROM '/tmp/ext1.dat' DELIMITER ',';
CREATE EXTERNAL TABLE ext1 (x integer) AS COPY FROM '/tmp/ext1.dat.bz2' BZIP
DELIMITER ',';
```

CREATE EXTERNAL TABLE ext1 (x integer, y integer) AS COPY (x as '5', y) FROM '/tmp/ext1.dat.bz2' BZIP DELIMITER ',';

See Also

Physical Schema in the Concepts Guide

COPY (page 714)

CREATE TABLE (page 770)

SELECT (page 870)

Using External Tables in the Administrator's Guide

CREATE FUNCTION Statements

You can use the Create Function statement to create two different kinds of functions:

- User-Defined SQL functions--User defined SQL functions let you define and store commonly-used SQL expressions as a function. User defined SQL functions are useful for executing complex queries and combining HP Vertica built-in functions. You simply call the function name you assigned in your query.

- User-Defined Scalar functions--User defined scalar functions (UDSFs) take in a single row of data and return a single value. These functions can be used anywhere a native HP Vertica function or statement can be used, except CREATE TABLE with its PARTITION BY or any segmentation clause.

While you use CREATE FUNCTION to create both SQL and scalar functions, you use a different syntax for each function type. For more information, see:

- ***CREATE FUNCTION (SQL Functions)*** (page 722)
- ***CREATE FUNCTION (UDF)*** (page 725)

About Creating User Defined Transform Functions (UDTFs)

You can use a similar SQL statement to create user-defined transform functions. User Defined Transform Functions (UDTFs) operate on table segments and return zero or more rows of data. The data they return can be an entirely new table, unrelated to the schema of the input table, including having its own ordering and segmentation expressions. They can only be used in the SELECT list of a query. For details see Using User Defined Transforms. To create a UDTF, see ***CREATE TRANSFORM FUNCTION*** (page 734).

CREATE AGGREGATE FUNCTION

Adds a User Defined Aggregate Function (UDAF) stored in a shared Linux library to the catalog. You must have already loaded this library using the ***CREATE LIBRARY*** (page 735) statement. When you call the SQL function, HP Vertica passes data values to the code in the library to process it.

Syntax

```
CREATE [ OR REPLACE ] AGGREGATE FUNCTION [[db-name.]schema.]function-name
```

```
... AS LANGUAGE 'language' NAME 'factory' LIBRARY library_name;
```

Parameters

`[ OR REPLACE ]`	If you do not supply this parameter, the CREATE AGGREGATE FUNCTION statement fails if an existing function matches the name and parameters of the function you are trying to define. If you do supply this parameter, the new function definition overwrites the old.
[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You must be connected to the database you specify. You cannot make changes to objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (mydb.myschema).
function-name	The name of the function to create. If the function name is schema-qualified (as above), the function is created in the specified schema. This name does not need to match the name of the factory, but it is less confusing if they are the same or similar.
`LANGUAGE '`*language*`'`	The programming language used to develop the function. Currently only `'C++'` is supported for UDAF.
`NAME '`*factory*`'`	The name of the factory class in the shared library that generates the object to handle the function's processing.
`LIBRARY` *library_name*	The name of the shared library that contains the C++ object to perform the processing for this function. This library must have been previously loaded using the ***CREATE LIBRARY*** (page 735) statement.

Notes

- The parameters and return value for the function are automatically determined by the CREATE AGGREGATE FUNCTION statement, based on data supplied by the factory class.
- When a User Defined Aggregate function that is defined multiple times with arguments of different data types is called, HP Vertica selects the function whose input parameters match the parameters in the function call to perform the processing.
- You can return a list of all SQL functions and User Defined Functions (including aggregates) by querying the system table `V_CATALOG.USER_FUNCTIONS` (page 982) or executing the vsql meta-command `\df`. Users see only the functions on which they have `EXECUTE` privileges.

Permissions

- Only a superuser can create or drop a User Defined Aggregate library.
- To create a User Defined Aggregate function, the user must have CREATE and USAGE privileges on the schema and USAGE privileges on the library.

- To use a User Defined Aggregate, the user must have USAGE privileges on the schema and EXECUTE privileges on the defined function. See ***GRANT (Function)*** (page 843) and ***REVOKE (Function)*** (page 864).

Example

The following example demonstrates loading a library named AggregateFunctions then defining a function named ag_avg and ag_cat that are mapped to the ag_cat AverageFactory and ConcatenateFactory classes in the library:

```
=> CREATE LIBRARY AggregateFunctions AS
'/opt/vertica/sdk/examples/build/AggregateFunctions.so';
CREATE LIBRARY

=> create aggregate function ag_avg as LANGUAGE 'C++' name 'AverageFactory' library
AggregateFunctions;
CREATE AGGREGATE FUNCTION

=> create aggregate function ag_cat as LANGUAGE 'C++' name 'ConcatenateFactory'
library AggregateFunctions;
CREATE AGGREGATE FUNCTION

=> \x
Expanded display is on.
select * from user_functions;
-[ RECORD 1
]----------+-----------------------------------------------------------------
schema_name            | public
function_name          | ag_avg
procedure_type         | User Defined Aggregate
function_return_type   | Numeric
function_argument_type | Numeric
function_definition    | Class 'AverageFactory' in Library
'public.AggregateFunctions'
volatility             |
is_strict              | f
is_fenced              | f
comment                |
-[ RECORD 2
]----------+-----------------------------------------------------------------
schema_name            | public
function_name          | ag_cat
procedure_type         | User Defined Aggregate
function_return_type   | Varchar
function_argument_type | Varchar
function_definition    | Class 'ConcatenateFactory' in Library
'public.AggregateFunctions'
volatility             |
is_strict              | f
is_fenced              | f
comment                |
```

See Also

CREATE LIBRARY (page 735)

DROP AGGREGATE FUNCTION

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Developing and Using User Defined Functions and Developing a User Defined Aggregate Function in the Programmer's Guide

CREATE ANALYTIC FUNCTION

Associates a User Defined Analytic Function (UDAnF) stored in a shared Linux library with a SQL function name. You must have already loaded the library containing the UDAnF using the ***CREATE LIBRARY*** (page 735) statement. When you call the SQL function, HP Vertica passes the arguments to the analytic function in the library to process.

Syntax

```
CREATE [ OR REPLACE ] ANALYTIC FUNCTION function-name
... AS [ LANGUAGE 'language' ] NAME 'factory'
... LIBRARY library_name
... [ FENCED | NOT FENCED ];
```

Parameters

function-name	The name to assign to the UDAnF. This is the name you use in your SQL statements to call the function.
LANGUAGE '*language*'	The programming language used to write the UDAnF. Currently, 'C++' is supported. If not supplied, C++ is assumed.
NAME '*factory*'	The name of the C++ factory class in the shared library that generates the object to handle the function's processing.
LIBRARY *library_name*	The name of the shared library that contains the C++ object to perform the processing for this function. This library must have been previously loaded using the ***CREATE LIBRARY*** (page 735) statement.
[FENCED | NOT FENCED]	Enables or disables Fenced Mode for this function. Fenced mode is enabled by default.

Permissions

- To CREATE a function, the user must have CREATE privilege on the schema to contain the function and USAGE privilege on the library containing the function.
- To use a function, the user must have USAGE privilege on the schema that contains the function and EXECUTE privileges on the function.
- To DROP a function, the user must either be a superuser, the owner of the function, or the owner of the schema which contains the function.

Notes

- The parameters and return value for the function are automatically determined by the CREATE ANALYTIC FUNCTION statement, based on data supplied by the factory class.
- You can assign multiple functions the same name if they accept different sets of arguments. See User Defined Function Overloading in the Programmer's Guide for more information.
- You can return a list of all UDFs by querying the system table ***V_CATALOG.USER_FUNCTIONS***. Users see only the functions on which they have `EXECUTE` privileges.

See Also

- Developing a User Defined Analytic Function in the Programmer's Guide.

CREATE FILTER

Adds a User Defined Load FILTER function. You must have already loaded this library using the ***CREATE LIBRARY*** (page 735) statement. When you call the SQL function, HP Vertica passes the parameters to the function in the library to process it.

Syntax

```
CREATE [ OR REPLACE ] FILTER [[db-name.]schema.]function-name
... AS LANGUAGE 'language' NAME 'factory' LIBRARY library_name
```

Parameters

`[ OR REPLACE ]`	If you do not supply this parameter, the CREATE FILTER statement fails if an existing function matches the name and parameters of the filter function you are trying to define. If you do supply this parameter, the new filter function definition overwrites the old.
[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You must be connected to the database you specify. You cannot make changes to objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you

	can use a database and a schema name (mydb.myschema).
function-name	The name of the filter function to create. If the filter function name is schema-qualified (as above), the function is created in the specified schema. This name does not need to match the name of the factory, but it is less confusing if they are the same or similar.
LANGUAGE '*language*'	The programming language used to develop the function. 'C++' is the only language supported by User Defined Load functions.
NAME '*factory*'	The name of the factory class in the shared library that generates the object to handle the filter function's processing. This is the same name used by the RegisterFactory class.
LIBRARY *library_name*	The name of the shared library that contains the C++ object to perform the processing for this filter function. This library must have been previously loaded using the ***CREATE LIBRARY*** (page 735) statement.

Notes

- The parameters and return value for the filter function are automatically determined by the CREATE FILTER statement, based on data supplied by the factory class.
- You can return a list of all SQL functions and User Defined Functions by querying the system table V_CATALOG.USER_FUNCTIONS (page 982) or executing the vsql meta-command \df. Users see only the functions on which they have EXECUTE privileges.

Permissions

- Only a superuser can create or drop a function that uses a UDx library.
- To use a User Defined Filter, the user must have USAGE privileges on the schema and EXECUTE privileges on the defined filter function. See ***GRANT (Function)*** (page 843) and ***REVOKE (Function)*** (page 864).

IMPORTANT! Installing an untrusted UDL function can compromise the security of the server. UDx's can contain arbitrary code. In particular, UD Source functions can read data from any arbitrary location. It is up to the developer of the function to enforce proper security limitations. Superusers must not grant access to UDx's to untrusted users.

Example

The following example demonstrates loading a library named iConverterLib, then defining a function named Iconverter that is mapped to the iConverterFactory factory class in the library:

```
=> CREATE LIBRARY iConverterLib as
'/opt/vertica/sdk/examples/build/IconverterLib.so';
CREATE LIBRARY

=> CREATE FILTER Iconverter AS LANGUAGE 'C++' NAME 'IconverterFactory' LIBRARY
IconverterLib;
CREATE FILTER FUNCTION
```

```
=> \x
Expanded display is on.
=> SELECT * FROM user_functions;
-[ RECORD 1 ]----------+--------------------
schema_name            | public
function_name          | Iconverter
procedure_type         | User Defined Filter
function_return_type   |
function_argument_type |
function_definition    |
volatility             |
is_strict              | f
is_fenced              | f
comment                |
```

See Also

CREATE LIBRARY (page 735)

DROP FILTER

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Developing User Defined Load (UDL) Functions in the Programmer's Guide

CREATE FUNCTION (SQL Functions)

Lets you store SQL expressions as functions in HP Vertica for use in queries. These functions are useful for executing complex queries or combining HP Vertica built-in functions. You simply call the function name you assigned.

This topic describes how to use CREATE FUNCTION to create a SQL function. If you want to create a user-defined scalar function (UDSF), see ***CREATE FUNCTION (UDF)*** (page 725).

In addition, if you want to see how to create a user-defined transform function (UDTF), see ***CREATE TRANSFORM FUNCTION.*** (page 734)

Syntax

```
CREATE [ OR REPLACE ] FUNCTION
... [[db-name.]schema.]function-name ( [ argname argtype  [, ...] ] )
```

```
... RETURN rettype
... AS
... BEGIN
...... RETURN expression;
... END;
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
function-name	Specifies a name for the SQL function to create. When using more than one schema, specify the schema that contains the function, as noted above.
argname	Specifies the name of the argument.
argtype	Specifies the data type for argument that is passed to the function. Argument types must match HP Vertica type names. See ***SQL Data Types*** (page 71).
rettype	Specifies the data type to be returned by the function.
`RETURN` *expression;*	Specifies the SQL function (function body), which must be in the form of 'RETURN expression.' expression can contain built-in functions, operators, and argument names specified in the `CREATE FUNCTION` statement. A semicolon at the end of the expression is required. **Note:** Only one `RETURN` expression is allowed in the `CREATE FUNCTION` definition. `FROM`, `WHERE`, `GROUP BY`, `ORDER BY`, `LIMIT`, aggregation, analytics and meta function are not allowed.

Permissions

- To CREATE a function, the user must have CREATE privilege on the schema to contain the function and USAGE privilege on the library containing the function.
- To use a function, the user must have USAGE privilege on the schema that contains the function and EXECUTE privileges on the function.
- To DROP a function, the user must either be a superuser, the owner of the function, or the owner of the schema which contains the function.

See ***GRANT (Function)*** (page 843) and ***REVOKE (Function)*** (page 864).

Notes

- A SQL function can be used anywhere in a query where an ordinary SQL expression can be used, except in the table partition clause or the projection segmentation clause.
- SQL Macros are flattened in all cases, including DDL.
- You can ***create views*** (page 804) on the queries that use SQL functions and then query the views. When you create a view, a SQL function replaces a call to the user-defined function with the function body in a view definition. Therefore, when the body of the user-defined function is replaced, the view should also be replaced.
- If you want to change the body of a SQL function, use the `CREATE OR REPLACE` syntax. The command replaces the function with the new definition. If you change only the argument name or argument type, the system maintains both versions under the same function name. See **Examples** section below.
- If multiple SQL functions with same name and argument type are in the search path, the first match is used when the function is called.
- The strictness and volatility (stable, immutable, or volatile) of a SQL Macro are automatically inferred from the function's definition. HP Vertica then performs constant folding optimization, when possible, and determines the correctness of usage, such as where an immutable function is expected but a volatile function is provided.
- You can return a list of all SQL functions by querying the system table `V_CATALOG.USER_FUNCTIONS` (page 982) and executing the vsql meta-command `\df`. Users see only the functions on which they have `EXECUTE` privileges.

Example

This following statement creates a SQL function called `myzeroifnull` that accepts an `INTEGER` argument and returns an `INTEGER` result.

```
=> CREATE FUNCTION myzeroifnull(x INT) RETURN INT
   AS BEGIN
     RETURN (CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END);
   END;
```

You can use the new SQL function (`myzeroifnull`) anywhere you use an ordinary SQL expression. For example, create a simple table:

```
=> CREATE TABLE tabwnulls(col1 INT);
=> INSERT INTO tabwnulls VALUES(1);
=> INSERT INTO tabwnulls VALUES(NULL);
=> INSERT INTO tabwnulls VALUES(0);

=> SELECT * FROM tabwnulls;
 a
---
 1

 0
(3 rows)
```

Use the `myzeroifnull` function in a `SELECT` statement, where the function calls `col1` from table tabwnulls:

```
=> SELECT myzeroifnull(col1) FROM tabwnulls;
 myzeroifnull
--------------
            1
            0
            0
(3 rows)
```

Use the `myzeroifnull` function in the `GROUP BY` clause:

```
=> SELECT COUNT(*) FROM tabwnulls GROUP BY myzeroifnull(col1);
 count
-------
     2
     1
(2 rows)
```

If you want to change a SQL function's body, use the `CREATE OR REPLACE` syntax. The following command modifies the `CASE` expression:

```
=> CREATE OR REPLACE FUNCTION zerowhennull(x INT) RETURN INT
   AS BEGIN
     RETURN (CASE WHEN (x IS NULL) THEN 0 ELSE x END);
   END;
```

To see how this information is stored in the HP Vertica catalog, see Viewing Information About SQL Functions in the <SQL_PROGRAMMERS_GUIDE>.

See Also

ALTER FUNCTION (page 656)

DROP FUNCTION (page 811)

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Using SQL Macros in the Programmer's Guide

CREATE FUNCTION (UDF)

Adds a User Defined Function (UDF) to the catalog. You must have already loaded this library using the ***CREATE LIBRARY*** (page 735) statement. When you call the SQL function, HP Vertica passes the parameters to the function in the library to process it.

This topic describes how to use CREATE FUNCTION to create a User Defined Function. If you want to create a SQL function, see ***CREATE FUNCTION (SQL Function)*** (page 722).

In addition, if you want to see how to create a user-defined transform function (UDTF), see ***CREATE TRANSFORM FUNCTION.*** (page 734)

Syntax

```
CREATE [ OR REPLACE ] FUNCTION [[db-name.]schema.]function-name
... AS LANGUAGE 'language' NAME 'factory' LIBRARY library_name
... [ IMMUTABLE | STABLE | VOLATILE ]
... [ CALLED ON NULL INPUT | RETURN NULL ON NULL INPUT | STRICT ]
... [ FENCED | NOT FENCED ];
```

Parameters

`[ OR REPLACE ]`	If you do not supply this parameter, the CREATE FUNCTION statement fails if an existing function matches the name and parameters of the function you are trying to define. If you do supply this parameter, the new function definition overwrites the old.
[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
function-name	The name of the function to create. If the function name is schema-qualified (as above), the function is created in the specified schema. This name does not need to match the name of the factory, but it is less confusing if they are the same or similar.
`LANGUAGE '`*language*`'`	The programming language used to develop the function. `'C++'` and `'R'` is supported.
`NAME '`*factory*`'`	The name of the factory class in the shared library that generates the object to handle the function's processing.
`LIBRARY` *library_name*	The name of the shared library/R functions that contains the C++ object or R functions to perform the processing for this function. This library must have been previously loaded using the ***CREATE LIBRARY*** (page 735) statement.

`[ IMMUTABLE | STABLE | VOLATILE ]`	Sets the volatility of the function: ▪ IMMUTABLE means that repeated calls to the function with the same input always returns the same output. ▪ STABLE means that repeated calls to the function with the same input *within the same statement* returns the same output. For example, a function that returns the current user name would be stable since the user cannot change within a statement, but could change between statements. ▪ VOLATILE means that repeated calls to the function with the same input can result in different output. If not supplied, HP Vertica assumes that the function is VOLATILE, and needs to be called for each invocation. Setting the volatility of the function helps HP Vertica optimize expressions. For example, if a function is IMMUTABLE, its results can be cached. **Caution:** specifying IMMUTABLE or STABLE when the function is actually VOLATILE can result in incorrect or inconsistent answers. **Note:** This parameter is deprecated in HP Vertica Version 6.0 and will be removed in a future release. Instead, your UDSF should declare its own volatility using the SDK API calls. For details, see Setting Null Input and Volatility Behavior in the Programmer's Guide.
`[ CALLED ON NULL INPUT | RETURN NULL ON NULL INPUT | STRICT ]`	Sets the null behavior of the function: ▪ CALLED ON NULL INPUT means that even if all parameters passed to the function are NULL, the function can return a non-null value. ▪ RETURN NULL ON NULL INPUT and STRICT mean that the function returns NULL if any of its input is NULL. If not specified, HP Vertica assumes that the function is CALLED ON NULL INPUT and must be called even if its input values are NULL. Setting the NULL behavior of a function allows HP Vertica to optimize expressions. For example, if the function is set to STRICT, it does not need to be called if one of the input parameters is NULL. **Note:** This parameter is deprecated in HP Vertica Version 6.0 and will be removed in a future release. Instead, your UDSF should declare its own strictness. For details, see Setting Null Input and Volatility Behavior in the Programmer's Guide.
`[ FENCED | NOT FENCED ]`	Enables or disables Fenced Mode for this function. Fenced mode is enabled by default. Functions written in R always run in fenced mode.

Permissions

- To CREATE a function, the user must have CREATE privilege on the schema to contain the function and USAGE privilege on the library containing the function.
- To use a function, the user must have USAGE privilege on the schema that contains the function and EXECUTE privileges on the function.
- To DROP a function, the user must either be a superuser, the owner of the function, or the owner of the schema which contains the function.

Notes

- The parameters and return value for the function are automatically determined by the CREATE FUNCTION statement, based on data supplied by the factory class.
- Multiple functions can share the same name if they have different parameters. When you call a multiply-defined function, HP Vertica selects the UDF function whose input parameters match the parameters in the function call to perform the processing. This behavior is similar to having multiple signatures for a method or function in other programming languages.
- You can return a list of all SQL functions and UDFs by querying the system table `V_CATALOG.USER_FUNCTIONS` (page 982) or executing the vsql meta-command `\df`. Users see only the functions on which they have `EXECUTE` privileges.

Example

The following example demonstrates loading a library named scalarfunctions, then defining a function named Add2ints that is mapped to the Add2intsInfo factory class in the library:

```
=> CREATE LIBRARY ScalarFunctions AS
'/opt/vertica/sdk/examples/build/ScalarFunctions.so';
CREATE LIBRARY
=> CREATE FUNCTION Add2Ints AS LANGUAGE 'C++' NAME 'Add2IntsFactory' LIBRARY
ScalarFunctions;
CREATE FUNCTION
=> \x
Expanded display is on.
=> SELECT * FROM USER_FUNCTIONS;
-[ RECORD 1 ]----------+-----------------------------------------------------
schema_name            | public
function_name          | Add2Ints
procedure_type         | User Defined Function
function_return_type   | Integer
function_argument_type | Integer, Integer
function_definition    | Class 'Add2IntsFactory' in Library
'public.ScalarFunctions'
volatility             | volatile
is_strict              | f
is_fenced              | t
comment                |

=> \x
Expanded display is off.
=> -- Try a simple call to the function
```

```
=> SELECT Add2Ints(23,19);
 Add2Ints
----------
       42
(1 row)
```

The Add2Ints function adds two numbers together. Adding always returns the same output for a specific input, so its results are immutable. The following example shows recreating the function with the volatility set to IMMUTABLE.

```
=> CREATE FUNCTION Add2Ints AS LANGUAGE 'C++' NAME 'Add2IntsFactory' LIBRARY
ScalarFunctions IMMUTABLE;
ROLLBACK:  Function with same name and number of parameters already exists:
Add2ints

=> -- Oops. Need to replace the old function.

=>  CREATE OR REPLACE FUNCTION Add2ints AS LANGUAGE 'C++' NAME 'Add2IntsFactory'
LIBRARY ScalarFunctions IMMUTABLE;
CREATE FUNCTION
```

Note: Using the IMMUTABLE parameter in CREATE FUNCTION is deprecated in HP Vertica Version 6.0 and will be removed in a future release. Instead, your UDSF should declare its own volatility. For details, see Setting Null Input and Volatility Behavior in the Programmer's Guide.

See Also

- ***CREATE LIBRARY*** (page 735)
- ***DROP FUNCTION*** (page 811)
- ***GRANT (Function)*** (page 843)
- ***REVOKE (Function)*** (page 864)
- ***V_CATALOG.USER_FUNCTIONS*** (page 982)
- Developing and Using User Defined Functions in the Programmer's Guide

CREATE PARSER

Adds a User Defined Load PARSER function. You must have already loaded this library using the ***CREATE LIBRARY*** (page 735) statement. When you call the SQL function, HP Vertica passes the parameters to the function in the library to process it.

Syntax

```
CREATE [ OR REPLACE ] PARSER [[db-name.]schema.]function-name
... AS LANGUAGE 'language' NAME 'factory' LIBRARY library_name
```

Parameters

`[ OR REPLACE ]`	If you do not supply this parameter, the CREATE PARSER statement fails if an existing function matches the name and parameters of the parser function you are trying to define. If you do supply this parameter, the new parser function

	definition overwrites the old.
[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You must be connected to the database you specify. You cannot make changes to objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (mydb.myschema).
function-name	The name of the parser function to create. If the parser function name is schema-qualified (as above), the function is created in the specified schema. This name does not need to match the name of the factory, but it is less confusing if they are the same or similar.
LANGUAGE '*language*'	The programming language used to develop the function. 'C++' is the only language supported by User Defined Load functions.
NAME '*factory*'	The name of the factory class in the shared library that generates the object to handle the parser function's processing. This is the same name used by the RegisterFactory class.
LIBRARY *library_name*	The name of the shared library that contains the C++ object to perform the processing for this parser function. This library must have been previously loaded using the ***CREATE LIBRARY*** (page 735) statement.

Notes

- The parameters and return value for the parser function are automatically determined by the CREATE PARSER statement, based on data supplied by the factory class.
- You can return a list of all SQL functions and User Defined Functions by querying the system table V_CATALOG.USER_FUNCTIONS (page 982) or executing the vsql meta-command \df. Users see only the functions on which they have EXECUTE privileges.

Permissions

- Only a superuser can create or drop a function that uses a UDx library.
- To use a User Defined Parser, the user must have USAGE privileges on the schema and EXECUTE privileges on the defined parser function. See ***GRANT (Function)*** (page 843) and ***REVOKE (Function)*** (page 864).

IMPORTANT! Installing an untrusted UDL function can compromise the security of the server. UDx's can contain arbitrary code. In particular, UD Source functions can read data from any arbitrary location. It is up to the developer of the function to enforce proper security limitations. Superusers must not grant access to UDx's to untrusted users.

Example

The following example demonstrates loading a library named BasicIntegrerParserLib, then defining a function named BasicIntegerParser that is mapped to the BasicIntegerParserFactory factory class in the library:

```
=> CREATE LIBRARY BasicIntegerParserLib as
'/opt/vertica/sdk/examples/build/BasicIntegerParser.so';
CREATE LIBRARY

=> CREATE PARSER BasicIntegerParser AS LANGUAGE 'C++' NAME
'BasicIntegerParserFactory' LIBRARY BasicIntegerParserLib;
CREATE PARSER FUNCTION


=> \x
Expanded display is on.
=> SELECT * FROM user_functions;
-[ RECORD 1 ]----------+--------------------
schema_name            | public
function_name          | BasicIntegerParser
procedure_type         | User Defined Parser
function_return_type   |
function_argument_type |
function_definition    |
volatility             |
is_strict              | f
is_fenced              | f
comment                |
```

See Also

CREATE LIBRARY (page 735)

DROP PARSER

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Developing User Defined Load (UDL) Functions in the Programmer's Guide

CREATE SOURCE

Adds a User Defined Load SOURCE function. You must have already loaded this library using the ***CREATE LIBRARY*** (page 735) statement. When you call the SQL function, HP Vertica passes the parameters to the function in the library to process it.

Syntax

```
CREATE [ OR REPLACE ] SOURCE [[db-name.]schema.]function-name
... AS LANGUAGE 'language' NAME 'factory' LIBRARY library_name
```

Parameters

`[ OR REPLACE ]`	If you do not supply this parameter, the CREATE SOURCE statement fails if an existing function matches the name and parameters of the source function you are trying to define. If you do supply this parameter, the new source function definition overwrites the old.
[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You must be connected to the database you specify. You cannot make changes to objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (mydb.myschema).
function-name	The name of the source function to create. If the source function name is schema-qualified (as above), the function is created in the specified schema. This name does not need to match the name of the factory, but it is less confusing if they are the same or similar.
`LANGUAGE '`*language*`'`	The programming language used to develop the function. `'C++'` is the only language supported by User Defined Load functions.
`NAME '`*factory*`'`	The name of the factory class in the shared library that generates the object to handle the source function's processing. This is the same name used by the RegisterFactory class.
`LIBRARY` *library_name*	The name of the shared library that contains the C++ object to perform the processing for this source function. This library must have been previously loaded using the ***CREATE LIBRARY*** (page 735) statement.

Notes

- The parameters and return value for the source function are automatically determined by the CREATE SOURCE statement, based on data supplied by the factory class.
- You can return a list of all SQL functions and User Defined Functions by querying the system table `V_CATALOG.USER_FUNCTIONS` (page 982) or executing the vsql meta-command `\df`. Users see only the functions on which they have `EXECUTE` privileges.

Permissions

- Only a superuser can create or drop a function that uses a UDx library.

- To use a User Defined Source, the user must have USAGE privileges on the schema and EXECUTE privileges on the defined source function. See ***GRANT (Function)*** (page 843) and ***REVOKE (Function)*** (page 864).

IMPORTANT! Installing an untrusted UDL function can compromise the security of the server. UDx's can contain arbitrary code. In particular, UD Source functions can read data from any arbitrary location. It is up to the developer of the function to enforce proper security limitations. Superusers must not grant access to UDx's to untrusted users.

Example

The following example demonstrates loading a library named curllib, then defining a function named curl that is mapped to the CurlSourceFactory factory class in the library:

```
=> CREATE LIBRARY curllib as '/opt/vertica/sdk/examples/build/cURLLib.so';
CREATE LIBRARY

=> CREATE SOURCE curl AS LANGUAGE 'C++' NAME 'CurlSourceFactory' LIBRARY curllib;
CREATE SOURCE

=> \x
Expanded display is on.
=> SELECT * FROM user_functions;
-[ RECORD 1 ]----------+--------------------
schema_name            | public
function_name          | curl
procedure_type         | User Defined Source
function_return_type   |
function_argument_type |
function_definition    |
volatility             |
is_strict              | f
is_fenced              | f
comment                |
```

See Also

CREATE LIBRARY (page 735)

DROP SOURCE

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Developing User Defined Load (UDL) Functions in the Programmer's Guide

CREATE TRANSFORM FUNCTION

Adds a User Defined Transform Function (UDTF) stored in a shared Linux library to the catalog. You must have already loaded this library using the ***CREATE LIBRARY*** (page 735) statement. When you call the SQL function, HP Vertica passes the input table to the transform function in the library to process.

This topic describes how to create a UDTF. If you want to create a user-defined function (UDF), see ***CREATE FUNCTION (UDF)*** (page 725). If you want to create a SQL function, see ***CREATE FUNCTION (SQL)*** (page 722).

Syntax

```
CREATE TRANSFORM FUNCTION function-name
... [ AS LANGUAGE 'language' ] NAME 'factory'
... LIBRARY library_name
... [ FENCED | NOT FENCED ];
```

Parameters

function-name	The name to assign to the UDTF. This is the name you use in your SQL statements to call the function.
LANGUAGE '*language*'	The programming language used to write the UDTF. Currently, 'C++' and 'R' is supported. If not supplied, C++ is assumed.
NAME '*factory*'	The name of the C++ factory class or R factory function in the shared library that generates the object to handle the function's processing.
LIBRARY *library_name*	The name of the shared library that contains the C++ object to perform the processing for this function. This library must have been previously loaded using the ***CREATE LIBRARY*** (page 735) statement.
[FENCED | NOT FENCED]	Enables or disables Fenced Mode for this function. Fenced mode is enabled by default. Functions written in R always run in fenced mode.

Permissions

- To CREATE a function, the user must have CREATE privilege on the schema to contain the function and USAGE privilege on the library containing the function.
- To use a function, the user must have USAGE privilege on the schema that contains the function and EXECUTE privileges on the function.
- To DROP a function, the user must either be a superuser, the owner of the function, or the owner of the schema which contains the function.

See GRANT (Transform Function) and REVOKE (Transform Function).

UDTF Query Restrictions

A query that includes a UDTF cannot contain:

- any statements other than the ***SELECT*** (page 870) statement containing the call to the UDTF and a PARTITION BY expression.
- any other ***analytic function*** (page 141).
- a call to another UDTF.
- a ***TIMESERIES*** (page 894) clause.
- a ***pattern matching*** (page 331) clause.
- a gap filling and interpolation clause.

Notes

- The parameters and return values for the function are automatically determined by the CREATE TRANSFORM FUNCTION statement, based on data supplied by the factory class.
- You can assign multiple functions the same name if they have different parameters. When you call a multiply-defined function, HP Vertica selects the UDF function whose input parameters match the parameters in the function call to perform the processing. This behavior is similar to having multiple signatures for a method or function in other programming languages.
- You can return a list of all UDFs by querying the system table ***V_CATALOG.USER_FUNCTIONS***. Users see only the functions on which they have `EXECUTE` privileges.

CREATE LIBRARY

Loads a C++ shared library or R file containing user defined functions (UDFs). You supply the absolute path to a Linux shared library (.so) file or R file (.R) that contains the functions you want to access. See Developing and Using User Defined Functions in the Programmer's Guide for details. If you supply the optional OR REPLACE argument, the library will replace any existing library with the same name.

Warning: User defined libraries are directly loaded by HP Vertica and may be run within the database process. By default, most UDF's developed in C++ run in fenced mode so that the function process runs outside of HP Vertica. However, if you choose not to run your code in fenced mode, or the type of UDF cannot be run in fenced mode (for example, User Defined Load), then your custom code can negatively impact database. Poorly-coded UDFs can cause instability or even database crashes.

Syntax

```
CREATE [OR REPLACE] LIBRARY [[db-name.]schema.]library_name AS 'library_path' [
LANGUAGE 'language' ]
```

Parameters

`[ OR REPLACE ]`	If you do not supply this parameter, the CREATE LIBRARY statement fails if an existing library matches the name the library you are trying to define. If you do supply this parameter, the new library replaces the old.
`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not

	unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
library_name	A name to assign to this library. This is the name you use in a CREATE FUNCTION statement to enable user defined functions stored in the library. Note that this name is arbitrary. It does not need to reflect the name of the library file, although it would be less confusing if did.
'*library_path*'	The absolute path and filename of the library to load located on the initiator node.
'*language*'	The programming language used to develop the function. `'R'` and '`C++`' are supported. Default is '`C++`'.

Permissions

Must be a superuser to create or drop a library.

Notes

- As part of the loading process, HP Vertica distributes the library file to other nodes in the database. Any nodes that are down or that are added to the cluster later automatically receive a copy of the library file when they join the cluster. Subsequent modification (or deletion) of the file/path provided in the CREATE LIBRARY statement has no effect.
- The CREATE LIBRARY statement performs some basic checks on the library file to ensure it is compatible with HP Vertica. The statement fails if it detects that the library was not correctly compiled or it finds other basic incompatibilities. However, there are many issues in shared libraries that CREATE LIBRARY cannot detect. Simply loading the library is no guarantee that it functions correctly.
- Libraries are added to the database catalog, and therefore persist across database restarts.

Examples

To load a library in the home directory of the dbadmin account with the name MyFunctions:

```
=> CREATE LIBRARY MyFunctions AS 'home/dbadmin/my_functions.so';
```

To load a library located in the directory where you started vsql:

```
=> \set libfile '\''`pwd`'/MyOtherFunctions.so\'';
=> CREATE LIBRARY MyOtherFunctions AS :libfile;
```

See Also

DROP LIBRARY (page 815)

ALTER LIBRARY (page 658)

CREATE FUNCTION (UDF) (page 725)

CREATE NETWORK INTERFACE

Identifies a network interface to which the node belongs. Use this statement when you want to configure import/export from individual nodes to other HP Vertica clusters.

Syntax

```
CREATE NETWORK INTERFACE network-interface-name ON node-name with 'ip address of node'
```

Parameters

network-interface-name	The name you assign to the network interface.
node-name	The name of the node.
IP address of node	The IP address of the node.

You can then configure a HP Vertica database node to use the network interface for import/export. (See Identify the Database or Node(s) used for Import/Export for more information.)

Permissions

Must be a superuser to create a network interface.

CREATE PROCEDURE

Adds an external procedure to HP Vertica. See Implementing External Procedures in the Programmer's Guide for more information about external procedures.

Syntax

```
CREATE PROCEDURE [[db-name.]schema.]procedure-name (
... [ argname ] [ argtype [,...] ] )
... AS 'exec-name'
... LANGUAGE 'language-name'
... USER 'OS-user'
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column

	(myschema.mytable.column1), and as full qualification, a database, schema, table, and column (mydb.myschema.mytable.column1).
procedure-name	Specifies a name for the external procedure. If the procedure-name is schema-qualified, the procedure is created in the specified schema.
argname	[Optional] Presents a descriptive argument name to provide a cue to procedure callers.
argtype	[Optional] Specifies the data type for argument(s) that will be passed to the procedure. Argument types must match HP Vertica type names. See ***SQL Data Types*** (page 71).
AS	Specifies the executable program in the procedures directory.
LANGUAGE	Specifies the procedure language. This parameter must be set to EXTERNAL.
USER	Specifies the user executed as. The user is the owner of the file. The user cannot be root. **Note:** The external program must allow execute privileges for this user.

Permissions

To create a procedure a superuser must have CREATE privilege on schema to contain procedure.

Notes

- A procedure file must be owned by the database administrator (OS account) or by a user in the same group as the administrator. (The procedure file owner cannot be root.) The procedure file must also have the set UID attribute enabled, and allow read and execute permission for the group.
- By default, only a database superuser can execute procedures. However, a superuser can grant the right to execute procedures to other users. See ***GRANT (Procedure)*** (page 833).

Example

This example illustrates how to create a procedure named *helloplanet* for the *helloplanet.sh* external procedure file. This file accepts one varchar argument.

Sample file:

```
#!/bin/bash
echo "hello planet argument: $1" >> /tmp/myprocedure.log
exit 0
```

Issue the following SQL to create the procedure:

```
CREATE PROCEDURE helloplanet(arg1 varchar) as 'helloplanet.sh' language
'external' USER 'release';
```

See Also

DROP PROCEDURE (page 816)

Installing External Procedure Executables in the Programmer's Guide

CREATE PROFILE

Creates a profile that controls password requirements for users.

Syntax

```
CREATE PROFILE name LIMIT
... [PASSWORD_LIFE_TIME {life-limit | DEFAULT | UNLIMITED}]
... [PASSWORD_GRACE_TIME {grace_period | DEFAULT | UNLIMITED}]
... [FAILED_LOGIN_ATTEMPTS {login-limit | DEFAULT | UNLIMITED}]
... [PASSWORD_LOCK_TIME {lock-period | DEFAULT | UNLIMITED}]
... [PASSWORD_REUSE_MAX {reuse-limit | DEFAULT | UNLIMITED}]
... [PASSWORD_REUSE_TIME {reuse-period | DEFAULT | UNLIMITED}]
... [PASSWORD_MAX_LENGTH {max-length | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_LENGTH {min-length | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_LETTERS {min-letters | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_UPPERCASE_LETTERS {min-cap-letters | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_LOWERCASE_LETTERS {min-lower-letters | DEFAULT | UNLIMITED}]
... [PASSWORD_MIN_DIGITS {min-digits | DEFAULT | UNLIMITED}]
```

Note: For all parameters, the special DEFAULT value means that the parameter's value is inherited from the DEFAULT profile. Any changes to the parameter in the DEFAULT profile is reflected by all of the profiles that inherit that parameter. Any parameter not specified in the CREATE PROFILE command is set to DEFAULT.

Parameters

Name	Description	Meaning of UNLIMITED value
name	The name of the profile to create	N/A
PASSWORD_LIFE_TIME *life-limit*	Integer number of days a password remains valid. After the time elapses, the user must change the password (or will be warned that their password has expired if PASSWORD_GRACE_TIME is set to a value other than zero or UNLIMITED).	Passwords never expire.

PASSWORD_GRACE_TIME *grace-period*	Integer number of days the users are allowed to login (while being issued a warning message) after their passwords are older than the PASSWORD_LIFE_TIME. After this period expires, users are forced to change their passwords on login if they have not done so after their password expired.	No grace period (the same as zero)
FAILED_LOGIN_ATTEMPTS *login-limit*	The number of consecutive failed login attempts that result in a user's account being locked.	Accounts are never locked, no matter how many failed login attempts are made.
PASSWORD_LOCK_TIME *lock-period*	Integer value setting the number of days an account is locked after the user's account was locked by having too many failed login attempts. After the PASSWORD_LOCK_TIME has expired, the account is automatically unlocked.	Accounts locked because of too many failed login attempts are never automatically unlocked. They must be manually unlocked by the database superuser.
PASSWORD_REUSE_MAX *reuse-limit*	The number of password changes that need to occur before the current password can be reused.	Users are not required to change passwords a certain number of times before reusing an old password.
PASSWORD_REUSE_TIME *reuse-period*	The integer number of days that must pass after a password has been set before the before it can be reused.	Password reuse is not limited by time.
PASSWORD_MAX_LENGTH *max-length*	The maximum number of characters allowed in a password. Value must be in the range of 8 to 100.	Passwords are limited to 100 characters.
PASSWORD_MIN_LENGTH *min-length*	The minimum number of characters required in a password. Valid range is 0 to *max-length*.	Equal to *max-length*.
PASSWORD_MIN_LETTERS *min-of-letters*	Minimum number of letters (a-z and A-Z) that must be in a password. Valid ranged is 0 to *max-length*.	0 (no minimum).
PASSWORD_MIN_UPPERCASE_LETTERS *min-cap-letters*	Minimum number of capital letters (A-Z) that must be in a password. Valid range is is 0 to *max-length*.	0 (no minimum).

PASSWORD_MIN_LOWERCASE_LETTERS *min-lower-letters*	Minimum number of lowercase letters (a-z) that must be in a password. Valid range is is 0 to *max-length*.	0 (no minimum).
PASSWORD_MIN_DIGITS *min-digits*	Minimum number of digits (0-9) that must be in a password. Valid range is is 0 to *max-length*.	0 (no minimum).
PASSWORD_MIN_SYMBOLS *min-symbols*	Minimum number of symbols (any printable non-letter and non-digit character, such as $, #, @, and so on) that must be in a password. Valid range is is 0 to *max-length*.	0 (no minimum).

Permissions

Must be a superuser to create a profile.

Note: Only the profile settings for how many failed login attempts trigger account locking and how long accounts are locked have an effect on external password authentication methods such as LDAP or Kerberos. All password complexity, reuse, and lifetime settings have an effect on passwords managed by HP Vertica only.

See Also

ALTER PROFILE (page 660)

DROP PROFILE (page 817)

CREATE PROJECTION

Creates metadata for a projection in the HP Vertica catalog. You can create a segmented projection, recommended for large tables. Unsegmented projections are recommended only for small tables, which are then replicated across all cluster nodes. You can also create projections using a combination of individual columns and grouped columns. You can optionally apply a specific access rank to one or more columns, and encoding for an individual column or group of columns.

Syntax

```
CREATE PROJECTION [ IF NOT EXISTS ]
...[[db-name.]schema.]projection-name
...[ ( { projection-column
...| { GROUPED ( column-reference1, column-reference2 [ ,... ])}
..... [ ACCESSRANK integer ]
..... [ ENCODING encoding-type (page 747) ] } [ ,... ] )
...]

AS SELECT table-column [ , ... ] FROM table-reference [ , ... ]
... [ WHERE join-predicate (page 64) [ AND join-predicate ] ...
... [ ORDER BY table-column [ , ... ] ]
... [ hash-segmentation-clause (page 750)
... | range-segmentation-clause (page 790)
... | UNSEGMENTED { NODE node | ALL NODES } ]
... [ KSAFE [ k-num ] ]
```

Parameters

`[ IF NOT EXISTS ]`	[Optional] Determines whether the statement generates a NOTICE message or an ERROR if *<object>-name* exists. Using `IF NOT EXISTS` generates a NOTICE if the specified object exists. Omitting the clause generates an error if *<object>-name* exists. Regardless of whether you use `IF NOT EXISTS`, HP Vertica does not create a new object if *<object>-name* exists. For more information, see also ON_ERROR_STOP.
[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
projection-name	Specifies the name of the projection to create. If the projection-name is schema-qualified, the projection is created in the specified schema. Otherwise, the projection is created in the same schema as the anchor table. If the projection-name you supply results in a naming conflict with existing catalog objects (projections), the CREATE PROJECTION statement fails. All projections include a base table name added

	automatically. You refer to the base table name to drop or alter a projection.

projection-column	[Optional] Specifies the name of one or more columns in the projection. The column data type is that of the corresponding column in the schema table (based on ordinal position).
	If you do not explicitly provide projection column names, the column names for the table specified in the SELECT statement are used. The following example automatically uses store and transaction as the projection column names for sales_p:
	`=> CREATE TABLE sales(store INTEGER, transaction INTEGER);`
	`=> CREATE PROJECTION sales_p AS SELECT * FROM sales KSAFE 1;`
	Note that you cannot use specific encodings on projection columns using the CREATE PROJECTION function this way.
	You can use different *projection-column* names to distinguish multiple columns with the same name in different tables so that no aliases are needed.

`[ ENCODING` *encoding-type* `]`	[Optional] Specifies the ***type of encoding*** (page 747) to use on the column. By default, the encoding type is auto. **Caution:** The NONE keyword is obsolete.

`[ ACCESSRANK` *integer* `]`	[Optional] Overrides the default access rank for a column. This is useful if you want to increase or decrease the speed at which a column is accessed. See Creating and Configuring Storage Locations and Prioritizing Column Access Speed in the Administrator's Guide.
	To use this option correctly, you must specify `ACCESSRANK` *n* directly with the column to which it applies. For instance, this statement correctly assigns an access rank to two of the three named columns `col_night`, and `col_day`:
	`CREATE PROJECTION PJ(col_night, accessrank 2, col_evening, col_day, accessrank 1)AS SELECT * FROM test`
	The following statement will fail, because the second access rank is not associated directly with a column:
	`CREATE PROJECTION PJ(col_night, col_evening, col_day, accessrank 2, accessrank 1) AS SELECT * FROM test`

`[ GROUPED (`*column-reference1, column-reference2* `[,...] ) ]`	[Optional] Groups the specified projection column references into a single disk file. Using the GROUPED parameter requires a minimum of two *column-reference* elements. You can specify more than one set of grouped columns, and intersperse them with non-grouped column references. Grouping projection columns minimizes file I/O for work loads that: ▪ Read a large percentage of the columns in a table. ▪ Perform single row look-ups. ▪ Query against many small columns. ▪ Frequently update data in these columns. If you have columns of data that are always accessed together, and not used in predicates, grouping the columns can increase query performance. However, once grouped, queries can no longer retrieve from disk all records for any individual columns within the group. **Note:** RLE compression is reduced when a RLE column is grouped

	with one or more non-RLE columns. When grouping columns you can: ▪ Group some of the columns: `(a, GROUPED(b, c), d)` ▪ Group all of the columns: `(GROUPED(a, b, c, d))` ▪ Create multiple groupings in the same projection: `(GROUPED(a, b), GROUPED(c, d))` **Note:** HP Vertica performs dynamic column grouping. For example, to provide better read and write efficiency for small loads, HP Vertica ignores any projection-defined column grouping (or lack thereof) and groups all columns together by default.
`SELECT` *table-column*	Specifies a list of schema table columns corresponding (in ordinal position) to the projection columns. **Note:** When creating a projection, the total number of projection column references, including those noted in the GROUPED statement, must equal the number of columns specified in the Select statement (unless the Select statement indicates `select *`).
table-reference	Specifies a list of schema tables containing the columns to include in the projection in the form: *table-name* `[ AS ]` *alias* `[ (` *column-alias* `[ , ...] ) ] [ , ...] ]`
`[ WHERE` *join-predicate* `]`	[Optional] Specifies foreign-key = primary-key equijoins between the large and smaller tables. No other predicates are allowed.
`[ ORDER BY` *table-column* `]`	[Optional] Specifies the columns to sort the projection on. You cannot specify an ascending or descending clause. HP Vertica always uses an ascending sort order in physical storage. **Note:** If you do not specify the `ORDER BY` *table-column* parameter, HP Vertica uses the order in which columns are specified in the column list as the sort order for the projection.
hash-segmentation-clause	[Optional] Segments a projection evenly and distributes the data across nodes using a built-in hash function. Creating a projection with hash segmentation results in optimal query execution. See ***hash-segmentation-clause*** (page 750). **Note:** An elastic projection (a segmented projection created when Elastic Cluster is enabled) created with a modularhash segmentation expression uses hash instead.
range-segmentation-clause	[Optional] 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*** (page 790).
`UNSEGMENTED` `{ NODE` *node* `| ALL NODES }`	[Optional] Lets you specify an unsegmented projection. The default for this parameter is to create an UNSEGMENTED projection on the initiator node. You can optionally use either of the following node specifications: ▪ NODE *node*—Creates an unsegmented projection only on the specified node. Projections for small tables must be UNSEGMENTED. ▪ ALL NODES—Automatically replicates the unsegmented

	projection on each node. To perform distributed query execution, HP Vertica requires an unsegmented copy of each small table superprojection on each node.
`KSAFE [ k-num ]`	[Optional] Specifies the K-safety level of the projection. The *k-num* integer determines how many replicated or segmented buddy projections are created. The value must be greater than or equal to the current K-safety level of the database and less than the total number of nodes. HP recommends that you use multiple projection syntax for K-safe clusters. If the KSAFE parameter is omitted, or specified without an integer value, the projection is created at the current system K-safety level. Unless you are creating a projection on a single-node database, the default KSAFE value is at least one. For instance, this example creates a superprojection for a database with a K-safety of one (1): `KSAFE 1` **Note:** KSAFE cannot be used with range segmentation.

Permissions

Projections get implicitly created when you insert data into a table, an operation that automatically creates a superprojection for the table.

Implicitly-created projections do not require any additional privileges to create or drop, other than privileges for table creation. Users who can create a table or drop a table can also create and drop the associated superprojection.

To explicitly create a projection using the ***CREATE PROJECTION*** (page 742) statement, a user must be a superuser or owner of the anchor table or have the following privileges:

- CREATE privilege on the schema in which the projection is created
- SELECT on all the base tables referenced by the projection
- USAGE on all the schemas that contain the base tables referenced by the projection

Explicitly-created projections can only be dropped by the table owner on which the projection is based for a single-table projection, or the owner of the anchor table for pre-join projections.

Projections and Super Projections

If you attempt to create a projection or a pre-join projection before a super-projection exists, HP Vertica displays a warning.

This example attempts to create a projection before a super-projection:

```
VMart=> create table tar (x integer, y integer);
CREATE TABLE
VMart=> create projection tar_p as select x from tar;
WARNING 4130:  No super projections created for table public.tar.
HINT:  Default super projections will be automatically created with the next DML
CREATE PROJECTION
```

This example attempts to create a pre-join projection when no super-projection exists:

```
VMart=> create projection foo_p as select near.x, far.x from near join far on near.x
= far.x unsegmented all nodes;
WARNING 4130:  No super projections created for table public.far.
HINT:  Default super projections will be automatically created with the next DML
CREATE PROJECTION
```

Checking Column Constraints

When you create a projection, HP Vertica checks column constraints during the process. For instance, if a join predicate includes a column with a FOREIGN_KEY constraint but without a NOT_NULL constraint, the CREATE PROJECTION statement fails.

Updating the Projection Using Refresh

Invoking the CREATE PROJECTION command does not load data into physical storage. If the tables over which the projection is defined already contain data, you must issue ***START_REFRESH*** (page 538) to update the projection. Depending on how much data is in the tables, updating a projection can be time-consuming. Once a projection is up-to-date, however, it is updated automatically as part of COPY, DELETE, INSERT, MERGE, or UPDATE statements.

Monitoring Projecting Refresh on Buddy Projects

A projection is not refreshed until after a buddy projection is created. After the CREATE PROJECTION is run, if you run `SELECT START_REFRESH()` the following message displays:

```
Starting refresh background process
```

However, the refresh does not begin until after a buddy projection is created. You can monitor the refresh operation by examining the `vertica.log` file or view the final status of the projection refresh by using `SELECT get_projections('table-name;)`. For example:

```
=> SELECT get_projections('customer_dimension');
                 get_projections
-------------------------------------------------------------
Current system K is 1.
# of Nodes: 4.
Table public.customer_dimension has 4 projections.

Projection Name: [Segmented] [Seg Cols] [# of Buddies] [Buddy
Projections] [Safe] [UptoDate]
-----------------------------------------------------------
public.customer_dimension_node0004 [Segmented: No] [Seg Cols: ] [K: 3]
[public.customer_dimension_node0003,
public.customer_dimension_node0002,
public.customer_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
public.customer_dimension_node0003 [Segmented: No] [Seg Cols: ] [K: 3]
[public.customer_dimension_node0004,
public.customer_dimension_node0002,
public.customer_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
```

```
public.customer_dimension_node0002 [Segmented: No] [Seg Cols: ] [K: 3]
[public.customer_dimension_node0004,
public.customer_dimension_node0003,
public.customer_dimension_node0001] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
public.customer_dimension_node0001 [Segmented: No] [Seg Cols: ] [K: 3]
[public.customer_dimension_node0004,
public.customer_dimension_node0003,
public.customer_dimension_node0002] [Safe: Yes] [UptoDate: Yes][Stats:
Yes]
(1 row)
```

Creating Unsegmented Projections with the ALL NODES Option

Using the UNSEGMENTED option to create a projection takes a snapshot of the nodes defined at execution time to generate a predictable node list order. Creating unsegmented projections results in each replicated projection having the following naming convention, with an appended node-name suffix:

projection-name_node-name

For example, if the cluster node names are NODE01, NODE02, and NODE03, creating an unsegmented projection with the following command, creates projections named `ABC_NODE01`, `ABC_NODE02`, and `ABC_NODE03`:

```
=> CREATE PROJECTION ABC ... UNSEGMENTED ALL NODES;
```

Creating unsegmented projections (with a node-name suffix), affects the projection name argument value for functions such as ***GET_PROJECTIONS*** (page 499) or ***GET_PROJECTION_STATUS*** (page 498). For these functions, you must provide the entire projection name, including the node-name suffix (`ABC_NODE01)`, rather than the projection name alone (`ABC)`.

To view a list of the nodes in a database, use the View Database command in the Administration Tools.

Example

The following example groups the highly correlated columns bid and ask. However, the stock column is stored separately.

```
=> CREATE TABLE trades (stock CHAR(5), bid INT, ask INT);
=> CREATE PROJECTION tradeproj (stock ENCODING RLE, GROUPED(bid ENCODING DELTAVAL,
ask))
      AS (SELECT * FROM trades) KSAFE 1;
```

encoding-type

HP Vertica supports the following encoding and compression types:

ENCODING AUTO (default)

For CHAR/VARCHAR, BOOLEAN, BINARY/VARBINARY, and FLOAT columns, Lempel-Ziv-Oberhumer-based (LZO) compression is used.

For INTEGER, DATE/TIME/TIMESTAMP, and INTERVAL types, the compression scheme is based on the delta between consecutive column values.

AUTO encoding is ideal for sorted, many-valued columns such as primary keys. It is also suitable for general purpose applications for which no other encoding or compression scheme is applicable. Therefore, it serves as the default if no encoding/compression is specified.

The CPU requirements for this type are relatively small. In the worst case, data might expand by eight percent (8%) for LZO and twenty percent (20%) for integer data.

ENCODING BLOCK_DICT

For each block of storage, HP Vertica compiles distinct column values into a dictionary and then stores the dictionary and a list of indexes to represent the data block.

BLOCK_DICT is ideal for few-valued, unsorted columns in which saving space is more important than encoding speed. Certain kinds of data, such as stock prices, are typically few-valued within a localized area once the data is sorted, such as by stock symbol and timestamp, and are good candidates for BLOCK_DICT. Long CHAR/VARCHAR columns are not good candidates for BLOCK_DICT encoding.

CHAR and VARCHAR columns that contain 0x00 or 0xFF characters should not be encoded with BLOCK_DICT. Also, BINARY/VARBINARY columns do not support BLOCK_DICT encoding.

The encoding CPU for BLOCK_DICT is significantly higher than for default encoding schemes. The maximum data expansion is eight percent (8%).

ENCODING BLOCKDICT_COMP

This encoding type is similar to BLOCK_DICT except that dictionary indexes are entropy coded. This encoding type requires significantly more CPU time to encode and decode and has a poorer worst-case performance. However, if the distribution of values is extremely skewed, using BLOCKDICT_COMP encoding can lead to space savings.

ENCODING COMMONDELTA_COMP

This compression scheme builds a dictionary of all the deltas in the block and then stores indexes into the delta dictionary using entropy coding.

This scheme is ideal for sorted FLOAT and INTEGER-based (DATE/TIME/TIMESTAMP/INTERVAL) data columns with predictable sequences and only the occasional sequence breaks, such as timestamps recorded at periodic intervals or primary keys. For example, the following sequence compresses well: 300, 600, 900, 1200, 1500, 600, 1200, 1800, 2400. The following sequence does not compress well: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55.

If the delta distribution is excellent, columns can be stored in less than one bit per row. However, this scheme is very CPU intensive. If you use this scheme on data with arbitrary deltas, it can lead to significant data expansion.

ENCODING DELTARANGE_COMP

This compression scheme is primarily used for floating-point data, and it stores each value as a delta from the previous one.

This scheme is ideal for many-valued FLOAT columns that are either sorted or confined to a range. Do not use this scheme for unsorted columns that contain NULL values, as the storage cost for representing a NULL value is high. This scheme has a high cost for both compression and decompression.

To determine if DELTARANGE_COMP is suitable for a particular set of data, compare it to other schemes. Be sure to use the same sort order as the projection, and select sample data that will be stored consecutively in the database.

ENCODING DELTAVAL

For INTEGER and DATE/TIME/TIMESTAMP/INTERVAL columns, data is recorded as a difference from the smallest value in the data block. This encoding has no effect on other data types.

DELTAVAL is best used for many-valued, unsorted integer or integer-based columns. The CPU requirements for this encoding type are minimal, and the data never expands.

ENCODING GCDDELTA

For INTEGER and DATE/TIME/TIMESTAMP/INTERVAL columns, and NUMERIC columns with 18 or fewer digits, data is recorded as the difference from the smallest value in the data block divided by the greatest common divisor (GCD) of all entries in the block. This encoding has no effect on other data types.

ENCODING GCDDELTA is best used for many-valued, unsorted, integer columns or integer-based columns, when the values are a multiple of a common factor. For example, timestamps are stored internally in microseconds, so data that is only precise to the millisecond are all multiples of 1000. The CPU requirements for decoding GCDDELTA encoding are minimal, and the data never expands, but GCDDELTA may take more encoding time than DELTAVAL.

ENCODING RLE

Run Length Encoding (RLE) replaces sequences (runs) of identical values with a single pair that contains the value and number of occurrences. Therefore, it is best used for low cardinality columns that are present in the ORDER BY clause of a projection.

The HP Vertica execution engine processes RLE encoding run-by-run and the HP Vertica optimizer gives it preference. Use it only when the run length is large, such as when low-cardinality columns are sorted.

The storage for RLE and AUTO encoding of CHAR/VARCHAR and BINARY/VARBINARY is always the same.

ENCODING NONE

Do not specify this value. It is obsolete and exists only for backwards compatibility. The result of ENCODING NONE is the same as ENCODING AUTO except when applied to CHAR and VARCHAR columns. Using ENCODING NONE on these columns increases space usage, increases processing time, and leads to problems if 0x00 or 0xFF characters are present in the data.

hash-segmentation-clause

Allows you to segment a projection based on a built-in hash function. The built-in hash function provides even data distribution across some or all nodes in a cluster, resulting in optimal query execution.

Note: Hash segmentation is the preferred method of segmentation. The Database Designer uses hash segmentation by default.

Syntax

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

Parameters

`SEGMENTED BY` *expression*	Can be a general SQL expression. However, HP recommends using the built-in ***HASH*** (page 308) or ***MODULARHASH*** (page 311) functions, specifying table columns as arguments. If you specify only a column name, HP Vertica gives a warning. Choose columns that have a large number of unique data values and acceptable skew in their data distribution. Primary key columns that meet the criteria could be an excellent choice for hash segmentation.
`ALL NODES`	Automatically distributes data evenly across all nodes at the time the `CREATE PROJECTION` statement is run. The ordering of the nodes is fixed.
`OFFSET` *offset*	[Optional.] An offset value indicating the node on which to start the segmentation distribution. Omitting the OFFSET clause is equivalent to OFFSET 0. The offset is an integer value, relative to 0, based on all available nodes when using the ALL NODES parameter. 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

- CREATE PROJECTION accepts the deprecated syntax `SITES` *node* for compatibility with previous releases.
- You must use the table column names in the expression, not the new projection column names.
- An elastic projection (a segmented projection created when Elastic Cluster is enabled) created with a modularhash segmentation expression uses hash instead.
- To use a `SEGMENTED BY` expression other than HASH or MODULARHASH, the following restrictions apply:
  - All leaf expressions must be either constants or ***column-references*** (see "***Column References***" on page 54) 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 <= x < 2^{63} (two to the sixty-third power or 2^63), and values are generally distributed uniformly over that range.
- If *expression* produces a value outside of the expected range (a negative value for example), no error occurs, and the row is added to the first segment of the projection.

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.

```
=> CREATE PROJECTION fact_ts_2 (
      f_price,
      f_cid,
      f_tid,
      f_cost,
      f_date) AS (
   SELECT price, cid, tid, cost, dwdate FROM fact)
SEGMENTED BY MODULARHASH(dwdate) ALL NODES OFFSET 2;
```

See Also

HASH (page 308) and ***MODULARHASH*** (page 311)

range-segmentation-clause

Allows you to segment a projection based on a known range of values stored in a specific column. Choosing a range of values from a specific column provides even distribution of data across a set of nodes, resulting in optimal query execution.

Note: HP recommends that you use hash segmentation, rather than range segmentation.

Syntax

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

Parameters (Range Segmentation)

`SEGMENTED BY` *expression*	Is a single ***column reference*** (see "***Column References***" on page 54) to a column in the SELECT list of the CREATE PROJECTION statement. 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. **Note:** 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*	Is 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 SELECT * FROM NODE_RESOURCES.
VALUES LESS THAN *value*	Specifies that the segment can contain only a range of data values less than *value*. The segments cannot overlap so the minimum value of the range is determined by the *value* of the previous segment (if any).
VALUES LESS THAN MAXVALUE	Specifies a sub-range containing data values with no upper limit. MAXVALUE is the maximum value represented by 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 54) for range segmentation. If you want to use a different expression, the following restrictions apply:
  - All leaf expressions must be either constants or ***column-references*** (see "***Column References***" on page 54) 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.
- 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.

See Also

NODE_RESOURCES (page 1047)

CREATE RESOURCE POOL

Creates a resource pool.

Syntax

```
CREATE RESOURCE POOL pool-name
... [ MEMORYSIZE {'sizeUnits' |DEFAULT }]
... [ MAXMEMORYSIZE {'sizeUnits' | NONE |DEFAULT }]
... [ EXECUTIONPARALLELISM {int | AUTO | DEFAULT} ]
... [ PRIORITY {integer |DEFAULT}]
... [ RUNTIMEPRIORITY { HIGH | MEDIUM | LOW |DEFAULT} ]
... [ RUNTIMEPRIORITYTHRESHOLD { integer |DEFAULT }]
... [ QUEUETIMEOUT {integer | NONE |DEFAULT}]
... [ PLANNEDCONCURRENCY {integer | NONE | DEFAULT |AUTO }]
... [ MAXCONCURRENCY {integer| NONE |DEFAULT }]
... [ RUNTIMECAP {interval | NONE | DEFAULT} ]
... [ SINGLEINITIATOR { bool | DEFAULT }]
```

Parameters

Note: If you specify DEFAULT for any parameter when creating a resource pool, HP Vertica uses the default value for the user-defined pool, stored in the ***RESOURCE_POOL_DEFAULTS*** (page 964) table.

pool-name	Specifies the name of the resource pool to create.
MEMORYSIZE '*sizeUnits*'	[Default 0%] Amount of memory allocated to the resource pool. See also MAXMEMORYSIZE parameter. Units can be one of the following: ▪ % percentage of total memory available to the Resource Manager. (In this case, size must be 0-100.). ▪ K Kilobytes ▪ M Megabytes ▪ G Gigabytes ▪ T Terabytes **Note:** This parameter refers to memory allocated to this pool per node and not across the whole cluster. The default of 0% means that the pool has no memory allocated to it and must exclusively borrow from the GENERAL pool.
MAXMEMORYSIZE '*sizeUnits*' | NONE	[Default unlimited] Maximum size the resource pool could grow by borrowing memory from the GENERAL pool. See ***Built-in Pools*** (page 757) for a discussion on how resource pools interact with the GENERAL pool. Units can be one of the following: ▪ % percentage of total memory available to the Resource Manager. (In this case, size must be 0-100). This notation has special meaning for the GENERAL

	pool, described in Notes below. ▪ K Kilobytes ▪ M Megabytes ▪ G Gigabytes ▪ T Terabytes If MAXMEMORYSIZE NONE is specified, there is no upper limit. **Notes:** The MAXMEMORYSIZE parameter refers to the maximum memory borrowed by this pool per node and not across the whole cluster. The default of unlimited means that the pool can borrow as much memory from GENERAL pool as is available.
EXECUTIONPARALLELISM	[Default: AUTO] Limits the number of threads used to process any single query issued in this resource pool. When set to AUTO, HP Vertica sets this value based on the number of cores, available memory, and amount of data in the system. Unless data is limited, or the amount of data is very small, HP Vertica sets this value to the number of cores on the node. Reducing this value increases the throughput of short queries issued in the pool, especially if the queries are executed concurrently. If you choose to set this parameter manually, set it to a value between 1 and the number of cores.
PRIORITY	[Default 0] An integer that represents priority of queries in this pool, when they compete for resources in the GENERAL pool. Higher numbers denote higher priority.
RUNTIMEPRIORITY	[Default: Medium] Determines the amount of run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to queries already running in the resource pool. Valid values are: ▪ HIGH ▪ MEDIUM ▪ LOW Queries with a HIGH run-time priority are given more CPU and I/O resources than those with a MEDIUM or LOW run-time priority.
RUNTIMEPRIORITYTHRESHOLD	[Default 2] Specifies a time limit (in seconds) by which a query must finish before the Resource Manager assigns to it the RUNTIMEPRIORITY of the resource pool. All queries begin runnng at a HIGH priority. When a query's duration exceeds this threshold, it is assigned the RUNTIMEPRIORITY of the resource pool.

QUEUETIMEOUT	[Default 300 seconds] An integer, in seconds, that represents the maximum amount of time the request is allowed to wait for resources to become available before being rejected. If set to NONE, the request can be queued for an unlimited amount of time.
PLANNEDCONCURRENCY	[Default: AUTO] Integer that represents the typical number of queries running concurrently in the system. When set to AUTO, this value is calculated automatically at query runtime. HP Vertica sets this parameter to the lower of these two calculations: ▪ Number of cores ▪ Memory/2GB When this parameter is set to AUTO, HP Vertica will never set it to a value less than 4. HP Vertica advises changing this value only after evaluating performance over a period of time. **Notes:** ▪ This is a cluster-wide maximum and not a per-node limit. ▪ For clusters where the number of cores differs on different nodes, AUTO can apply differently on each node. Distributed queries run like the minimal effective planned concurrency. Single node queries run with the planned concurrency of the initiator. ▪ If you created or upgraded your database in 4.0 or 4.1, the PLANNEDCONCURRENCY setting on the GENERAL pool defaults to a too-small value for machines with large numbers of cores. To adjust to a more appropriate value: ▪ => ALTER RESOURCE POOL general PLANNEDCONCURRENCY ▪ <#cores>; ▪ You need to set this parameter only if you created a database before 4.1, patchset 1.
MAXCONCURRENCY	[Default unlimited] An integer that represents the maximum number of concurrent execution slots available to the resource pool. If MAXCONCURRENCY NONE is specified, there is no limit. **Note:** This is a cluster-wide maximum and not a per-node limit.
RUNTIMECAP	[Default: NONE] Sets the maximum amount of time any query on the pool can execute. Set RUNTIMECAP using interval, such as '1 minute' or '100 seconds' (see ***Interval Values*** (page 37) for details). This value cannot exceed one year. Setting this value to NONE specifies that there is no time limit on queries running on the pool. If the user or session also has a RUNTIMECAP, the shorter limit applies.
SINGLEINITIATOR	[Default false] This parameter is included for backwards compatibility only. Do not change the value.

Permissions

Must be a superuser to create a resource pool.

Notes

- Resource pool names are subject to the same rules as HP Vertica ***identifiers*** (page 22). ***Built-in pool*** (page 757) names cannot be used for user-defined pools.
- New resource pools can be created or altered without shutting down the system.
- When a new pool is created (or its size altered), MEMORYSIZE amount of memory is taken out of the GENERAL ***pool*** (page 757). If the GENERAL pool does not currently have sufficient memory to create the pool due to existing queries being processed, a request is made to the system to create a pool as soon as resources become available. The pool is created immediately and memory is moved to the pool as it becomes available. Such memory movement has higher priority than any query.

  The pool is in operation as soon as the specified amount of memory becomes available. You can monitor whether the ALTER has been completed in the V_MONITOR.RESOURCE_POOL_STATUS (page 965) system table.
- Under normal operation, MEMORYSIZE is required to be less than MAXMEMORYSIZE and an error is returned during CREATE/ALTER operations if this size limit is violated. However, under some circumstances where the node specification changes by addition/removal of memory, or if the database is moved to a different cluster, this invariant could be violated. In this case, MAXMEMORYSIZE is increased to MEMORYSIZE.
- If two pools have the same PRIORITY, their requests are allowed to borrow from the GENERAL pool in order of arrival.

See Guidelines for Setting Pool Parameters in the Administrator's Guide for details about setting these parameters.

Example

The following command creates a resource pool with MEMORYSIZE of 1800MB to ensure that the CEO query has adequate memory reserved for it:

```
=> CREATE RESOURCE POOL ceo_pool MEMORYSIZE '1800M' PRIORITY 10;
\pset expanded
Expanded display is on.
SELECT * FROM resource_pools WHERE name = 'ceo_pool';
-[ RECORD 1 ]-------+------------
name                | ceo_pool
is_internal         | f
memorysize          | 1800M
maxmemorysize       |
priority            | 10
queuetimeout        | 300
plannedconcurrency  | 4
maxconcurrency      |
singleinitiator     | f
```

Assuming the CEO report user already exists, associate this user with the above resource pool using `ALTER USER` statement.

```
=> ALTER USER ceo_user RESOURCE POOL ceo_pool;
```

Issue the following command to confirm that the ceo_user is associated with the ceo_pool:

```
=> SELECT * FROM users WHERE user_name ='ceo_user';
-[ RECORD 1 ]-+------------------
user_id       | 45035996273713548
user_name     | ceo_user
is_super_user | f
resource_pool | ceo_pool
memory_cap_kb | unlimited
```

See Also

ALTER RESOURCE POOL (page 663)

CREATE USER (page 801)

DROP RESOURCE POOL (page 819)

SET SESSION RESOURCE POOL (page 916)

SET SESSION MEMORYCAP (page 915)

Managing Workloads in the Administrator's Guide

Built-in Pools

HP Vertica is preconfigured with built-in pools for various system tasks. The built-in pools can be reconfigured to suit your usage. The following sections describe the purpose of built-in pools and the default settings.

Built-in Pool	Settings
`GENERAL`	A special, catch-all pool used to answer requests that have no specific resource pool associated with them. Any memory left over after memory has been allocated to all other pools is automatically allocated to the `GENERAL` pool. The `MEMORYSIZE` parameter of the `GENERAL` pool is undefined (variable), however, the `GENERAL` pool must be at least 1GB in size and cannot be smaller than 25% of the memory in the system. The `MAXMEMORYSIZE` parameter of the `GENERAL` pool has special meaning; when set as a % value it represents the percent of total physical RAM on the machine that the Resource Manager can use for queries. By default, it is set to 95%. The `GENERAL.MAXMEMORYSIZE` governs the total amount of RAM that the Resource Manager can use for queries, regardless of whether it is set to a percent or to a specific value (for example, '10GB') Any user-defined pool can “borrow” memory from the `GENERAL` pool to satisfy requests that need extra memory until the `MAXMEMORYSIZE` parameter of that pool is reached. If the pool is configured to have `MEMORYSIZE` equal to `MAXMEMORYSIZE`, it cannot

	borrow any memory from the GENERAL pool and is said to be a standalone resource pool. When multiple pools request memory from the GENERAL pool, they are granted access to general pool memory according to their priority setting. In this manner, the GENERAL pool provides some elasticity to account for point-in-time deviations from normal usage of individual resource pools.
SYSQUERY	The pool that runs queries against ***system monitoring and catalog tables*** (page 933). The SYSQUERY pool reserves resources for system table queries so that they are never blocked by contention for available resources.
SYSDATA	The pool reserved for temporary storage of intermediate results of queries against ***system monitoring and catalog tables*** (page 933). If the SYSDATA pool size is too low, HP Vertica cannot execute queries for large system tables or during high concurrent access to system tables.
WOSDATA	The Write Optimized Store (WOS) resource pool. Data loads to the WOS automatically spill to the ROS once it exceeds a certain amount of WOS usage; the PLANNEDCONCURRENCY parameter of the WOS is used to determine this spill threshold. For instance, if PLANNEDCONCURRENCY of the WOSDATA pool is set to 4, once a load has occupied one quarter of the WOS, it spills to the ROS. See Scenario: Tuning for Continuous Load and Query in the Administrator's Guide.
TM	The Tuple Mover (TM) pool. You can use the MAXCONCURRENCY parameter for the TM pool to allow more than one concurrent TM operation to occur. See Scenario: Tuning Tuple Mover Pool Settings in the Administrator's Guide.
RECOVERY	The pool used by queries issued when recovering another node of the database. The MAXCONCURRENCY parameter is used to determine how many concurrent recovery threads to use. You can use the PLANNEDCONCURRENCY parameter (by default, set to twice the MAXCONCURRENCY) to tune how to apportion memory to recovery queries. See Scenario: Tuning for Recovery in the Administrator's Guide.
REFRESH	The pool used by queries issued by the ***PROJECTION_REFRESHES*** (page 1056) operations. Refresh does not currently use multiple concurrent threads; thus, changes to the MAXCONCURRENCY values have no effect. See Scenario: Tuning for Refresh in the Administrator's Guide.
DBD	The Database Designer pool, used to control resource usage for the DBD internal processing. Since the Database Designer is such a resource-intensive process, the DBD pool is configured with a zero (0) second QUEUETIMEOUT value. Whenever resources are under pressure, this timeout setting causes the DBD to time out immediately, and not be queued to run later. The Database Designer then requests the user to run the designer later, when resources are more available. HP recommends that you do not reconfigure this pool.

Upgrade from Earlier Versions of HP Vertica

For a database being upgraded from earlier versions of, HP Vertica automatically translates most existing parameter values into the new resource pool settings.

The PLANNEDCONCURRENCY and MAXCONCURRENCY parameters of the resource pools must be manually tuned per Guidelines for Setting Pool Parameters in the Administrator's Guide.

See Also

RESOURCE_AQUISITIONS (page 1081)

Built-in Pool Configuration

The following tables list the default configuration setting values of built-in resource pools for a new database and for a database upgraded from prior versions of HP Vertica.

Note: Some of the built-in resource pool parameter values have restrictions, which are noted in the tables.

GENERAL

Setting	Value
MEMORYSIZE	N/A (cannot be set)
MAXMEMORYSIZE	Default: 95% of Total RAM on the node. Setting this value to 100% generates warnings that swapping could result. MAXMEMORYSIZE has the following restrictions: ▪ Must be 1GB or greater. ▪ Must not be less than 25% of total system RAM.
PRIORITY	0
RUNTIMEPRIORITY	Medium
RUNTIMEPRIORITYTHRESHOLD	2
QUEUETIMEOUT	300
RUNTIMECAP	NONE

PLANNEDCONCURRENCY	[Default: Auto] An integer that represents the number of concurrent queries that are normally expected to be running against the resource pool. When set to the default value of AUTO, HP Vertica automatically sets PLANNEDCONCURRENCY at query runtime, choosing the lower of these two values: ▪ # of cores ▪ Memory/2BG The value 4 is the minimum value for PLANNEDCONCURRENCY. HP Vertica advises changing this value only after evaluating performance over a period of time. **Notes:** ▪ See Best Practices For Workload Management in the Administrator's Guide for guidance on how to tune. ▪ The PLANNEDCONCURRENCY setting for the GENERAL pool defaults to a too-small value for machines with large numbers of cores. To adjust to a more appropriate value: ▪ => ALTER RESOURCE POOL general PLANNEDCONCURRENCY <#cores>; See Guidelines for Setting Pool Parameters in the Administrator's Guide
MAXCONCURRENCY	Unlimited Restrictions: Setting to 0 generates warnings that no system queries may be able to run in the system.
SINGLEINITIATOR	False. This parameter is included for backwards compatibility only. Do not change the value.

SYSQUERY

Setting	Value
MEMORYSIZE	64M Restrictions: Setting to <20M generates warnings because it could prevent system queries from running and make problem diagnosis difficult.
MAXMEMORYSIZE	Unlimited
EXECUTIONPARALLELISM	AUTO
PRIORITY	110
RUNTIMEPRIORITY	HIGH
RUNTIMEPRIORITYTHRESHOLD	0
QUEUETIMEOUT	300
RUNTIMECAP	NONE.
PLANNEDCONCURRENCY	See GENERAL

MAXCONCURRENCY	Unlimited Restrictions: Setting to 0 generates warnings that no system queries may be able to run in the system.
SINGLEINITIATOR	False. This parameter is included for backwards compatibility only. Do not change the value.

SYSDATA

Setting	Value
MEMORYSIZE	100m
MAXMEMORYSIZE	10% Restriction: Setting To <4m generates warnings that no system queries may be able to run in the system.
EXECUTIONPARALLELISM	N/A (cannot be set)
PRIORITY	N/A (cannot be set)
RUNTIMEPRIORITY	N/A (cannot be set)
RUNTIMEPRIORITYTHRESHOLD	N/A (cannot be set)
QUEUETIMEOUT	N/A (cannot be set)
RUNTIMECAP	N/A (cannot be set)
PLANNEDCONCURRENCY	N/A (cannot be set)
MAXCONCURRENCY	N/A (cannot be set)
SINGLEINITIATOR	N/A (cannot be set)

WOSDATA

Setting	Value
MEMORYSIZE	0%
MAXMEMORYSIZE	25% or 2GB, whichever is less.
EXECUTIONPARALLELISM	N/A (cannot be set)
PRIORITY	N/A (cannot be set)
RUNTIMEPRIORITY	N/A (cannot be set)
RUNTIMEPRIORITYTHRESHOLD	N/A (cannot be set)
QUEUETIMEOUT	N/A (cannot be set)
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	2*# nodes
MAXCONCURRENCY	N/A (cannot be set)
SINGLEINITIATOR	N/A (cannot be set)

TM

Setting	Value
MEMORYSIZE	100M
MAXMEMORYSIZE	Unlimited
EXECUTIONPARALLELISM	AUTO
PRIORITY	105
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	60
QUEUETIMEOUT	300
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	1
MAXCONCURRENCY	2 Restrictions: Cannot set to 0 or NONE (unlimited)
SINGLEINITIATOR	True. This parameter is included for backwards compatibility. Do not change the value.

REFRESH

Setting	Value
MEMORYSIZE	0%
MAXMEMORYSIZE	Unlimited
EXECUTIONPARALLELISM	AUTO
PRIORITY	-10
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	60
QUEUETIMEOUT	300
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	4
MAXCONCURRENCY	Unlimited Restrictions: cannot set to 0
SINGLEINITIATOR	True. This parameter is included for backwards compatibility. Do not change the value.

RECOVERY

Setting	Value
MEMORYSIZE	0%
MAXMEMORYSIZE	Unlimited Restrictions: cannot set to < 25%.
EXECUTIONPARALLELISM	AUTO
PRIORITY	107
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	60
QUEUETIMEOUT	300
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	Twice MAXCONCURRENCY
MAXCONCURRENCY	(# of cores / 2) + 1 Restrictions: Cannot set to 0 or NONE (unlimited)
SINGLEINITIATOR	True. This parameter is included for backwards compatibility. Do not change the value.

DBD

Setting	Value
MEMORYSIZE	0%
MAXMEMORYSIZE	Unlimited
EXECUTIONPARALLELISM	AUTO
PRIORITY	0
RUNTIMEPRIORITY	MEDIUM
RUNTIMEPRIORITYTHRESHOLD	0
QUEUETIMEOUT	0
RUNTIMECAP	NONE
PLANNEDCONCURRENCY	See GENERAL
MAXCONCURRENCY	Unlimited
SINGLEINITIATOR	False. This parameter is included for backwards compatibility. Do not change the value.

CREATE ROLE

Creates a new, empty role. You must then add permissions to the role using one of the GRANT statements.

Syntax

```
CREATE ROLE role;
```

Parameters

role	The name for the new role.

Permissions

Must be a superuser to create a role.

See Also

ALTER ROLE RENAME (page 667)

DROP ROLE (page 820)

CREATE SCHEMA

Defines a new schema.

Syntax

```
CREATE SCHEMA [ IF NOT EXISTS ] [db-name.]schema [ AUTHORIZATION user-name ]
```

Parameters

`[ IF NOT EXISTS ]`	[Optional] Determines whether the statement generates a NOTICE message or an ERROR if <*object*>-*name* exists. Using `IF NOT EXISTS` generates a NOTICE if the specified object exists. Omitting the clause generates an error if <*object*>-*name* exists. Regardless of whether you use `IF NOT EXISTS`, HP Vertica does not create a new object if <*object*>-*name* exists. For more information, see also ON_ERROR_STOP.
[db-name.]	[Optional] Specifies the current database name. Using a database name prefix is optional, and does not affect the command in any way. You must be connected to the specified database.
schema	Specifies the name of the schema to create.
`AUTHORIZATION` *user-name*	Assigns ownership of the schema to a user. If a user name is not provided, the user who creates the schema is assigned ownership. Only a Superuser is allowed to create a schema that is owned by a different user.

Privileges

To create a schema, the user must either be a superuser or have CREATE privilege for the database. See ***GRANT (Database)*** (page 832).

Optionally, CREATE SCHEMA could include the following sub-statements to create tables within the schema:

- ***CREATE TABLE*** (page 770)
- GRANT

With the following exceptions, these sub-statements are treated as if they have been entered as individual commands after the CREATE SCHEMA statement has completed:

- If the AUTHORIZATION statement is used, all tables are owned by the specified user.
- The CREATE SCHEMA statement and all its associated sub-statements are completed as one transaction. If any of the statements fail, the entire CREATE SCHEMA statement is rolled back.

Examples

The following example creates a schema named `s1` with no objects.

```
=> CREATE SCHEMA s1;
```

The following command creates schema s2 if it does not already exist:

```
=> CREATE SCHEMA IF NOT EXISTS schema2;
```

If the schema already exists, HP Vertica returns a rollback message:

```
=> CREATE SCHEMA IF NOT EXISTS schema2;
NOTICE 4214:  Object "schema2" already exists; nothing was done
```

The following series of commands create a schema named `s1` with a table named `t1` and grants Fred and Aniket access to all existing tables and ALL privileges on table `t1`:

```
=> CREATE SCHEMA s1;
=> CREATE TABLE t1 (c INT);
=> GRANT USAGE ON SCHEMA s1 TO Fred, Aniket;
=> GRANT ALL ON TABLE t1 TO Fred, Aniket;
```

See Also

ALTER SCHEMA (page 668)

SET SEARCH_PATH (page 912)

DROP SCHEMA (page 821)

CREATE SEQUENCE

Defines a new named sequence number generator object.

Use sequences or auto-incrementing columns for primary key columns. For example, to generate only even numbers in a sequence, specify a start value of 2, and increment the sequence by 2. For more information see Using Sequences in the Administrator's Guide. Sequences guarantee uniqueness and avoid constraint enforcement problems and their associated overhead.

Syntax

```
CREATE SEQUENCE [[db-name.]schema.]sequence_name
... [ INCREMENT [ BY ] positive_or_negative ]
... [ MINVALUE minvalue | NO MINVALUE ]
... [ MAXVALUE maxvalue | NO MAXVALUE ]
... [ START [ WITH ] start ]
... [ CACHE cache ]
... [ CYCLE | NO CYCLE ]
```

Parameters

[[*db-name.*]*schema.*]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
sequence_name	The name (optionally schema-qualified) of the sequence to create. The name must be unique among sequences, tables, projections, and views. If you do not specify a sequence_name, HP Vertica assigns an internally-generated name in the current schema.
INCREMENT [BY] *positive_or_negative*	Specifies how much to increment (or decrement) the current sequence value. A positive value creates an ascending sequence; a negative value makes a descending sequence. The default value is 1.
MINVALUE *minvalue* | NO MINVALUE	Determines the minimum value a sequence can generate. If you do not specify this clause, or you specify NO MINVALUE, default values are used. The defaults are 1 and $-2^{63}-1$ for ascending and descending sequences, respectively.
MAXVALUE *maxvalue* | NO MAXVALUE	Determines the maximum value for the sequence. If this clause is not supplied or you specify NO MAXVALUE, default values are used. The defaults are $2^{63}-1$ and -1 for ascending and descending sequences, respectively.
START [WITH] *startvalue*	Specifies a specific start value of the sequence (*startvalue*). The default values are *minvalue* for ascending sequences and *maxvalue* for descending sequences. **Note:** The [WITH] option is ignored and has no effect.
CACHE *cache*	Specifies how many sequence numbers are preallocated and stored in memory for faster access. The default is 250,000 with a minimum value of 1. Specifying a cache value of 1 indicates that only one value can be generated at a time, since no cache is assigned. **Notes:**

	▪ If you use the CACHE clause when creating a sequence, each session has its own cache on each HP Vertica node. ▪ Sequences that specify a cache size that is insufficient for the number of sequence values could cause a performance degradation.
`CYCLE | NO CYCLE`	Allows the sequence to restart when an ascending or descending sequence reaches the *maxvalue* or *minvalue*. If the sequence reaches a limit, the next number generated is the *minvalue* or *maxvalue*, respectively. If you specify NO CYCLE when creating a sequence, any calls to `NEXTVAL` (page 351) after the sequence reaches its maximum/minimum value return an error. NO CYCLE is the default.

Permissions

To create a sequence, the user must have CREATE privilege on the schema to contain the sequence. Only the owner and superusers can initially access the sequence. All other users must be granted access to the sequence by a superuser or the owner.

To create a table with a sequence, the user must have SELECT privilege on the sequence and USAGE privilege on the schema that contains the sequence.

Note: Referencing a named sequence in a ***CREATE TABLE*** (page 770) statement requires SELECT privilege on the sequence object and USAGE privilege on the schema of the named sequence.

Incrementing and Obtaining Sequence Values

After creating a sequence, use the ***NEXTVAL*** (page 351) function to create a cache in which the sequence value is stored. Use the ***CURRVAL*** (page 353) function to get the current sequence value.

You cannot use NEXTVAL or CURRVAL to act on a sequence in a SELECT statement:

- in a WHERE clause
- in a GROUP BY or ORDER BY clause
- in a DISTINCT clause
- along with a UNION
- in a subquery

Additionally, you cannot use NEXTVAL or CURRVAL to act on a sequence in:

- a subquery of UPDATE or DELETE
- a view

You can use subqueries to work around some of these restrictions. For example, to use sequences with a DISTINCT clause:

```
=> SELECT t.col1, shift_allocation_seq.nextval
   FROM (
     SELECT DISTINCT col1 FROM av_temp1) t;
```

Removing a Sequence

Use the ***DROP SEQUENCE*** (page 822) function to remove a sequence. You cannot drop a sequence upon which other objects depend. Sequences used in a default expression of a column cannot be dropped until all references to the sequence are removed from the default expression.

DROP SEQUENCE ... CASCADE is not supported.

Examples

The following example creates an ascending sequence called `my_seq`, starting at 100:

```
=> CREATE SEQUENCE my_seq START 100;
```

After creating a sequence, you must call the ***NEXTVAL*** (page 351) function at least once in a session to create a cache for the sequence and its initial value. Subsequently, use NEXTVAL to increment the sequence. Use the ***CURRVAL*** (page 353) function to get the current value.

The following NEXTVAL function instantiates the newly-created `my_seq` sequence and sets its first number:

```
=> SELECT NEXTVAL('my_seq');
 nextval
---------
     100
(1 row)
```

If you call CURRVAL before NEXTVAL, the system returns an error:

```
ERROR:  Sequence my_seq has not been accessed in the session
```

The following command returns the current value of this sequence. Since no other operations have been performed on the newly-created sequence, the function returns the expected value of 100:

```
=> SELECT CURRVAL('my_seq');
 currval
---------
     100
(1 row)
```

The following command increments the sequence value:

```
=> SELECT NEXTVAL('my_seq');
 nextval
---------
     101
(1 row)
```

Calling the CURRVAL again function returns only the current value:

```
=> SELECT CURRVAL('my_seq');
 currval
---------
     101
(1 row)
```

The following example shows how to use the `my_seq` sequence in an INSERT statement.

```
=> CREATE TABLE customer (
      lname VARCHAR(25),
      fname VARCHAR(25),
      membership_card INTEGER,
      id INTEGER
);
=> INSERT INTO customer VALUES ('Hawkins' ,'John', 072753, NEXTVAL('my_seq'));
```

Now query the table you just created to confirm that the ID column has been incremented to 102:

```
=> SELECT * FROM customer;
  lname   | fname | membership_card | id
---------+-------+-----------------+-----
 Hawkins | John  |           72753 | 102
(1 row)
```

The following example shows how to use a sequence as the default value for an INSERT command:

```
=> CREATE TABLE customer2(
      id INTEGER DEFAULT NEXTVAL('my_seq'),
      lname VARCHAR(25),
      fname VARCHAR(25),
      membership_card INTEGER
);
=> INSERT INTO customer2 VALUES (default,'Carr', 'Mary', 87432);
```

Now query the table you just created. The ID column has been incremented again to 103:

```
=> SELECT * FROM customer2;
 id  | lname | fname | membership_card
-----+-------+-------+-----------------
 103 | Carr  | Mary  |           87432
(1 row)
```

The following example shows how to use NEXTVAL in a SELECT statement:

```
=> SELECT NEXTVAL('my_seq'), lname FROM customer2;
 NEXTVAL | lname
---------+-------
     104 | Carr
(1 row)
```

As you can see, each time you call NEXTVAL(), the value increments.

The CURRVAL() function returns the current value.

See Also

ALTER SEQUENCE (page 669)

CREATE TABLE ***column-constraint*** (page 783)

CURRVAL (page 353)

DROP SEQUENCE (page 822)

GRANT (Sequence) (page 838)

NEXTVAL (page 351)

Using Sequences and Sequence Privileges in the Administrator's Guide

CREATE SUBNET

Identifies the subnet to which the nodes of an HP Vertica database belong. Use this statement when you want to configure import/export from a database to other HP Vertica clusters.

Syntax

```
CREATE SUBNET subnet-name with 'subnet prefix'
```

Parameters

subnet-name	The name you assign to the subnet.
subnet prefix	The routing prefix expressed in quad-dotted decimal representation. Refer to `v_monitor.network_interfaces` system table to get the prefix for all available IP networks.

You can then configure the database to use the subnet for import/export. (See Identify the Database or Node(s) used for Import/Export for more information.)

Permissions

Must be a superuser to create a subnet.

CREATE TABLE

Creates a table in the logical schema or an external table definition.

Syntax

```
CREATE TABLE [ IF NOT EXISTS ] [[db-name.]schema.]table-name
{
... ( column-definition (table) (page 779)  [ , ... ] )
... | [ table-constraint ( column_name, ... )]
... | [ column-name-list  (create table)  (page 780) ] AS [COPY] [ [ AT EPOCH LATEST
]
... | [ AT TIME 'timestamp' ] ] [ /*+ direct */ ] query
... | [ LIKE [[db-name.]schema.]existing-table [ INCLUDING PROJECTIONS | EXCLUDING
PROJECTIONS ] ]
}
... [ ORDER BY table-column [ , ... ] ]
... [ ENCODED BY column-definition [ , ... ]
... [ hash-segmentation-clause (page 750)
```

```
... | range-segmentation-clause (page 790)
... | UNSEGMENTED { NODE node | ALL NODES } ]
... [ KSAFE [k_num] ]
... [ PARTITION BY partition-clause ]
```

Parameters

`[ IF NOT EXISTS ]`	[Optional] Determines whether the statement generates a NOTICE message or an ERROR if <*object*>-*name* exists. Using `IF NOT EXISTS` generates a NOTICE if the specified object exists. Omitting the clause generates an error if <*object*>-*name* exists. Regardless of whether you use `IF NOT EXISTS`, HP Vertica does not create a new object if <*object*>-*name* exists. For more information, see also ON_ERROR_STOP.
`[[db-name.]schema.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`). If you do not specify a schema, the table is created in the default schema.
table-name	Identifies the name of the table to create. Schema-name specifies the schema where the table is created. If you omit schema-name, the new table is created in the first schema listed in the current ***search_path.*** (page 912)
column-definition	Defines one or more columns. See ***column-definition (table)*** (page 779).
column-name-list	Renames columns when creating a table from a query (CREATE TABLE AS SELECT). See ***column-name-list*** (page 780).
`AT EPOCH LATEST | AT TIME 'timestamp'`	Used with AS query to query historical data. You can specify AT EPOCH LATEST to include data from the latest committed DML transaction or specify a specific epoch based on its time stamp. **Note:** You cannot use either of these options when creating external tables.

`/*+ direct */`	Writes the data directly to disk (ROS) bypassing memory (WOS).
	HP Vertica accepts optional spaces before and after the plus (+) sign and the `direct` hint. Space characters between the opening `/*` or the closing `*/` are not permitted. The following directives are all acceptable:
	`/*+direct*/`
	`/* + direct*/`
	`/*+ direct*/`
	`/*+direct */`

`AS` *query*	Creates a new table from the results of a query and fills it with data from the query. For example:
	`CREATE TABLE promo AS SELECT ... ;`
	Column renaming is supported as part of the process:
	`CREATE TABLE promo (name, address, ...) AS SELECT customer_name, customer_address ... ;`
	The query table-column must be followed by the FROM clause to identify the table from which to copy the columns. See the example at the bottom of this topic as well as the ***SELECT*** (page 870) statement.
	If the query output has expressions other than simple columns (for example, constants, functions, etc) then you must either specify an alias for that expression, or list all columns in the column name list.
	Note: If any of the columns returned by *query* would result in a zero-width column, the column is automatically converted to a VARCHAR(80) column. For example, the following statement:
	`CREATE TABLE example AS SELECT '' AS X;`
	would attempt to create a table containing column X as a zero-width VARCHAR. Instead, HP Vertica automatically converts this column to a VARCHAR(80) to prevent the creation of a zero-width VARCHAR column. HP Vertica requires variable-width data type columns to be at least 1 character wide.

`[LIKE` *existing-table* `[INCLUDING | EXCLUDING PROJECTIONS]]`	Replicates an existing table to create a new table. The optional `INCLUDING PROJECTIONS` clause creates the current and non-pre-join projections of the original table when you populate the new table. If the table has an associated storage policy, the associated policy is also replicated. As a DDL statement, `EXCLUDING PROJECTIONS` is the default value. **NOTE:** HP Vertica does not support using `CREATE TABLE` *new_table* `LIKE` *table_exist* `INCLUDING PROJECTIONS` if *table_exist* is a temporary table. For more information about using this option, see Creating a Table Like Another in the Administrator's Guide.

`[ ORDER BY table-column ]`	[Optional] Specifies the sort order for the superprojection that is automatically created for the table. Data is in ascending order only. If you do not specify the sort order, HP Vertica uses the order in which columns are specified in the column definition as the sort order for the projection. For example: `ORDER BY col2, col1, col5` **Note:** You cannot use this option when creating external tables.
`ENCODED BY column-definition`	[CREATE TABLE AS query Only] This parameter is useful to specify the column encoding and/ or the access rank for specific columns in the query when a column-definition is not used to rename columns for the table to be created. See ***column-definition (table)*** (page 779) for examples. If you rename table columns when creating a table from a query, you can supply the encoding type and access rank in the column name list instead. **Note:** You cannot use this option when creating external tables.
hash-segmentation-clause	[Optional] Lets you segment the superprojection based on a built-in hash function that provides even distribution of data across nodes, resulting in optimal query execution. See ***hash-segmentation-clause*** (page 750). **Note:** You cannot use this option when creating external tables. **Note:** An elastic projection (a segmented projection created when Elastic Cluster is enabled) created with a modular hash segmentation expression uses hash instead.
range-segmentation-clause	[Optional] Lets you segment the superprojection 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*** (page 790). **Note:** You cannot use this option when creating external tables.
`UNSEGMENTED` `{ NODE node | ALL NODES }`	[Optional] Lets you specify an unsegmented projection. The default for this parameter is to create an UNSEGMENTED projection on the initiator node. You can optionally use either of the following node specifications: ▪ NODE *node*—Creates an unsegmented projection only on the specified node. Projections for small tables must be UNSEGMENTED. ▪ ALL NODES—Automatically replicates the unsegmented projection on each node. To perform distributed query execution, HP Vertica requires an unsegmented copy of each small table superprojection on each node. **Note:** You cannot use this option when creating external tables.
`KSAFE [ k ]`	[Optional] Specifies the K-safety level of the automatic projection created for the table.

	Note: You cannot use this option when creating external tables. The integer *k* determines how many unsegmented or segmented buddy projections are created. The value must be greater than or equal to the current K-safety level of the database and less than the total number of nodes. If KSAFE or its value are not specified, the superprojection is created at the current system K-safety level. For example: `K-SAFE 1` **Note:** When a hash-segmentation-clause is used with KSAFE, HP Vertica automatically creates *k_num*+1 buddy projections to meet the K-safety requirement.
`PARTITION BY` *partition-clause*	[Not supported for queries (`CREATE TABLE AS SELECT`)] ▪ The partition clause must calculate a single non-null value for each row. Multiple columns can be referenced, but a single value must be returned for each row. ▪ All leaf expressions must be either constants or columns of the table. ▪ All other expressions must be functions and operators; aggregate functions and queries are not permitted in the expression. ▪ SQL functions used in the partitioning expression must be immutable. **Note:** You cannot use this option when creating external tables.

Permissions

- To create a table, you must have CREATE privilege on the table schema.
- To create a table with a named sequence, you must have SELECT privilege on the sequence object and USAGE privilege on the schema associated with the sequence.
- Referencing a named sequence in a CREATE TABLE statement requires the following privileges:
  - SELECT privilege on sequence object
  - USAGE privilege on sequence schema
- To create a table with the LIKE clause, you must have owner permissions on the source table

Automatic projection creation

To get your database up and running quickly, HP Vertica automatically creates a default projection for each table created through the ***CREATE TABLE*** (page 770) and ***CREATE TEMPORARY TABLE*** (page 791) statements. Each projection created automatically (or manually) includes a base projection name prefix. You must use the projection prefix when altering or dropping a projection (***ALTER PROJECTION RENAME*** (page 659), ***DROP PROJECTION*** (page 818)).

How you use the CREATE TABLE statement determines when the projection is created:

- If you create a table without providing the projection-related clauses, HP Vertica automatically creates a superprojection for the table when you use an INSERT INTO or COPY statement to load data into the table for the first time. The projection is created in the same schema as the table. Once HP Vertica has created the projection, it loads the data.
- If you use CREATE TABLE AS SELECT to create a table from the results of a query, the table is created first and a projection is created immediately after, using some of the properties of the underlying SELECT query.
- (Advanced users only) If you use any of the following parameters, the default projection is created immediately upon table creation using the specified properties:
  - ***column-definition*** (page 779) (ENCODING encoding-type and ACCESSRANK integer)
  - ORDER BY table-column
  - ***hash-segmentation-clause*** (page 788)
  - ***range-segmentation-clause*** (page 790)
  - UNSEGMENTED { NODE *node* | ALL NODES }
  - KSAFE

  Note: Before you define a superprojection in the above manner, read Creating Custom Designs in the Administrator's Guide.

Characteristics of default automatic projections

A default auto-projection has the following characteristics:

- It is a superprojection.
- It uses the default ***encoding-type*** (page 747) AUTO.
- If created as a result of a CREATE TABLE AS SELECT statement, uses the encoding specified in the query table.
- Auto-projections use hash segmentation.
- The number of table columns used in the segmentation expression can be configured, using the `MaxAutoSegColumns` configuration parameter. See General Parameters in the Administrator's Guide. Columns are segmented in this order:
  - Short (<8 bytes) data type columns first
  - Larger (> 8 byte) data type columns
  - Up to 32 columns (default for `MaxAutoSegColumns` configuration parameter)
  - If segmenting more than 32 columns, use nested hash function

Auto-projections are defined by the table properties and creation methods, as follows:

If table...	Sort order is...	Segmentation is...
Is created from input stream (COPY or INSERT INTO)	Same as input stream, if sorted.	On PK column (if any), on all FK columns (if any), on the first 31 configurable columns of the table
Is created from CREATE TABLE AS SELECT query	Same as input stream, if sorted. If not sorted, sorted using following rules.	Same segmentation columns f query output is segmented The same as the load, if output of query is unsegmented or unknown

Has FK and PK constraints	FK first, then PK columns	PK columns
Has FK constraints only (no PK)	FK first, then remaining columns	Small data type (< 8 byte) columns first, then large data type columns
Has PK constraints only (no FK)	PK columns	PK columns
Has no FK or PK constraints	On all columns	Small data type (< 8 byte) columns first, then large data type columns

Default automatic projections and segmentation get your database up and running quickly. HP recommends that you start with these projections and then use the Database Designer to optimize your database further. The Database Designer creates projections that optimize your database based on the characteristics of the data and, optionally, the queries you use.

Partition clauses

Creating a table with a partition clause causes all projections anchored on that table to be partitioned according to the clause. For each partitioned projection, logically, there are as many partitions as the number of unique values returned by the partitioned expression applied over the rows of the projection.

All projections include a base projection name prefix, which HP Vertica adds automatically. To ensure a unique projection name, HP Vertica adds a version number within the name if necessary. You use projection names to drop or rename a projection.

Note: Due to the impact on the number of ROS containers, explicit and implicit upper limits are imposed on the number of partitions a projection can have; these limits, however, are detected during the course of operation, such as during COPY.

Creating a partitioned table does not necessarily force all data feeding into a table's projection to be segmented immediately. Logically, the partition clause is applied after the segmented by clause.

Partitioning specifies how data is organized at individual nodes in a cluster and after projection data is segmented; only then is the data partitioned at each node based on the criteria in the partitioning clause.

SQL functions used in the partitioning expression must be immutable, which means they return the exact same value whenever they are invoked and independently of session or environment settings, such as LOCALE. For example, the TO_CHAR function is dependent on locale settings and cannot be used. The RANDOM function also produces different values on each invocation, so cannot be used.

Data loaded with the COPY command is automatically partitioned according to the table's PARTITION BY clause.

For more information, see "Restrictions on Partitioning Expressions" in Defining Partitions in the Administrator's Guide

Examples

The following example creates a table named `Product_Dimension` in the `Retail` schema. HP Vertica creates a default superprojection when data is loaded into the table:

```
=> CREATE TABLE Retail.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
);
```

The following example creates a table named `Employee_Dimension`. HP Vertica creates its associated superprojection in the `Public` schema when data is loaded into the table. Instead of using the sort order from the column definition, the superprojection uses the sort order specified by the ORDER BY clause:

```
=> CREATE TABLE Public.Employee_Dimension (
       Employee_key                   integer PRIMARY KEY NOT NULL,
       Employee_gender                varchar(8) ENCODING RLE,
       Employee_title                 varchar(8),
       Employee_first_name            varchar(64),
       Employee_middle_initial        varchar(8),
       Employee_last_name             varchar(64), )
ORDER BY Employee_gender, Employee_last_name, Employee_first_name;
```

The following example creates a table called `time` and partitions the data by year. HP Vertica creates a default superprojection when data is loaded into the table:

```
=> CREATE TABLE time( ..., date_col date NOT NULL, ...)
=> PARTITION BY extract('year' FROM date_col);
```

The following example creates a table named `location` and partitions the data by state. HP Vertica creates a default superprojection when data is loaded into the table:

```
=> CREATE TABLE location(..., state VARCHAR NOT NULL, ...)
=> PARTITION BY state;
```

The following statement uses `SELECT AS` to create a table called `promo` and load data from columns in the existing `customer_dimension` table in which the customer's `annual_income` is greater than $1,000,000. The data is ordered by state and annual income.

```
=> CREATE TABLE promo
   AS SELECT
      customer_name,
      customer_address,
      customer_city,
      customer_state,
      annual_income
   FROM customer_dimension
   WHERE annual_income>1000000
```

```
   ORDER BY customer_state, annual_income;
```

The following table uses SELECT AS to create a table called promo and load data from the latest committed DML transaction (AT EPOCH LATEST).

```
=> CREATE TABLE promo
   AS AT EPOCH LATEST SELECT
      customer_name,
      customer_address,
      customer_city,
      customer_state,
      annual_income
FROM customer_dimension;
```

The next example creates a new table (`date_dimcopy`) based on the VMart database `date_dimension` table, and including projections.

```
VMart=> create table date_dimcopy like date_dimension including projections;
CREATE TABLE
```

Selecting all from the new table, `date_dimcopy`, lists the same definitions as the original `date_dimension` table.

```
VMart=> select * from date_dimcopy;
 date_key | date | full_date_description | day_of_week | day_number_in_calendar_month |
day_number_in_calendar_year | day_number_in_fiscal_month | day_number_in_fiscal_year |
last_day_in_week_indicator | last_day_in_month_indicator | calendar_week_number_in_year |
calendar_month_name | calendar_month_number_in_year | calendar_year_month | calendar_quarter |
calendar_year_quarter | calendar_half_year | calendar_year | holiday_indicator | weekday_indicator
| selling_season
----------+------+----------------------+-------------+-----------------------------+----------
------------------+--------------------------+-------------------------+------------------
--------+---------------------------+-----------------------------+--------------------+-----
------------------------+--------------------+-----------------+----------------------+-----
--------------+--------------+------------------+------------------+---------------
```

(0 rows)

The following examples illustrate creating a table, and then attempting to create the same table, with and without the IF NOT EXISTS argument, to show the results.

1. Create a simple table:

   ```
   => CREATE TABLE t (a INT);
   CREATE TABLE
   ```

2. Try to create the same table, without IF NOT EXISTS, so a rollback occurs:

   ```
   => CREATE TABLE t (a INT, b VARCHAR(256));
   ROLLBACK:  object "t" already exists
   ```

3. Try again to create the table, using the IF NOT EXISTS clause, so a notice occurs:

   ```
   => CREATE TABLE IF NOT EXISTS t (a INT, b VARCHAR(256));
   NOTICE:  object "t" already exists; nothing was done
   CREATE TABLE
   ```

The IF NOT EXISTS argument is useful for SQL scripts where you want to create a table if it does not already exist, and reuse the existing table if it does.

See Also

Physical Schema in the Concepts Guide

COPY (page 699)

CREATE EXTERNAL TABLE AS COPY (page 714)

CREATE TEMPORARY TABLE (page 791)

DROP_PARTITION (page 473)

DROP PROJECTION (page 818)

DUMP_PARTITION_KEYS (page 479)

DUMP_PROJECTION_PARTITION_KEYS (page 480)

DUMP_TABLE_PARTITION_KEYS (page 481)

PARTITION_PROJECTION (page 515)

PARTITION_TABLE (page 516)

SELECT (page 870)

Partitioning Tables and Auto Partitioning in the Administrator's Guide

Using External Tables in the Administrator's Guide

column-definition (table)

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

Syntax

```
column-name data-type {
... [ column-constraint (page 783) ]
... [ ENCODING encoding-type ]
... [ ACCESSRANK integer ] }
```

Parameters

column-name	Specifies the name of a column to be created or added.
data-type	Specifies one of the following data types: ***BINARY*** (page 72) ***BOOLEAN*** (page 76) ***CHARACTER*** (page 76) ***DATE/TIME*** (page 78) ***NUMERIC*** (page 103) ***DOUBLE PRECISION (FLOAT)*** (page 105) **Tip:** When specifying the maximum column width in a CREATE TABLE statement, use the width in bytes (octets) for any of the string types. Each

	UTF-8 character may require four bytes, but European languages generally require a little over one byte per character, while Oriental language generally require a little under three bytes per character.
column-constraint	Specifies a ***column-constraint*** (page 783) to add on the column.
ENCODING *encoding-type*	[Optional] Specifies the ***type of encoding*** (page 747) to use on the column. By default, the encoding type is auto. **Caution:** The NONE keyword is obsolete.
ACCESSRANK *integer*	[Optional] Overrides the default access rank for a column. This is useful if you want to increase or decrease the speed at which a column is accessed. See Creating and Configuring Storage Locations and Prioritizing Column Access Speed in the Administrator's Guide.

Example

The following example creates a table named `Employee_Dimension` and its associated superprojection in the `Public` schema. The `Employee_key` column is designated as a primary key, and RLE encoding is specified for the `Employee_gender` column definition:

```
=> CREATE TABLE Public.Employee_Dimension (
    Employee_key                    integer PRIMARY KEY NOT NULL,
    Employee_gender                 varchar(8) ENCODING RLE,
    Employee_title                  varchar(8),
    Employee_first_name             varchar(64),
    Employee_middle_initial         varchar(8),
    Employee_last_name              varchar(64),
);
```

column-name-list (table)

Is used to rename columns when creating a table from a query (`CREATE TABLE AS SELECT`). It can also be used to specify the ***encoding-type*** (page 747) and access rank of the column.

Syntax

```
column-name-list
... [ ENCODING encoding-type ]
... [ ACCESSRANK integer ] [ , ... ]
... [ GROUPED ( projection-column-reference [,...] ) ]
```

Parameters

column-name	Specifies the new name for the column.
ENCODING *encoding-type*	Specifies the type of encoding to use on the column. By default, the encoding-type is auto. See ***encoding-type*** (page 747) for a complete list. **Caution:** Using the NONE keyword for strings could negatively affect the behavior of string columns.

ACCESSRANK *integer*	Overrides the default access rank for a column. This is useful if you want to increase or decrease the speed at which a column is accessed. See Creating and Configuring Storage Locations and Prioritizing Column Access Speed in the Administrator's Guide.
GROUPED	Groups two or more columns into a single disk file. This minimizes file I/O for work loads that: ▪ Read a large percentage of the columns in a table. ▪ Perform single row look-ups. ▪ Query against many small columns. ▪ Frequently update data in these columns. If you have data that is always accessed together and it is not used in predicates, you can increase query performance by grouping these columns. Once grouped, queries can no longer independently retrieve from disk all records for an individual column independent of the other columns within the group. **Note:** RLE compression is reduced when a RLE column is grouped with one or more non-RLE columns. When grouping columns you can: ▪ Group some of the columns: ▪ `(a, GROUPED(b, c), d)` ▪ Group all of the columns: ▪ `(GROUPED(a, b, c, d))` ▪ Create multiple groupings in the same projection: ▪ `(GROUPED(a, b), GROUPED(c, d))` **Note:** HP Vertica performs dynamic column grouping. For example, to provide better read and write efficiency for small loads, HP Vertica ignores any projection-defined column grouping (or lack thereof) and groups all columns together by default.

Notes if you are using a query:

Neither the data type nor column constraint can be specified for a column in the column-name-list. These are derived by the columns in the query table identified in the FROM clause. If the query output has expressions other than simple columns (for example, constants, functions, etc) then either an alias must be specified for that expression, or all columns must be listed in the column name list.

You can supply the encoding type and access rank in either the column-name-list or the column list in the query, but not both.

The following statements are both allowed:

```
=> CREATE TABLE promo (state ENCODING RLE ACCESSRANK 1, zip ENCODING RLE, ...)
   AS SELECT * FROM customer_dimension
   ORDER BY customer_state, ... ;

=> CREATE TABLE promo
   AS SELECT * FROM customer_dimension
   ORDER BY customer_state
```

```
   ENCODED BY customer_state ENCODING RLE ACCESSRANK 1, customer_zip ENCODING RLE
...;
```

The following statement is not allowed because encoding is specified in both column-name-list and ENCODED BY clause:

```
=> CREATE TABLE promo (state ENCODING RLE ACCESSRANK 1, zip ENCODING RLE, ...)
   AS SELECT * FROM customer_dimension
   ORDER BY customer_state
   ENCODED BY customer_state ENCODING RLE ACCESSRANK 1, customer_zip ENCODING RLE
...;
```

Example

The following example creates a table named `employee_dimension` and its associated superprojection in the public schema. Note that encoding-type RLE is specified for the `employee_gender` column definition:

```
=> CREATE TABLE public.employee_dimension (
    employee_key             INTEGER PRIMARY KEY NOT NULL,
    employee_gender          VARCHAR(8) ENCODING RLE,
    employee_title           VARCHAR(8),
    employee_first_name      VARCHAR(64),
    employee_middle_initial  VARCHAR(8),
    employee_last_name       VARCHAR(64)
);
```

Using the Vmart schema, the following example creates a table named `promo` from a query that selects data from columns in the `customer_dimension` table. RLE encoding is specified for the state column in the column name list.

```
=> CREATE TABLE promo (
     name,
     address,
     city,
     state ENCODING RLE, income )
   AS SELECT customer_name,
     customer_address,
     customer_city,
     customer_state,
     annual_income
FROM customer_dimension
WHERE annual_income > 1000000
ORDER BY customer_state, annual_income;
```

column-constraint

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

Syntax

```
[ CONSTRAINT constraint-name ] {
...[ NULL | NOT NULL ]
...| PRIMARY KEY
...| REFERENCES table-name [( column [ , ... ] )]
...| UNIQUE
...[ DEFAULT default ]
...[ AUTO_INCREMENT ]
...[ IDENTITY [ ( cache ) | ( start, increment[, cache ] ) ] ]
}
```

Parameters

CONSTRAINT *constraint-name*	[Optional] Assigns a name to the constraint. HP recommends that you name all constraints.
[NULL | NOT NULL]	Specifies that the column can contain null values (NULL, the default), or cannot (NOT NULL). Using 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, since no default value exists.
PRIMARY KEY	Adds a referential integrity constraint defining the column as the primary key.
REFERENCES *table-name* [*column*]	Specifies the table to which the FOREIGN KEY column constraint applies. You can specify a FOREIGN KEY constraint explicitly or just by a reference to the table and column with the PRIMARY KEY. If you omit the column, the default value is the primary key of the referenced table.
UNIQUE	Constrains the data that a column (or group of columns) contains to be unique with respect to all the rows in the table. If you insert or update the column with a duplicate value, HP Vertica does not give an error. To ensure that the column has unique values, run ***ANALYZE_STATISTICS*** (page 440).
DEFAULT *default*	Specifies a default value for a column if the column is used in an INSERT operation and no value is specified for the column. If no value is specified for the column and there is no default, the default is NULL. Default value usage: ▪ A default value can be set for a column of any data type. ▪ The default value can be any variable-free expression, as long as it matches the data type of the column. ▪ Variable-free expressions can contain constants, SQL

	functions, null-handling functions, system information functions, string functions, numeric functions, formatting functions, nested functions, and all HP Vertica-supported operators Default value restrictions: ▪ Expressions can contain only constant arguments. ▪ Subqueries and cross-references to other columns in the table are not permitted in the expression. ▪ The return value of a default expression cannot be NULL. ▪ The return data type of the default expression after evaluation either matches that of the column for which it is defined, or an implicit cast between the two data types is possible. For example, a character value cannot be cast to a numeric data type implicitly, but a number data type can be cast to character data type implicitly. ▪ Default expressions, when evaluated, conform to the bounds for the column. ▪ Volatile functions are not supported when adding columns to existing tables. See ***ALTER TABLE*** (page 672). **Note:** HP Vertica attempts to check the validity of default expressions, but some errors might not be caught until run time.
`AUTO_INCREMENT`	Creates a table column whose values are automatically generated by the database. The initial value of an AUTO_INCREMENT column is always 1. You cannot specify a different initial value. Each time you add a row to the table, HP Vertica increments the column value by 1. You cannot change the value of an AUTO_INCREMENT column.
`IDENTITY [ ( cache ) | ( start, increment[, cache ] ) ]`	Specifies a column whose values are automatically generated by the database. You can use IDENTITY columns as primary keys. IDENTITY column parameters are evaluated in order as follows: ▪ One parameter: Evaluated as *cache*, indicates the number of unique numbers each node allocates per session. *cache* must be a positive integer. Default: 250,000. Minimum: 1. ▪ Two parameters: Evaluated as *start*, *increment*. ▪ *start* specifies the number at which to start the IDENTITY column. Default: 1. ▪ *increment* specifies how much to increment the value from the previous row's value. Default: 1. ▪ Three parameters: Evaluated as *start*, *increment*, *cache*. **Note:** You cannot change the value of an IDENTITY column once the table exists.

Permissions

Table owner or user WITH GRANT OPTION is grantor.

- REFERENCES privilege on table to create foreign key constraints that reference this table

- USAGE privilege on schema that contains the table

Notes

- HP Vertica supports only one IDENTITY or one AUTO_INCREMENT column per table.
- When you use ***ALTER TABLE*** (page 672) to change the table owner, HP Vertica transfers ownership of the associated IDENTITY/AUTO_INCREMENT sequences but not other REFERENCES sequences.
- You can specify a FOREIGN KEY constraint explicitly by using the FOREIGN KEY parameter, or implicitly, using a REFERENCES parameter to the table with the PRIMARY KEY. If you omit the column in the referenced table, HP Vertica uses the primary key:

```
=> CREATE TABLE fact(c1 INTEGER PRIMARY KEY NOT NULL);
=> CREATE TABLE dim (c1 INTEGER REFERENCES fact NOT NULL);
```

- Columns that are given a PRIMARY constraint must also be set NOT NULL. HP Vertica automatically sets these columns to be NOT NULL if you do not do so explicitly.
- HP Vertica supports variable-free expressions in the column DEFAULT clause. See ***COPY*** (page 699) [*Column as Expression*].
- If you are using a CREATE TABLE AS SELECT statement, the column-constraint parameter does not apply. Constraints are set by the columns in the query table identified in the FROM clause.
- An AUTO-INCREMENT or IDENTITY value is never rolled back, even if a transaction that tries to insert a value into a table is not committed.

Example

The following command creates the `store_dimension` table and sets the default column value for `store_state` to MA:

```
=> CREATE TABLE store_dimension (store_state CHAR (2) DEFAULT MA);
```

The following command creates the `public.employee_dimension` table and sets the default column value for `hire_date` to `current_date()`:

```
=> CREATE TABLE public.employee_dimension (hire_date DATE DEFAULT
current_date());
```

The following example uses the IDENTITY column-constraint to create a table with an ID column that has an initial value of 1. It is incremented by 1 every time a row is inserted.

```
=> CREATE TABLE Premium_Customer(
     ID IDENTITY(1,1),
     lname VARCHAR(25),
     fname VARCHAR(25),
     store_membership_card INTEGER
   );
=> INSERT INTO Premium_Customer (lname, fname, store_membership_card )
   VALUES ('Gupta', 'Saleem', 475987);
```

Confirm the row you added and see the ID value:

```
=> SELECT * FROM Premium_Customer;
 ID | lname | fname  | store_membership_card
----+-------+--------+----------------------
```

```
  1 | Gupta | Saleem |                475987
(1 row)
```

Now add another row:

```
=> INSERT INTO Premium_Customer (lname, fname, store_membership_card)
   VALUES ('Lee', 'Chen', 598742);
```

Calling the LAST_INSERT_ID function returns value 2 because you previously inserted a new customer (Chen Lee), and this value is incremented each time a row is inserted:

```
=> SELECT LAST_INSERT_ID();
last_insert_id
---------------
              2
(1 row)
```

View all the ID values in the `Premium_Customer` table:

```
=> SELECT * FROM Premium_Customer;
 ID | lname | fname  | store_membership_card
----+-------+--------+----------------------
  1 | Gupta | Saleem |                475987
  2 | Lee   | Chen   |                598742
(2 rows)
```

The following example uses the AUTO_INCREMENT column-constraint to create a table with an ID column that automatically increments every time a row is inserted.

```
=> CREATE TABLE Premium_Customer(
     ID AUTO_INCREMENT,
     lname VARCHAR(25),
     fname VARCHAR(25),
     store_membership_card INTEGER
   );
=> INSERT INTO Premium_Customer (lname, fname, store_membership_card )
   VALUES ('Gupta', 'Saleem', 475987);
```

Confirm the row you added and see the ID value:

```
=> SELECT * FROM Premium_Customer;
 ID | lname | fname  | store_membership_card
----+-------+--------+----------------------
  1 | Gupta | Saleem |                475987
(1 row)
```

Now add two rows:

```
=> INSERT INTO Premium_Customer (lname, fname, store_membership_card)
   VALUES ('Lee', 'Chen', 598742);
=> INSERT INTO Premium_Customer (lname, fname, store_membership_card)
   VALUES ('Brown', 'John', 642159);

=> SELECT * FROM Premium_Customer;
 ID | lname | fname  | store_membership_card
----+-------+--------+----------------------
  1 | Gupta | Saleem |                475987
  2 | Lee   | Chen   |                598742
  3 | Brown | John   |                642159
```

```
(3 rows)
```

This time the LAST_INSERT_ID returns a value of 3:

```
=> SELECT LAST_INSERT_ID();
 LAST_INSERT_ID
----------------
              3
(1 row)
```

The next examples illustrate the three valid ways to use IDENTITY arguments. The first example uses a cache of 100, and the defaults for start value (1) and increment value (1):

```
=> CREATE TABLE t1(x IDENTITY(100), y INT);
```

The second example specifies the start and increment values as 1, and defaults to a cache value of 250,000:

```
=> CREATE TABLE t2(y IDENTITY(1,1), x INT);
```

The third example specifies start and increment values of 1, and a cache value of 100:

```
=> CREATE TABLE t3(z IDENTITY(1,1,100), zx INT);
```

For additional examples, see ***CREATE SEQUENCE*** (page 765).

table-constraint

Adds a constraint to the metadata of a table. See Adding Constraints in the Administrator's Guide.

Syntax

```
[ CONSTRAINT constraint_name ]
... { PRIMARY KEY ( column [ , ... ] )
... | FOREIGN KEY ( column [ , ... ] ) REFERENCES table [( column [ , ... ] )]
... | UNIQUE ( column [ , ... ] )
... }
```

Parameters

CONSTRAINT *constraint-name*	Assigns a name to the constraint. HP recommends that you name all constraints.
PRIMARY KEY (*column* [, ...])	Adds a referential integrity constraint defining one or more NOT NULL columns as the primary key.
FOREIGN KEY (*column* [, ...])	Adds a referential integrity constraint defining one or more columns as a foreign key.
REFERENCES *table* [(*column* [, ...])]	Specifies the table to which the FOREIGN KEY constraint applies. If you omit the optional *column* definition of the referenced table, the default is the primary key of *table*.
UNIQUE (*column* [, ...])	Specifies that the data contained in a column or a group of columns is unique with respect to all the rows in the table.

Permissions

Table owner or user WITH GRANT OPTION is grantor.

- REFERENCES privilege on table to create foreign key constraints that reference this table
- USAGE privilege on schema that contains the table

Specifying Primary and Foreign Keys

You must define PRIMARY KEY and FOREIGN KEY constraints in all tables that participate in inner joins.

You can specify a foreign key table constraint either explicitly (with the FOREIGN KEY parameter), or implicitly using the REFERENCES parameter to reference the table with the primary key. You do not have to explicitly specify the columns in the referenced table, for example:

```
CREATE TABLE fact(c1 INTEGER PRIMARY KEY);
CREATE TABLE dim (c1 INTEGER REFERENCES fact);
```

Adding Constraints to Views

Adding a constraint to a table that is referenced in a view does not affect the view.

Examples

The Retail Sales Example Database described in the Getting Started Guide 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.

hash-segmentation-clause (table)

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.

Note: Hash segmentation is the preferred method of segmentation. The Database Designer uses hash segmentation by default.

Syntax

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

Parameters

`SEGMENTED BY` *expression*	Can be a general SQL expression. However, HP recommends using the built-in ***HASH*** (page 308) or ***MODULARHASH*** (page 311) functions, specifying table columns as arguments. If you specify only a column name, HP Vertica gives a warning. Choose columns that have a large number of unique data values and acceptable skew in their data distribution. Primary key columns that meet

	the criteria could be an excellent choice for hash segmentation.
`ALL NODES`	Automatically distributes the data evenly across all nodes at the time the projection is created. The ordering of the nodes is fixed.
`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

- Table column names must be used in the expression, not the projection column names.
- To use a `SEGMENTED BY` expression other than HASH or MODULARHASH, the following restrictions apply:
  - All leaf expressions must be either constants or ***column-references*** (see "***Column References***" on page 54) 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 $0 <= x < 2^{63}$ (two to the sixty-third power or 2^63), and values are generally distributed uniformly over that range.
  - If *expression* produces a value outside of the expected range (a negative value for example), no error occurs, and the row is added to the first segment of the projection.
- The hash-segmentation-clause within the CREATE TABLE or CREATE TEMP TABLE statement does not support the OFFSET keyword, which is available in the CREATE PROJECTION command. The OFFSET is set to zero (0).
- When a hash-segmentation-clause is used with KSAFE [k_num], HP Vertica automatically creates k_num+1 buddy projections to meet the K-safety requirement.

Example

This example segments the default superprojection and its buddies for the Public.Employee_Dimension table using HASH segmentation across all nodes based on the Employee_key column:

```
=> CREATE TABLE Public.Employee_Dimension (
       Employee_key                    integer PRIMARY KEY NOT NULL,
       Employee_gender                 varchar(8) ENCODING RLE,
       Employee_title                  varchar(8),
       Employee_first_name             varchar(64),
       Employee_middle_initial         varchar(8),
       Employee_last_name              varchar(64),
)
SEGMENTED BY HASH(Employee_key) ALL NODES;
```

See Also

HASH (page 308) and ***MODULARHASH*** (page 311)

range-segmentation-clause (table)

Allows you to segment a projection based on a known range of values stored in a specific column. Choosing a range of values from a specific column provides even distribution of data across a set of nodes, resulting in optimal query execution.

Note: HP 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
```

Parameters (Range Segmentation)

`SEGMENTED BY` *expression*	Is a single ***column reference*** (see "***Column References***" on page 54) to a column in the column definition of the CREATE TABLE statement. 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. **Note:** 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*	Is 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 `SELECT * FROM NODE_RESOURCES`.
`VALUES LESS THAN` *value*	Specifies that the segment can contain only a range of data values less than *value*. The segments cannot overlap so the minimum value of the range is determined by the *value* of the previous segment (if any).
`VALUES LESS THAN MAXVALUE`	Specifies a sub-range containing data values with no upper limit. MAXVALUE is the maximum value represented by 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 54) for range segmentation. If you want to use a different expression, the following restrictions apply:
  - All leaf expressions must be either constants or column-references 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.
- 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.

See Also

NODE_RESOURCES (page 1047)

CREATE TEMPORARY TABLE

Creates a temporary table.

Syntax

```
CREATE [ GLOBAL | LOCAL ] TEMPORARY | TEMP
... TABLE [ IF NOT EXISTS ] [[db-name.]schema.].table-name {
... ( column-definition (temp table) (page 795) [ , ... ] )
... | [ column-name-list (create table) (page 780) ] }
... [ ON COMMIT { DELETE | PRESERVE } ROWS ]
... [ AS [ { AT EPOCH { integer | LATEST } | AT TIME 'timestamp'} ]
... [ /*+ direct */ ] query ]
... [ ORDER BY table-column [ , ... ] ]
... [ ENCODED BY column-definition [ , ... ] ]
....[ hash-segmentation-clause (page 750)
.....| range-segmentation-clause (page 790)
.....| UNSEGMENTED { NODE node
.....| ALL NODES } ]
....[ KSAFE [ k-num ] ]
....| [ NO PROJECTION ] ]
```

Parameters

GLOBAL	[Optional] Specifies that the table definition is visible to all sessions. This is the default value when creating a temporary table. Temporary table data is visible only to the session that inserts the data into the table.
LOCAL	[Optional] Specifies that the table definition is visible only to the session in which it is created.

`TEMPORARY | TEMP`	Specifies that the table is a temporary table.
`[ IF NOT EXISTS ]`	[Optional] Determines whether the statement generates a NOTICE message or an ERROR if *<object>-name* exists. Using `IF NOT EXISTS` generates a NOTICE if the specified object exists. Omitting the clause generates an error if *<object>-name* exists. Regardless of whether you use `IF NOT EXISTS`, HP Vertica does not create a new object if *<object>-name* exists. For more information, see also ON_ERROR_STOP.
`[[db-name.]schema.]`	Specifies the schema in which to create the table. If you do not specify a schema-name, the statement creates the table in the first schema listed in the current ***search_path*** (page 912). You can specify a schema (and optional database name) only for a global temporary table. Schema-name is not supported for local temporary tables, which are always created in a special schema.
`table-name`	Specifies the name of the temporary table to create.
`column-definition`	Defines one or more columns. See ***column-definition (temp table)*** (page 795).
`column-name-list`	Renames columns when creating a temporary table from a query, such as: `CREATE TEMPORARY TABLE as select...` See ***column-name-list*** (see "***column-name-list (table)***" on page 780).
`ON COMMIT { PRESERVE | DELETE } ROWS`	[Optional] Specifies whether data is transaction- or session-scoped: ▪ DELETE marks the temporary table for transaction-scoped data. HP Vertica truncates the table (delete all its rows) after each commit. DELETE ROWS is the default. ▪ PRESERVE marks the temporary table for session-scoped data, which is preserved beyond the lifetime of a single transaction. HP Vertica truncates the table (delete all its rows) when you terminate a session.
`AS [ { AT EPOCH { integer | LATEST } | AT TIME 'timestamp'} ]`	Specifies the epoch to use with an `AS` clause to query historical: ▪ AT EPOCH LATEST -- data from the latest committed DML transaction ▪ AT EPOCH *integer* -- an epoch specified as an integer ▪ AT TIME 'timestamp' -- an epoch based on its timestamp

`/*+ direct */`	Writes the data directly to disk (ROS) bypassing memory (WOS).
	HP Vertica accepts optional spaces before and after the plus (`+`) sign and the `direct` hint. Space characters between the opening `/*` or the closing `*/` are not permitted. The following directives are all acceptable:
	`/*+direct*/`
	`/* + direct*/`
	`/*+ direct*/`
	`/*+direct */`
	Note: If you create a temporary table using the `direct` hint, you still need to use the `ON COMMIT PRESERVE ROWS` option for the rows.

AS *query*	[Optional.] Creates a new table from the results of a query and populates it with data from the query as long as you also specify ON COMMIT PRESERVE ROWS: CREATE GLOBAL TEMP TABLE temp_table1 ON COMMIT PRESERVE ROWS AS SELECT ...; If you specify ON COMMIT DELETE ROWS, the temporary table is created, but no data is inserted from the query: CREATE GLOBAL TEMP TABLE temp_table1 ON COMMIT DELETE ROWS AS SELECT ...; Column renaming is supported as part of the process: CREATE TEMP TABLE temp-table1 (name, address, ...) AS SELECT customer_name, customer_address ... ;
ORDER BY *table-column*	[Optional] Specifies the superprojection sort order. HP Vertica creates a superprojection automatically for the table when you load data into the table. If you do not use this option to indicate the sort order, such as: ORDER BY col2, col1, col5 the projection is created with the column order specified in the column definition. **Note:** Data is in ascending order only.
ENCODED BY column-definition	[Applicable only with CREATE TEMPORARY TABLE AS query] Specifies the column encoding and/or the access rank for specific columns in the query when you do not use a column-definition to rename columns for the table being created. See ***column-definition (temp table)*** (page 795) for examples. If you rename table columns when creating a temporary table from a query, you can supply the encoding type and access rank in the column name list instead.
hash-segmentation-clause	[Optional] Segments the superprojection based on a built-in hash function that provides even data distribution across nodes, resulting in optimal query execution. See ***hash-segmentation-clause*** (page 750). **Note:** An elastic projection (a segmented projection created when Elastic Cluster is enabled) created with a modularhash segmentation expression uses hash instead.
range-segmentation-clause	[Optional] Segments the superprojection based on a range of values stored in a specific column and chosen to provide even distribution of data across a set of nodes, resulting in optimal query execution. See ***range-segmentation-clause*** (see "***range-segmentation-clause (temp table)***" on page 800).
UNSEGMENTED { NODE *node* | ALL NODES }	[Optional] Lets you specify an unsegmented projection. The default for this parameter is to create an UNSEGMENTED projection on the initiator node. You can optionally use either of the following node specifications: ▪ NODE *node*—Creates an unsegmented projection only on the specified node. Projections for small tables must be UNSEGMENTED. ▪ ALL NODES—Automatically replicates the unsegmented projection on each node. To perform distributed query

	execution, HP Vertica requires an unsegmented copy of each small table superprojection on each node.
KSAFE [*k-num*]	[Optional] Specifies the K-safety level of the automatic projection created for the table. The integer K determines how many unsegmented or segmented buddy projections to create. The value must be greater than or equal to the current K-safety level of the database, and less than the total number of nodes. If you do not specify a KSAFE value, the superprojection is created at the current system K-safety level. For example: K-SAFE 1 **Note:** When a hash-segmentation-clause is used with KSAFE, HP Vertica automatically creates k_num+1 buddy projections to meet the K-safety requirement.
NO PROJECTION	[Optional] Prevents automatically creating a default superprojection for the temporary table until data is loaded. You cannot use the NO PROJECTION option with queries (CREATE TEMPORARY TABLE AS SELECT), ORDER BY, ENCODED BY, KSAFE, ***hash-segmentation-clause*** (page 750), or ***range-segmentation-clause*** (page 790).

Notes

- You cannot add projections to non-empty, session-scoped temporary tables (ON COMMIT PRESERVE ROWS). Make sure that projections exist before you load data. See the "Automatic Projection Creation" in the ***CREATE TABLE*** (page 770) statement.
- Although adding projections is allowed for tables with ON COMMIT DELETE ROWS specified, be aware that you could lose all the data.
- The V_TEMP_SCHEMA namespace is automatically part of the search path. Thus, temporary table names do not need to be preceded with the schema.
- Queries that involve temporary tables have the same restrictions on SQL support as queries that do not use temporary tables.
- Single-node (pinned to the initiator node only) projections are supported.
- AT EPOCH LATEST queries that refer to session-scoped temporary tables work the same as those for transaction-scoped temporary tables. Both return all committed and uncommitted data regardless of epoch. For example, you can commit data from a temporary table in one epoch, advance the epoch, and then commit data in a new epoch.
- Moveout and mergeout operations cannot be used on session-scoped temporary data.
- If you issue the ***TRUNCATE TABLE*** (page 927) statement on a temporary table, only session-specific data is truncated with no affect on data in other sessions.
- The DELETE ... FROM TEMP TABLE syntax does not truncate data when the table was created with PRESERVE; it marks rows for deletion. See ***DELETE*** (page 807) for additional details.
- In general, session-scoped temporary table data is not visible using system (virtual) tables.
- Views are supported for temporary tables.
- ANALYZE_STATISTICS (page 440) is supported on local temporary tables, but not on global temporary tables.
- Table partitions are not supported for temporary tables.

- Temporary tables do not recover. If a node fails, queries that use the temporary table also fail. Restart the session and populate the temporary table.
- Pre-join projections that refer to both temporary and non-temporary tables are not supported.
- You cannot use temporary tables as dimensions when the fact table is non-temporary in pre-join projections. All small tables in a pre-join projection must be at least as persistent as the large table and other small tables. The persistence scale is: Normal > session-scoped temporary table > transaction-scoped temporary table.
  - If the the anchor table of a pre-join projection is a transaction-scoped temp table, you can use any type of table (temp or normal) in the pre-join.
  - If the anchor table of a pre-join is a normal table, you can use only normal tables as dimension tables.
  - If there is a snowflake dimension session-scoped temporary table, its dimension tables may not be a transaction-scoped temporary tables.

See Also

ALTER TABLE (page 672), ***CREATE TABLE*** (page 770), ***DELETE*** (page 807), ***DROP TABLE*** (page 823)

Creating Temporary Tables in the Administrator's Guide

Subqueries in the Programmer's Guide

Transactions in the Concepts Guide

column-definition (temp table)

A column definition specifies the name, data type, default, and other characteristics to be applied to a column.

Syntax

```
column-name data-type [ DEFAULT ] [ NULL | NOT NULL ]
  [ ENCODING encoding-type ] [ ACCESSRANK integer ] ]
```

Parameters

column-name	Specifies the name of the temporary table to be created.
data-type	Specifies one of the following data types: ▪ BINARY ▪ BOOLEAN ▪ CHARACTER ▪ DATE/TIME ▪ NUMERIC

DEFAULT *default*	Specifies a default value for a column if the column is used in an INSERT operation and no value is specified for the column. If no value is specified for the column and there is no default, the default is NULL. Default value usage: ▪ A default value can be set for a column of any data type. ▪ The default value can be any variable-free expression, as long as it matches the data type of the column. ▪ Variable-free expressions can contain constants, SQL functions, null-handling functions, system information functions, string functions, numeric functions, formatting functions, nested functions, and all HP Vertica-supported operators Default value restrictions: ▪ Expressions can contain only constant arguments. ▪ Subqueries and cross-references to other columns in the table are not permitted in the expression. ▪ The return value of a default expression cannot be NULL. ▪ The return data type of the default expression after evaluation either matches that of the column for which it is defined, or an implicit cast between the two data types is possible. For example, a character value cannot be cast to a numeric data type implicitly, but a number data type can be cast to character data type implicitly. ▪ Default expressions, when evaluated, conform to the bounds for the column. ▪ Volatile functions are not supported when adding columns to existing tables. See ***ALTER TABLE*** (page 672). **Note:** HP Vertica attempts to check the validity of default expressions, but some errors might not be caught until run time.
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.
ENCODING *encoding-type*	[Optional] Specifies the ***type of encoding*** (page 747) to use on the column. By default, the encoding type is auto. **Caution:** The NONE keyword is obsolete.
ACCESSRANK *integer*	[Optional] Overrides the default access rank for a column. This is useful if you want to increase or decrease the speed at which a column is accessed. See Creating and Configuring Storage Locations and Prioritizing Column Access Speed in the Administrator's Guide.

column-name-list (temp table)

A column name list is used to rename columns when creating a temporary table from a query (CREATE TEMPORARY TABLE AS SELECT). It can also be used to specify the ***encoding type*** (see "***encoding-type***" on page 747) and access rank of the column.

Syntax

```
column-name-list [ ENCODING encoding-type ] [ ACCESSRANK integer ] [ , ... ]
    [ GROUPED( projection-column-reference [,...] ) ]
```

Parameters

column-name-list	Specifies the new name for the column.
ENCODING *encoding-type*	[Optional] Specifies the type of encoding to use on the column. By default, the encoding-type is auto. See ***encoding type*** (see "***encoding-type***" on page 747) for a complete list. Caution: Using the NONE keyword for strings could negatively affect the behavior of string columns.
ACCESSRANK *integer*	[Optional] Overrides the default access rank for a column. This is useful if you want to increase or decrease the speed at which a column is accessed. See Creating and Configuring Storage Locations and Prioritizing Column Access Speed in the Administrator's Guide.
GROUPED	Groups two or more columns into a single disk file. This minimizes file I/O for work loads that: ▪ Read a large percentage of the columns in a table. ▪ Perform single row look-ups. ▪ Query against many small columns. ▪ Frequently update data in these columns. If you have data that is always accessed together and it is not used in predicates, you can increase query performance by grouping these columns. Once grouped, queries can no longer independently retrieve from disk all records for an individual column independent of the other columns within the group. **Note:** RLE compression is reduced when a RLE column is grouped with one or more non-RLE columns. When grouping columns you can: ▪ Group some of the columns: ▪ (a, GROUPED(b, c), d) ▪ Group all of the columns: ▪ (GROUPED(a, b, c, d)) ▪ Create multiple groupings in the same projection: ▪ (GROUPED(a, b), GROUPED(c, d)) **Note:** HP Vertica performs dynamic column-grouping. For example, to provide better read and write efficiency for small loads, HP Vertica ignores any projection-defined column grouping (or lack thereof) and groups all

	columns together by default.

Notes:

If you are using a CREATE TEMPORARY TABLE AS SELECT statement:

- The data-type cannot be specified for a column in the column name list. It is derived by the column in the query table identified in the FROM clause
- You can supply the encoding type and access rank in either the column name list or the column list in the query, but not both.

  The following statements are both allowed:

```
=> CREATE TEMPORARY TABLE temp_table1 (state ENCODING RLE ACCESSRANK 1, zip
ENCODING RLE, ...)
   AS SELECT * FROM customer_dimension
   ORDER BY customer_state, ... ;
=> CREATE TEMPORARY TABLE temp_table1
   AS SELECT * FROM customer_dimension
   ORDER BY customer_state
  ENCODED BY customer_state ENCODING RLE ACCESSRANK 1, customer_zip ENCODING
RLE ...;
```

  The following statement is not allowed:

```
=> CREATE TEMPORARY TABLE temp_table1 (state ENCODING RLE ACCESSRANK 1, zip
ENCODING RLE, ...)
   AS SELECT * FROM customer_dimension
   ORDER BY customer_state
  ENCODED BY customer_state ENCODING RLE ACCESSRANK 1, customer_zip ENCODING
RLE ...;
```

Example

The following example creates a temporary table named temp_table2 and its associated superprojection. Note that encoding-type RLE is specified for the y column definition:

```
=> CREATE GLOBAL TEMP TABLE temp_table2 (
       x  NUMERIC,
       y  NUMERIC ENCODING RLE,
       b  VARCHAR(8),
       z  VARCHAR(8) );
```

The following example creates a table named temp_table3 from a query that selects data from columns in the customer_dimension table. RLE encoding is specified for the state column in the column name list.

```
=> CREATE TABLE temp_table3 (name, address, city, state ENCODING RLE, income)
   AS SELECT
      customer_name,
      customer_address,
      customer_city,
      customer_state,
      annual_income
   FROM customer_dimension
   WHERE annual_income > 1000000
```

```
ORDER BY customer_state, annual_income;
```

hash-segmentation-clause (temp table)

By default, a superprojection for a temp table is segmented and not pinned. You can choose either hash-segmentation (preferred) or range-segmentation, if you have more than one node.

Hash segmentation allows you to segment a table based on a built-in hash function. The built-in hash function provides even distribution of data across some or all nodes in a cluster, resulting in optimal query execution. Projections created in this manner are not pinned.

Note: Hash segmentation is the preferred method of segmentation. The Database Designer uses hash segmentation by default.

Syntax

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

Parameters

`SEGMENTED BY` *expression*	Can be a general SQL expression. However, HP recommends using the built-in ***HASH*** (page 308) or ***MODULARHASH*** (page 311) functions, specifying table columns as arguments. If you specify only a column name, HP Vertica gives a warning. Choose columns that have a large number of unique data values and acceptable skew in their data distribution. Primary key columns that meet the criteria could be an excellent choice for hash segmentation.
`ALL NODES`	Automatically distributes the data evenly across all nodes at the time the projection is created. The ordering of the nodes is fixed.
`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

- You must use the table column names in the expression, not the projection column names.
- To use a `SEGMENTED BY` expression other than HASH or MODULARHASH, the following restrictions apply:
  - All leaf expressions must be either constants or ***column-references*** (see "***Column References***" on page 54) 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 <= x < 2^{63} (two to the sixty-third power or 2^63), and values are generally distributed uniformly over that range.
  - If *expression* produces a value outside of the expected range (a negative value for example), no error occurs, and the row is added to the first segment of the projection.
- The hash-segmentation-clause within the CREATE TABLE or CREATE TEMP TABLE statement does not support the OFFSET keyword, which is available in the CREATE PROJECTION command. The OFFSET is set to zero (0).

Example

This example segments the default superprojection and its buddies using HASH segmentation based on column 1 (C1).

```
=> CREATE TEMPORARY TABLE ... SEGMENTED BY HASH(C1) ALL NODES;
```

See Also

HASH (page 308) and ***MODULARHASH*** (page 311)

range-segmentation-clause (temp table)

By default, a superprojection for a temp table is segmented and not pinned. If you have more than one node, you can segment the table using either hash-segmentation (preferred), or range-segmentation.

Allows you to segment a projection based on a known range of values stored in a specific column. Choosing a range of values from a specific column provides even distribution of data across a set of nodes, resulting in optimal query execution.

Projections created in this manner are not pinned.

Note: HP 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
```

Parameters (Range Segmentation)

`SEGMENTED BY` *expression*	Is a single ***column reference*** (see "***Column References***" on page 54) to a column in the SELECT list of the CREATE PROJECTION statement. 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. **Note:** 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*	Is 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 SELECT * FROM NODE_RESOURCES.
VALUES LESS THAN *value*	Specifies that the segment can contain only a range of data values less than *value*. The segments cannot overlap so the minimum value of the range is determined by the *value* of the previous segment (if any).
VALUES LESS THAN MAXVALUE	Specifies a sub-range containing data values with no upper limit. MAXVALUE is the maximum value represented by 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 54) for range segmentation. If you want to use a different expression, the following restrictions apply:
  - All leaf expressions must be either constants or column-references 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.
- 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.

See Also

NODE_RESOURCES (page 1047)

CREATE USER

Adds a name to the list of authorized database users.

Syntax

```
CREATE USER name
... [ ACCOUNT {LOCK | UNLOCK} ]
... [ IDENTIFIED BY 'password' ]
... [ MEMORYCAP {'memory-limit' | NONE} ]
... [ PASSWORD EXPIRE ]
```

```
... [ PROFILE {profile | DEFAULT} ]
... [ RESOURCE POOL pool-name ]
... [ RUNTIMECAP {'time-limit' | NONE} ]
... [ TEMPSPACECAP {'space-limit' | NONE} ]
... [ SEARCH_PATH {schema[,schema2,...]} | DEFAULT ]
```

Parameters

name	Specifies the name of the user to create; names that contain special characters must be double-quoted. **Tip:** HP Vertica database user names are logically separate from user names of the operating system in which the server runs. If all the users of a particular server also have accounts on the server's machine, it makes sense to assign database user names that match their operating system user names. However, a server that accepts remote connections could have many database users who have no local operating system account, and in such cases there need be no connection between database user names and OS user names.
ACCOUNT LOCK | UNLOCK	Locks or unlocks the a user's access to the database. UNLOCK is the default for new users, so the keyword is optional. You'll most commonly use UNLOCK with the ALTER USER command. Specifying LOCK prevents a new user from logging in, which might be useful if you want to create an account for users who don't need access yet. **Tip:** A superuser can automate account locking by setting a maximum number of failed login attempts through the CREATE PROFILE (page 739) statement. See also Profiles in the Administrator's Guide.
IDENTIFIED BY *'password'*	Sets the new user's password. Supplying an empty string for *password* creates a user without a password, as does omitting the IDENTIFIED BY *'password'* clause. If a user does not have a password, he or she will not be prompted for one when connecting. Providing a password using the IDENTIFIED BY clause requires that the given password conform to the password complexity policy set by the user's profile. User profiles are either specified with the PROFILE parameter, or associated with a default profile if a superuser omits the PROFILE parameter. See Password Guidelines and Creating a Database Name and Password for password policies.
PASSWORD EXPIRE	Expires the user's password immediately. The user will be forced to change the password when he or she next logs in. The grace period setting (if any) in the user's profile is overridden. **Note:** PASSWORD EXPIRE has no effect when using external password authentication methods such as LDAP or Kerberos.
MEMORYCAP '*memory-limit*' | NONE	Limits the amount of memory that the user's requests can use. This value is a number representing the amount of space, followed by a unit (for example, '10G'). The unit can be one of the

	following: ▪ % percentage of total memory available to the Resource Manager. (In this case value for the size must be 0-100) ▪ K Kilobytes ▪ M Megabytes ▪ G Gigabytes ▪ T Terabytes Setting this value to `NONE` means the user's sessions have no limits on memory use. This is the default value.
`PROFILE` *`profile`* | `DEFAULT`	Assigns the user to the profile named *`profile`*. Profiles set the user's password policy. See Profiles in the Administrator's Guide for details. Using the value DEFAULT here assigns the user to the default profile. If this parameter is omitted, the user is assigned to the default profile.
`RESOURCE POOL` *`pool-name`*	Sets the name of the resource pool from which to request the user's resources. This command creates a usage grant for the user on the resource pool unless the resource pool is publicly usable.
`RUNTIMECAP '`*`time-limit`*`' | NONE`	Sets the maximum amount of time any of the user's queries can execute. time-limit is an interval, such as '1 minute' or '100 seconds' (see ***Interval Values*** (page 37) for details). The maximum duration allowed is one year. Setting this value to NONE means there is no time limit on the user's queries. If `RUNTIMECAP` is also set for the resource pool or session, HP Vertica always uses the shortest limit.
`TEMPSPACECAP '`*`space-limit`*`' | NONE`	Limits the amount of temporary file storage the user's requests can use. This parameter's value has the same format as the `MEMORYCAP` value.
`SEARCH_PATH` *`schema`*`[,`*`schema2`*`,...] | DEFAULT`	Sets the user's default search path that tells HP Vertica which schemas to search for unqualified references to tables and UDFs. See Setting Search Paths in the Administrator's Guide for an explanation of the schema search path. The DEFAULT keyword sets the search path to: `"$user", public, v_catalog, v_monitor, v_internal`

Permissions

Must be a superuser to create a user.

Notes

- User names created with double-quotes are case sensitive. For example:
  ```
  => CREATE USER "FrEd1";
  ```
  In the above example, the login name must be an exact match. If the user name was created without double-quotes (for example, FRED1), then the user can log in as FRED1, FrEd1, fred1, and so on.
  Note: ***ALTER USER*** (page 679) and ***DROP USER*** (page 826) are case-*in*sensitive.

- 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.
- You can change a user password by using the ALTER USER statement. If you want to configure a user to not have any password authentication, you can set the empty password '' in CREATE or ALTER USER statements, or omit the IDENTIFIED BY parameter in CREATE USER.
- By default, users have the right to create temporary tables in the database.

Examples

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

See Also

ALTER USER (page 679)

DROP USER (page 826)

Managing Workloads in the Administrator's Guide

Setting a Run-Time Limit for Queries

CREATE VIEW

Defines a new view. Views are read only. You cannot perform insert, update, delete, or copy operations on a view.

Syntax

```
CREATE [ OR REPLACE ] VIEW viewname [ ( column-name [, ...] ) ] AS query ]
```

Parameters

`[ OR REPLACE ]`	When you supply this option, HP Vertica overwrites any existing view with the name *viewname*. If you do not supply this option and a view with that name already exists, CREATE VIEW returns an error.
viewname	Specifies the name of the view to create. The view name must be unique. Do not use the same name as any table, view, or projection within the database. If the view name is not provided, the user name is used as the view name.
column-name	[Optional] Specifies the list of names to be used as column names for the view. Columns are presented from left to right in the order given. If not specified, HP Vertica automatically deduces the column names from the query.
query	Specifies the query that the view executes. HP Vertica also uses the query to deduce the list of names to be used as columns names for the view if they are not specified. Use a ***SELECT*** (page 870) statement to specify the query. The SELECT statement can refer to tables, temp tables, and other views.

Permissions

To create a view, the user must be a superuser or have CREATE privileges on the schema in which the view is created.

Privileges required on base objects for the view owner must be directly granted, not through roles:

- If a non-owner runs a SELECT query on the view, the view owner must also have SELECT ... WITH GRANT OPTION privileges on the view's base tables or views. This privilege must be directly granted to the owner, rather than through a role.
- If a view owner runs a SELECT query on the view, the owner must also have SELECT privilege directly granted (not through a role) on a view's base objects (table or view).

Transforming a SELECT Query to Use a View

When HP Vertica processes a query that contains a view, the view is treated as a subquery because the view name is replaced by the view's defining query. The following example defines a view (ship) and illustrates how a query that refers to the view is transformed.

Create a new view, called ship:

```
CREATE VIEW ship AS SELECT * FROM public.shipping_dimension;
```

Review the original query, and then the transformed subquery version:

```
SELECT * FROM ship;
```

```
SELECT * FROM (SELECT * FROM public.shipping_dimension) AS ship;
```

Dropping a view

Use the ***DROP VIEW*** (page 827) statement to drop a view. Only the specified view is dropped. HP Vertica does not support CASCADE functionality for views, and it does not check for dependencies. Dropping a view causes any view that references it to fail.

Renaming a view

Use the ***ALTER VIEW*** (page 681) statement to rename a view.

Example

```
=> CREATE VIEW myview AS
   SELECT SUM(annual_income), customer_state
   FROM public.customer_dimension
   WHERE customer_key IN
     (SELECT customer_key
      FROM store.store_sales_fact)
   GROUP BY customer_state
   ORDER BY customer_state ASC;
```

The following example uses the *myview* view with a WHERE clause that limits the results to combined salaries of greater than 2,000,000,000.

```
=> SELECT * FROM myview WHERE SUM > 2000000000;
```

```
     SUM     | customer_state
-------------+----------------
  2723441590 | AZ
 29253817091 | CA
  4907216137 | CO
  3769455689 | CT
  3330524215 | FL
  4581840709 | IL
  3310667307 | IN
  2793284639 | MA
  5225333668 | MI
  2128169759 | NV
  2806150503 | PA
  2832710696 | TN
 14215397659 | TX
  2642551509 | UT
(14 rows)
```

See Also

ALTER VIEW (page 681)

SELECT (page 870)

DROP VIEW (page 827), ***GRANT (View)*** (page 845)

REVOKE (View) (page 866)

DELETE

Marks tuples as no longer valid in the current epoch, marking the records for deletion in the WOS, rather than deleting data from disk storage. By default, delete uses the WOS and if the WOS fills up, overflows to the ROS. You cannot delete records from a projection.

Syntax

```
DELETE [ /*+ direct */ ] [ /*+ label(label-name)*/ ]
... FROM [[db-name.]schema.]table
... WHERE clause (page 901)
```

Parameters

`/*+ direct */`	Writes the data directly to disk (ROS) bypassing memory (WOS). **Note:** If you delete using the `direct` hint, you still need to issue a COMMIT or ROLLBACK command to finish the transaction.
`/*+ label (label-name)*/`	Passes a user-defined label to a query as a hint, letting you quickly identify labeled queries for profiling and debugging. See Query Labeling in the Administrator's Guide.
`[[db-name.]schema.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`). When using more than one schema, specify the schema that contains the table in your `DELETE` statement.
table	Specifies the name of a base table or temporary table.

Permissions

Table owner or user with GRANT OPTION is grantor.

- DELETE privilege on table
- USAGE privilege on schema that contains the table

- SELECT privilege on the referenced table when executing a DELETE statement that references table column values in a WHERE or SET clause

Using the DELETE Statement

DELETE statements support subqueries and joins, which is useful for deleting values in a table based on values that are stored in other tables. See the Examples section below.

The delete operation deletes rows that satisfy the `WHERE` clause from the specified table. On completing successfully, the DELETE statement returns a count of the number of deleted rows. A count of 0 is not an error, but indicates that no rows matched the condition. If no `WHERE` clause exists, the statement deletes all table rows, resulting in an empty table.

To remove all rows from a temporary table, use a `DELETE` statement with no `WHERE` clause. In this case, the rows are not stored in the system, which improves performance. The `DELETE` statement removes all rows, but preserves the columns, projections, and constraints, thus making it easy to re-populate the table.

If you include a `WHERE` clause with a `DELETE` statement on a temporary table, `DELETE` behaves the same as for base tables, marking all delete vectors for storage, and you lose any performance benefits. If the delete operation succeeds on temporary tables, you cannot roll back to a prior savepoint.

To truncate a temporary table, without ending the transaction, use `DELETE FROM temp_table.`

Examples

To use the `DELETE` or `UPDATE` (page 929) statements with a ***WHERE Clause*** (page 901), the user must have both `SELECT` (page 870) and `DELETE` privileges on the table.

The following command truncates a temporary table called `temp1`:

```
=> DELETE FROM temp1;
```

The following command deletes all records from base table `T` where `C1 = C2 - C1`.

```
=> DELETE FROM T WHERE C1=C2-C1;
```

The following command deletes all records from the customer table in the retail schema where the state attribute is in MA or NH:

```
=> DELETE FROM retail.customer WHERE  state IN ('MA', 'NH');
```

For examples on how to nest a subquery within a DELETE statement, see Subqueries in UPDATE and DELETE in the Programmer's Guide.

See Also

DROP TABLE (page 823) and ***TRUNCATE TABLE*** (page 927)

Deleting Data and Best Practices for DELETE and UPDATE in the Administrator's Guide

DISCONNECT

Closes a previously-established connection to another HP Vertica database. You must have previously used the CONNECT statement to perform a COPY FROM VERTICA or EXPORT TO VERTICA statement.

Syntax

```
DISCONNECT database-name
```

Parameters

database-name	The name of the database whose connection to close.

Permissions

No special permissions required.

Example

```
=> DISCONNECT ExampleDB;
DISCONNECT
```

See Also

CONNECT (page 697)

COPY FROM VERTICA (page 711)

EXPORT TO VERTICA (page 829)

DROP AGGREGATE FUNCTION

Drops a User Defined Aggregate Function (UDAF) from the HP Vertica catalog.

Syntax

```
DROP AGGREGATE FUNCTION [[db-name.]schema.]function-name [, ...]
... ( [ [ argname ] argtype  [, ...] ] )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, use a table and

	column (mytable.column1), a schema, table, and column (myschema.mytable.column1), and, as full qualification, a database, schema, table, and column (mydb.myschema.mytable.column1).
function-name	Specifies a name of the SQL function to drop. If the function name is schema-qualified, the function is dropped from the specified schema (as noted above).
argname	Specifies the name of the argument, typically a column name.
argtype	Specifies the data type for argument(s) that are passed to the function. Argument types must match HP Vertica type names. See ***SQL Data Types*** (page 71).

Notes

- To drop a function, you must specify the argument types because there could be several functions that share the same name with different parameters.
- HP Vertica does not check for dependencies, so if you drop a SQL function where other objects reference it (such as views or other SQL functions), HP Vertica returns an error when those objects are used and not when the function is dropped.

Permissions

Only the superuser or owner can drop the function.

Example

The following command drops the `ag_avg` function:

```
=> DROP AGGREGATE FUNCTION ag_avg(numeric);
DROP AGGREGATE FUNCTION
```

See Also

ALTER FUNCTION (page 656)

CREATE AGGREGATE FUNCTION

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Using SQL Macros, Using User Defined Functions and Developing a User Defined Aggregate Function in the Programmer's Guide

DROP FUNCTION

Drops a SQL function or User Defined Function (UDF) from the HP Vertica catalog.

Syntax

```
DROP FUNCTION [[db-name.]schema.]function-name [, ...]
... ( [ [ argname ] argtype  [, ...] ] )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
function-name	Specifies a name of the SQL function to drop. If the function name is schema-qualified, the function is dropped from the specified schema (as noted above).
argname	Specifies the name of the argument, typically a column name.
argtype	Specifies the data type for argument(s) that are passed to the function. Argument types must match HP Vertica type names. See ***SQL Data Types*** (page 71).

Permissions

Must be a superuser, function owner, or schema owner to drop a function.

Notes

- To drop a function, you must specify the argument types because there could be several functions that share the same name with different parameters.
- HP Vertica does not check for dependencies, so if you drop a SQL function where other objects reference it (such as views or other SQL functions), HP Vertica returns an error when those objects are used and not when the function is dropped.

Example

The following command drops the `zerowhennull` function in the `macros` schema:

```
=>  DROP FUNCTION macros.zerowhennull(x INT);
```

```
DROP FUNCTION
```

See Also

ALTER FUNCTION (page 656)

CREATE FUNCTION (page 722)

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Using SQL Macros in the Programmer's Guide

DROP SOURCE

Drops a User Defined Load Source function from the HP Vertica catalog.

Syntax

```
DROP SOURCE [[db-name.]schema.]source-name();
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, use a table and column (mytable.column1), a schema, table, and column (myschema.mytable.column1), and, as full qualification, a database, schema, table, and column (mydb.myschema.mytable.column1).
source-name	Specifies the name of the source function to drop. If the function name is schema-qualified, the function is dropped from the specified schema (as noted above). You must include empty parenthesis after the function name.

Permissions

Only the superuser or owner can drop the source function.

Example

The following command drops the `curl` source function:

```
=> drop source curl();
DROP SOURCE
```

See Also

ALTER FUNCTION (page 656)

CREATE SOURCE

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Developing User Defined Load (UDL) Functions in the Programmer's Guide

DROP FILTER

Drops a User Defined Load Filter function from the HP Vertica catalog.

Syntax

```
DROP FILTER [[db-name.]schema.]filter-name();
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, use a table and column (mytable.column1), a schema, table, and column (myschema.mytable.column1), and, as full qualification, a database, schema, table, and column (mydb.myschema.mytable.column1).
filter-name	Specifies the name of the filter function to drop. If the function name is schema-qualified, the function is dropped from the specified schema (as noted above). You must include empty parenthesis after the function name.

Permissions

Only the superuser or owner can drop the filter function.

Example

The following command drops the `Iconverter` filter function::

```
=> drop filter Iconverter();
DROP FILTER
```

See Also

ALTER FUNCTION (page 656)

CREATE FILTER

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Developing User Defined Load (UDL) Functions in the Programmer's Guide

DROP PARSER

Drops a User Defined Load Parserfunction from the HP Vertica catalog.

Syntax

```
DROP PARSER[[db-name.]schema.]parser-name();
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, use a table and column (mytable.column1), a schema, table, and column (myschema.mytable.column1), and, as full qualification, a database, schema, table, and column (mydb.myschema.mytable.column1).
parser-name	Specifies the name of the parser function to drop. If the function name is schema-qualified, the function is dropped from the specified schema (as noted above). You must include empty parenthesis after the function name.

Permissions

Only the superuser or owner can drop the parser function.

Example

The following command drops the BasicIntegerParser parser function:

```
=> DROP PARSER BasicIntegerParser();
DROP PARSER
```

See Also

ALTER FUNCTION (page 656)

CREATE PARSER

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

V_CATALOG.USER_FUNCTIONS (page 982)

Developing User Defined Load (UDL) Functions in the Programmer's Guide

DROP LIBRARY

Removes a shared library from the database. The user defined function s(UDFs) in the library are no longer available. See Using User Defined Functions in the Programmer's Guide for details.

Syntax

```
DROP LIBRARY [[db-name.]schema.]library_name [CASCADE]
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full

	qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
library_name	The name of the library to drop. This must be the same name given to ***CREATE LIBRARY*** (page 735) to load the library.
`[CASCADE]`	Drops any functions that have been defined using the library. This statement fails if CASCADE is not specified and there is one or more UDFs that have been defined using the library.

Permissions

Must be a superuser to drop a library.

Notes

The library file is deleted from the managed directories on the HP Vertica nodes.

Example

To drop the library named MyFunctions:

```
=> DROP LIBRARY MyFunctions CASCADE;
```

DROP NETWORK INTERFACE

Removes a network interface from HP Vertica. You can use the CASCADE option to also remove the network interface from any node definition. (See Identify the Database or Node(s) used for Import/Export for more information.)

Syntax

```
DROP NETWORK INTERFACE network-interface-name [CASCADE]
```

Parameters

The parameters are defined as follows:

network-interface-name	The network interface you want to remove.

Permissions

Must be a superuser to drop a network interface.

DROP PROCEDURE

Removes an external procedure from HP Vertica.

Syntax

```
DROP PROCEDURE [[db-name.]schema.]name ( [ argname ] [ argtype ] [,...] )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
name	Specifies the name of the procedure to be dropped.
argname	[Optional] The argument name or names used when creating the procedure.
argtype	Optional unless an argument type was specified when the procedure was created. The argument type or types used when creating the procedure.

Permissions

Must be a superuser or procedure owner and have USAGE privilege on schema that contain procedure or be the schema owner.

Note

- Only the database superuser can drop procedures.
- Only the reference to the procedure is removed. The external file remains in the *<database>/procedures* directory on each node in the database.

Example

```
=> DROP PROCEDURE helloplanet(arg1 varchar);
```

See Also

CREATE PROCEDURE (page 737)

DROP PROFILE

Removes a profile from the database. Only the superuser can drop a profile.

Syntax

```
DROP PROFILE name [, ...] [ CASCADE ]
```

Parameters

name	The name of one or more profiles (separated by commas) to be removed.

CASCADE	Moves all users assigned to the profile or profiles being dropped to the DEFAULT profile. If you do not include CASCADE in the DROP PROFILE command and a targeted profile has users assigned to it, the command returns an error.

Permissions

Must be a superuser to drop a profile.

Notes

You cannot drop the DEFAULT profile.

See Also

ALTER PROFILE (page 660)

CREATE PROFILE (page 739)

DROP PROJECTION

Marks a projection to be dropped from the catalog so it is unavailable to user queries.

Syntax

```
DROP PROJECTION { base-projname |  projname-node [ , ...] }
... [ RESTRICT | CASCADE ]
```

Parameters

base-projname	Specifies the base projection to drop, along with all replicated buddies on all nodes simultaneously. When using more than one schema, specify the schema containing the base projection. projname can be 'projname' or 'schema.projname'.
projname-node	Drops only the specified projection on the specified node.When using more than one schema, specify the schema that contains the projection to drop. projname can be 'projname' or 'schema.projname'.
RESTRICT	Drops the projection only if it does not contain any objects. RESTRICT is the default.
CASCADE	Drops the projection even if it contains one or more objects.

Permissions

To drop a projection, the user must own the anchor table for which the projection was created and have USAGE privilege on schema that contains the projection OR be the schema owner.

Notes

To prevent data loss and inconsistencies, tables must contain one superprojection, so DROP PROJECTION fails if a projection is the table's only superprojection. In such cases, use the DROP TABLE command.

To a drop all projections:

```
=> DROP PROJECTION prejoin_p;
```

To drop the projection on node 2:

```
=> DROP PROJECTION prejoin_p_site02;
```

Alternatively, you can issue a command like the following, which drops projections on a particular schema:

```
=> DROP PROJECTION schema1.fact_proj_a, schema1.fact_proj_b;
```

If you want to drop a set of buddy projections, you could be prevented from dropping them individually using a sequence of DROP PROJECTION statements due to K-safety violations. See ***MARK_DESIGN_KSAFE*** (page 510) for details.

See Also

CREATE PROJECTION (page 742), ***DROP TABLE*** (page 823), ***GET_PROJECTIONS*** (page 499), ***GET_PROJECTION_STATUS*** (page 498), and ***MARK_DESIGN_KSAFE*** (page 510)

Adding Nodes in the Administrator's Guide

DROP RESOURCE POOL

Drops a user-created resource pool. All memory allocated to the pool is returned back to the `GENERAL` ***pool*** (page 757).

Syntax

`DROP RESOURCE POOL` *`pool-name`*

Parameters

`pool-name`	Specifies the name of the resource pool to be dropped.

Permissions

Must be a superuser to drop a resource pool.

Transferring Resource Requests

Any requests queued against the pool are transferred to the `GENERAL` pool according to the priority of the pool compared to the `GENERAL` pool. If the pool's priority is higher than the `GENERAL` pool, the requests are placed at the head of the queue; otherwise the requests are placed at the end of the queue.

Any users who are using the pool are switched to use the `GENERAL` pool with a NOTICE:

```
NOTICE:  Switched the following users to the General pool: username
```

DROP RESOURCE POOL returns an error if the user does not have permission to use the GENERAL pool. Existing sessions are transferred to the GENERAL pool regardless of whether the session's user has permission to use the GENERAL pool. This can result in additional user privileges if the pool being dropped is more restrictive than the GENERAL pool. To prevent giving users additional privileges, follow this procedure to drop restrictive pools:

1. ***Revoke the permissions on the pool*** (page 857) for all users.
2. Close any sessions that had permissions on the pool.
3. Drop the resource pool.

Example

The following command drops the resource pool that was created for the CEO:

```
=> DROP RESOURCE POOL ceo_pool;
```

See Also

ALTER RESOURCE POOL (page 663)

CREATE RESOURCE POOL (page 753)

Managing Workloads in the Administrator's Guide

DROP ROLE

Removes a role from the database. Only the database superuser can drop a role.

Use the CASCADE option to drop a role that is assigned to one or more users or roles.

Syntax

```
DROP ROLE role [CASCADE];
```

Parameters

role	The name of the role to drop
CASCADE	Revoke the role from users and other roles before dropping the role

Permissions

Must be a superuser to drop a role.

Example

```
=> DROP ROLE appadmin;
NOTICE:  User bob depends on Role appadmin
ROLLBACK:  DROP ROLE failed due to dependencies
DETAIL:  Cannot drop Role appadmin because other objects depend on it
```

```
HINT:  Use DROP ROLE ... CASCADE to remove granted roles from the dependent
users/roles
=> DROP ROLE appadmin CASCADE;
DROP ROLE
```

See Also

ALTER ROLE RENAME (page 667)

CREATE ROLE (page 764)

DROP SCHEMA

Permanently removes a schema from the database. Be sure that you want to remove the schema before you drop it, because DROP SCHEMA is an irreversible process. Use the CASCADE parameter to drop a schema containing one or more objects.

Syntax

```
DROP SCHEMA [db-name.]schema [, ...] [ CASCADE | RESTRICT ]
```

Parameters

[db-name.]	[Optional] Specifies the current database name. Using a database name prefix is optional, and does not affect the command in any way. You must be connected to the specified database.
schema	Specifies the name of the schema to drop.
CASCADE	Drops the schema even if it contains one or more objects.
RESTRICT	Drops the schema only if it does not contain any objects (the default).

Privileges

Schema owner

Restrictions

You cannot drop the PUBLIC schema.

Notes

- A schema owner can drop a schema even if the owner does not own all the objects within the schema. All the objects within the schema are also dropped.
- If a user is accessing an object within a schema that is in the process of being dropped, the schema is not deleted until the transaction completes.
- Canceling a DROP SCHEMA statement can cause unpredictable results.

Examples

The following example drops schema S1 only if it doesn't contain any objects:

```
=> DROP SCHEMA S1;
```

The following example drops schema S1 whether or not it contains objects:

```
=> DROP SCHEMA S1 CASCADE;
```

DROP SEQUENCE

Removes the specified sequence number generator.

Syntax

```
DROP SEQUENCE [[db-name.]schema.]name [ , ... ]
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`). When using more than one schema, specify the schema that contains the sequence to drop.
name	Specifies the name of the sequence to drop.

Permissions

To drop a sequence, the user must be the sequence owner or schema owner.

Notes

- For sequences specified in a table's default expression, the default expression fails the next time you try to load data. HP Vertica does not check for these instances.
- The CASCADE keyword is not supported. Sequences used in a default expression of a column cannot be dropped until all references to the sequence are removed from the default expression.

Example

The following command drops the sequence named `sequential`.

```
=> DROP SEQUENCE sequential;
```

See Also

ALTER SEQUENCE (page 669)

CREATE SEQUENCE (page 765)

CURRVAL (page 353)

GRANT (Sequence) (page 838)

NEXTVAL (page 351)

Using Sequences and Sequence Privileges in the Administrator's Guide

DROP SUBNET

Removes a subnet from HP Vertica. You can use the CASCADE option to also remove the subnet from any database definition. (See Identify the Database or Node(s) used for Import/Export for more information.)

Syntax

```
DROP SUBNET subnet-name [CASCADE]
```

Parameters

The parameters are defined as follows:

subnet-name	The subnet you want to remove.

If you remove a subnet, be sure your database is not configured to allow export on the public subnet. (See Identify the Database or Node(s) used for Import/Export for more information.)

Permissions

Must be a superuser to drop a subnet.

DROP TABLE

Removes a table and, optionally, its associated projections.

Syntax

```
DROP TABLE [ IF EXISTS ] [[db-name.]schema.]table [, ...] [ CASCADE ]
```

Parameters

[IF EXISTS]	If specified, DROP TABLE does not report an error if one or of the tables to be dropped does not exist. This clause is useful in SQL scripts where you want to drop a table if it exists before recreating it.

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the table name. When using more than one schema, specify the schema that contains the table in the DROP TABLE statement.
CASCADE	[Optional] Drops all projections that include the table. **NOTE:** You cannot use the CASCADE option when dropping external tables. For more information, see External Table Support in the Administrator's Guide.

Permissions

Table owner with USAGE privilege on schema that contains the table or schema owner

Note: The schema owner can drop a table but cannot truncate a table.

Notes

- Canceling a DROP TABLE statement can cause unpredictable results.
- Make sure that all other users have disconnected before using DROP TABLE.
- Views that reference a table that is dropped and then replaced by another table with the same name continue to function and use the contents of the new table, as long as the new table contains the same columns and column names.
- Use the multiple projection syntax in K-safe clusters.

Examples

If you try to drop a table with associated projections, a message listing the projections displays. 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.
=> DROP TABLE d1 CASCADE;
```

```
DROP TABLE
=> CREATE TABLE mytable (a INT, b VARCHAR(256));
CREATE TABLE
=> DROP TABLE IF EXISTS mytable;
DROP TABLE
=> DROP TABLE IF EXISTS mytable; -- Doesn't exist
NOTICE:  Nothing was dropped
DROP TABLE
```

See Also

DELETE (page 807)

DROP PROJECTION (page 818)

TRUNCATE TABLE (page 927)

Adding Nodes and Deleting Data in the Administrator's Guide

DROP TRANSFORM FUNCTION

Drops a User Defined Transform Function (UDTF) from the HP Vertica catalog.

Syntax

```
DROP TRANSFORM FUNCTION [[db-name.]schema.]name [, ...]
... ( [ [ argname ] argtype  [, ...] ] )
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
name	Specifies the name of the transform function to drop.
argname	Specifies the name of the argument, typically a column name.
argtype	Specifies the data type for argument(s) that are passed to the function. Argument types must match HP Vertica type names. See ***SQL Data Types*** (page 71).

Permissions

Must be a superuser, function owner, or schema owner to drop a function.

Notes

To drop a transform function, you must specify the argument types because there could be several functions that share the same name with different parameters.

Example

The following command drops the `tokenize` UDTF in the `macros` schema:

```
=> DROP TRANSFORM FUNCTION macros.tokenize(varchar);
DROP TRANSFORM FUNCTION
```

See Also

CREATE TRANSFORM FUNCTION (page 734)

Using User Defined Transforms in the Programmer's Guide

DROP USER

Removes a name from the list of authorized database users.

Syntax

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

Parameters

name	Specifies the name or names of the user to drop.
CASCADE	[Optional] Drops all user-defined objects created by the user dropped, including schema, table and all views that reference the table, and the table's associated projections.

Permissions

Must be a superuser to drop a user.

Examples

`DROP USER <name>` fails if objects exist that were created by the user, such as schemas, tables and their associated projections:

```
=> DROP USER user1;
   NOTICE:  Table T_tbd1 depends on User user1
   ROLLBACK:  DROP failed due to dependencies
   DETAIL:  Cannot drop User user1 because other objects depend on it
   HINT:  Use DROP ... CASCADE to drop the dependent objects too
```

`DROP USER <name> CASCADE` succeeds regardless of any pre-existing user-defined objects. The statement forcibly drops all user-defined objects, such as schemas, tables and their associated projections:

```
=> DROP USER user1 CASCADE;
```

Caution: Tables owned by the user being dropped cannot be recovered after you issue `DROP USER CASCADE`.

`DROP USER` <username> succeeds if no user-defined objects exist (no schemas, tables or projections defined by the user):

```
=> CREATE USER user2;
=> DROP USER user2;
```

See Also

ALTER USER (page 679)

CREATE USER (page 801)

DROP VIEW

Removes the specified view. This statement drops only the specified view. HP Vertica does not support a cascade parameter for views, nor does it check for dependencies on the dropped view. Any other views that reference the dropped view will fail.

If you drop a view and replace it with another view or table with the same name and column names, any views dependent on the dropped view continue to function using the new view. If you change the column data type in the new view, the server coerces the old data type to the new one if possible. Otherwise, it returns an error.

Syntax

```
DROP VIEW name [ , ... ]
```

Parameters

name	Specifies the name of the view to drop.

Permissions

To drop a view, the user must be the view owner and have USAGE privileges on the schema or be the schema owner.

Examples

```
=> DROP VIEW myview;
```

END

Ends the current transaction and makes all changes that occurred during the transaction permanent and visible to other users.

Syntax

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

Parameters

WORK | TRANSACTION	Have no effect; they are optional keywords for readability.

Permissions

No special permissions required.

Notes

COMMIT (page 697) is a synonym for END.

See Also

- Transactions
- Creating and Rolling Back Transactions
- ***BEGIN*** (page 682)
- ***ROLLBACK*** (page 867)
- ***START TRANSACTION*** (page 926)

EXPLAIN

Returns the query plan execution strategy to standard output.

Syntax

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

Returns

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

```
=> EXPLAIN SELECT * FROM hTicks h FULL OUTER JOIN aTicks a ON (h.time = a.time);

------------------------------
QUERY PLAN DESCRIPTION:
------------------------------

 Access Path:
+-JOIN HASH [FullOuter] [Cost: 31, Rows: 4 (NO STATISTICS)] (PATH ID: 1) Outer (FILTER) Inner (FILTER)
|  Join Cond: (h."time" = a."time")
|  Execute on: All Nodes
| +-- Outer -> STORAGE ACCESS for h [Cost: 15, Rows: 4 (NO STATISTICS)] (PATH ID: 2)
| |      Projection: public.HTicks_node0001
| |      Materialize: h.stock, h."time", h.price
| |      Execute on: All Nodes
| +-- Inner -> STORAGE ACCESS for a [Cost: 15, Rows: 4 (NO STATISTICS)] (PATH ID: 3)
| |      Projection: public.ATicks_node0001
| |      Materialize: a.stock, a."time", a.price
| |      Execute on: All Nodes
```

Permissions

Privileges required to run this command are the same privileges required to run the query you preface with the EXPLAIN keyword.

Notes

- Information about understanding the EXPLAIN command's output is described in Understanding Query Plans in the Administrator's Guide.
- Query plans rely on reasonably representative statistics of your data. See Collecting Statistics in the Administrator's Guide for details.

EXPORT TO VERTICA

Exports an entire table, columns from a table, or the results of a ***SELECT*** (page 870) statement to another HP Vertica database. Exported data is written to the target database using AUTO mode. Exporting data to another database requires first establishing a connection to the target database using the ***CONNECT*** (page 697) statement. See Exporting Data for more setup information.

You can export data from an earlier HP Vertica release, as long as the earlier release is a version of the last major release. For instance, for Version 6.x, you can export data from any version of 5.x, but not from 4.x.

By default, using EXPORT TO VERTICA to copy data to another database occurs over the HP Vertica private network. Connecting to a public network requires some configuration. For information about using this statement to copy data across a public network, see Importing/Exporting From Public Networks.

Syntax

```
EXPORT TO VERTICA database.[dest-schema.]dest-table
... [(dest-column [,dest-column2,...])]
... { AS SELECT (page 870) select-expression
... | FROM [source-schema.]source-table
... [(source-column [,source-column2,...])]};
```

Parameters

database	A string containing the name of the database to receive the exported data. There must be an active connection to this database for the export to succeed.
.[*dest-schema.*] *dest-table*	The table to store the exported data (schema specification is optional). This table must already exist.
dest-column [,*dest-column2*,...]	A list of columns in the target table to store the exported data.

AS SELECT *select-expression*	A standard SELECT expression that selects the data to be exported. See ***SELECT*** (page 870) for the syntax.
FROM [*source-schema.*] *source-table*	The table that contains the data to be exported (schema optional)
source-column [,*source-column2*,...]	A list of the columns in the source table to export. If present, only these columns are exported.

Notes

Importing and exporting data fails if either side of the connection is a single-node cluster installed to localhost, or you do not specify a host name or IP address.

Permissions

- SELECT privileges on the source table
- USAGE privilege on source table schema
- INSERT privileges for the destination table in target database
- USAGE privilege on destination table schema

Source and Destination Column Mapping

To complete a successful export, the EXPORT TO statement maps source table columns to destination table columns. If you do not supply a list of source and destination columns, EXPORT attempts to match columns in the source table with corresponding columns in the destination table. Auto-projections for the target table are similar to the projections for the source table.

You can optionally supply lists of either source columns to be copied, columns in the destination table where data should be stored, or both. Specifying the lists lets you select a subset of source table columns to copy to the destination table. Since source and destination lists are not required, results differ depending on which list is present. The following table presents the results of supplying one or more lists:

	Omit Source Column List	Supply Source Column List

Omit Destination Column List	Matches all columns in the source table to columns in the destination table. The number of columns in the two tables need not match, but the destination table must not have fewer columns than the source.	Copies content only from the supplied list of source table columns. Matches columns in the destination table to columns in the source list. The number of columns in the two tables need not match, but the destination table must not have fewer columns than the source.
Supply Destination Column List	Matches columns in the destination column list to columns in the source. The number of columns in the destination list must match the number of columns in the source table.	Matches columns from the source table column lists to those in the destination table. The lists must have the same number of columns.

Examples

First, open the connection to the other database, then perform a simple export of an entire table to an identical table in the target database.

```
=> CONNECT TO VERTICA testdb USER dbadmin PASSWORD '' ON 'VertTest01',5433;
CONNECT
=> EXPORT TO VERTICA testdb.customer_dimension FROM customer_dimension;
Rows Exported
---------------
         23416
(1 row)
```

The following statement demonstrates exporting a portion of a table using a simple ***SELECT*** (page 870) statement.

```
=> EXPORT TO VERTICA testdb.ma_customers AS SELECT customer_key, customer_name,
annual_income
-> FROM customer_dimension WHERE customer_state = 'MA';
Rows Exported
---------------
          3429
(1 row)
```

This statement exports several columns from one table to several different columns in the target database table using column lists. Remember that when supplying both a source and destination column list, the number of columns must match.

```
=> EXPORT TO VERTICA testdb.people (name, gender, age) FROM customer_dimension
-> (customer_name, customer_gender, customer_age);
Rows Exported
---------------
         23416
(1 row)
```

You can export tables (or columns) containing Identity and Auto-increment values, but the sequence values are not incremented automatically at their destination.

You can also use the EXPORT TO VERTICA statement with a ***SELECT*** (page 870) AT EPOCH LATEST expression to include data from the latest committed DML transaction.

See Also

CONNECT (page 697)

COPY FROM VERTICA (page 711)

DISCONNECT (page 809)

GRANT Statements

GRANT (Database)

Grants the right to create schemas within the database to a user or role. By default, only the superuser has the right to create a database schema.

Syntax

```
GRANT {
... { CREATE [, ...]
... | { TEMPORARY | TEMP }
... | ALL [ PRIVILEGES ]
... | CONNECT } }
... ON DATABASE database-name [, ...]
... TO { username | rolename } [, ...]
... [ WITH GRANT OPTION ]
```

Parameters

CREATE	Allows the user to create schemas within the specified database.
TEMPORARY | TEMP	Allows the user to create temp tables in the database. **Note:** This privilege is provided by default with ***CREATE USER*** (page 801).
CONNECT	Allows the user to connect to a database.
ALL	Applies to all privileges.
PRIVILEGES	Is for SQL standard compatibility and is ignored.
database-name	Identifies the database in which to grant the privilege.
username | *rolename*	Grants the privilege to the specified user or role.
WITH GRANT OPTION	Allows the recipient of the privilege to grant it to other users.

Example

The following example grants Fred the right to create schemas on vmartdb.

```
=> GRANT CREATE ON DATABASE vmartdb TO Fred;
```

See Also

REVOKE (Database) (page 855)

Granting and Revoking Privileges in the Administrator's Guide

GRANT (Procedure)

Grants privileges on a procedure to a database user or role. Only the superuser can grant privileges to a procedure. To grant privileges to a schema containing the procedure, users must have `USAGE` privileges. See ***GRANT (Schema)*** (page 837).

Syntax

```
GRANT { EXECUTE | ALL }
   ON PROCEDURE  [ [ db-name.]schema.]procedure-name [, ...]
   ( [ argname ] argtype [,... ] )
   TO { username | role | PUBLIC } [, ...]
```

Parameters

`{ EXECUTE | ALL }`	The type of privilege to grant the procedure. Either EXECUTE or ALL are applicable privileges to grant. When using more than one schema, specify the schema that contains the procedure.
`[[ db-name.]schema-name.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
procedure-name	The SQL or User Defined procedure on which to grant the privilege. If using more than one schema, you must specify the schema that contains the procedure.
argname	The optional argument name for the procedure.
argtype	The required argument data type or types of the procedure.

{ *username* | role | PUBLIC } [,...]	The recipient of the procedure privileges, which can be one or more users, one or more roles, or all users and roles (PUBLIC). ▪ *username* – Indicates a specific user ▪ *role* – Specifies a particular role ▪ PUBLIC – Indicates that all users and roles have granted privileges to the procedure.

Example

The following command grants EXECUTE privileges on the `tokenize` procedure to users Bob and Jules, and to the Operator role:

```
=> GRANT EXECUTE ON PROCEDURE tokenize(varchar) TO Bob, Jules, Operator;
```

See Also

REVOKE (procedure) (page 856)

Granting and Revoking Privileges in the Administrator's Guide

GRANT (Resource Pool)

Grants privileges on one or more resource pools to a database user or role. Once granted usage rights, users can switch to using the resource pool with ***ALTER USER*** (page 679) (*username*) or with ***SET SESSION RESOURCE POOL*** (page 916).

Syntax

```
GRANT USAGE
   ON RESOURCE POOL resource-pool [, ...]
   TO { username | role | PUBLIC } [, ...]
```

Parameters

resource-pool	The resource pools on which to grant the privilege.
{ *username* | role | PUBLIC } [,...]	The recipient of the procedure privileges, which can be one or more users, one or more roles, or all users and roles (PUBLIC). ▪ *username* – Indicates one or more user names. ▪ *role* – Indicates one or more roles. ▪ PUBLIC – Indicates that all users and roles have granted privileges to the procedure.

See Also

REVOKE (Resource Pool) (page 857)

Granting and Revoking Privileges in the Administrator's Guide

GRANT (Role)

Adds a predefined role to users or other roles. Granting a role does not activate the role automatically; the user must enable it using the ***SET ROLE*** (page 910) command.

Granting a privilege to a role immediately affects active user sessions. When you grant a new privilege, it becomes immediately available to every user with the role active.

Syntax

```
GRANT role [,...]
... TO { user | role | PUBLIC } [, ...]
... [ WITH ADMIN OPTION ];
```

Parameters

role [,...]	The name of one or more roles to be granted to users or roles
user | *role* | PUBLIC	The name of a user or other role to be granted the role. If the keyword PUBLIC is supplied, then all users have access to the role.
WITH ADMIN OPTION	Grants users and roles administrative privileges for the role. They are able to grant the role to and revoke the role from other users or roles.

Notes

HP Vertica will return a NOTICE if you grant a role with or without admin option, to a grantee who has already been granted that role. For example:

```
=> GRANT commentor to Bob;
NOTICE 4622:  Role "commentor" was already granted to user "Bob"
```

Creating Roles

These examples create three roles, `appdata`, `applogs`, and `appadmin`, and grant the role to a user, `bob`:

```
=> CREATE ROLE appdata;
CREATE ROLE
=> CREATE ROLE applogs;
CREATE ROLE
=> CREATE ROLE appadmin;
CREATE ROLE
```

```
=> GRANT appdata TO bob;
GRANT ROLE
```

Activating a Role

After granting a role to a user, the role must be activated. You can activate a role on a session basis, or as part of the user's login.

To activate a role for a user's session:

```
=> CREATE ROLE appdata;
CREATE ROLE
=> GRANT appdata TO bob;
GRANT ROLE
=> SET ROLE appdata
```

To activate a role as part of the the user's login:

```
=> CREATE ROLE appdata;
CREATE ROLE
=> GRANT appdata TO bob;
GRANT ROLE
=> ALTER USER bob DEFAULT ROLE appdata;
```

Granting One Role To Another

Grant two roles to another role:

```
=> GRANT appdata, applogs TO appadmin; -- grant to other roles
GRANT ROLE
```

Now, any privileges assigned to either appdata or applogs are automatically assigned to appadmin as well.

Checking for Circular References

When you grant one role to another role, HP Vertica combines the newly granted role's permissions with the existing role's permissions. HP Vertica also checks for circular references when you grant one role to another. The GRANT ROLE function fails with an error if a circular reference is found.

```
=> GRANT appadmin TO appdata;
WARNING:  Circular assignation of roles is not allowed
HINT:  Cannot grant appadmin to appdata
GRANT ROLE
```

Granting Administrative Privileges

A superuser can assign a user or role administrative access to a role by supplying the optional WITH ADMIN OPTION argument to the ***GRANT*** (page 835) statement. Administrative access allows the user to grant and revoke access to the role for other users (including granting them administrative access). Giving users the ability to grant roles lets a superuser delegate role administration to other users.

As with all user privilege models, database superusers should be cautious when granting any user a role with administrative privileges. For example, if the database superuser grants two users a role with administrative privileges, both users can revoke the role of the other user. This example shows granting the `appadmin` role (with administrative privileges) to users `bob` and `alice`. After each user has been granted the `appadmin` role, either use can connect as the other will full privileges.

```
=> GRANT appadmin TO bob, alice WITH ADMIN OPTION;
GRANT ROLE
=> \connect - bob
You are now connected as user "bob".
=> REVOKE appadmin FROM alice;
REVOKE ROLE
```

See Also

REVOKE (Role) (page 858)

Granting and Revoking Privileges in the Administrator's Guide

GRANT (Schema)

Grants privileges on a schema to a database user or role.

Syntax

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

Parameters

`CREATE`	Allows the user read access to the schema and the right to create tables and views within the schema.
`USAGE`	Allows the user access to the objects contained within the schema. This allows the user to look up objects within the schema. Note that the user must also be granted access to the individual objects. See the ***GRANT TABLE*** (page 842) and ***GRANT VIEW*** (page 845) statements.
`ALL`	Applies to all privileges.
`PRIVILEGES`	Is for SQL standard compatibility and is ignored.
`[db-name.]`	[Optional] Specifies the current database name. Using a database name prefix is optional, and does not affect the command in any way. You must be connected to the specified database.

schema	Identifies the schema to which you are granting privileges.
username	Grants the privilege to a specific user.
role	Grants the privilege to a specific role.
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.

See Also

REVOKE (Schema) (page 859)

Granting and Revoking Privileges in the Administrator's Guide

GRANT (Sequence)

Grants privileges on a sequence generator to a user or role. Optionally grants privileges on all sequences within one or more schemas.

Syntax

```
GRANT { SELECT | ALL [ PRIVILEGES ] }
... ON SEQUENCE [[db-name.]schema.]sequence-name [ , ... ]
... | ON ALL SEQUENCES IN SCHEMA schema-name [, ...]
... TO { username | role | PUBLIC } [ , ... ]
... [ WITH GRANT OPTION ]
```

Parameters

SELECT	Allows the right to use both the ***CURRVAL()*** (page 353) and ***NEXTVAL()*** (page 351) functions on the specified sequence.
PRIVILEGES	Is for SQL standard compatibility and is ignored.
[[*db-name.*]*schema.*]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
sequence-name	Specifies the sequence on which to grant the privileges. When using more than one schema, specify the schema that contains the sequence on which to grant privileges.

ON ALL SEQUENCES IN SCHEMA	Grants privileges on all sequences within one or more schemas to a user and/or role.
username	Grants the privilege to the specified user.
role	Grants the privilege to the specified role.
PUBLIC	Grants the privilege to all users.
WITH GRANT OPTION	Allows the user to grant the same privileges to other users.

Notes

The user must also be granted USAGE on the schema that contains the sequence. See ***GRANT (Schema)*** (page 837).

See Also

REVOKE (Sequence) (page 860)

Granting and Revoking Privileges in the Administrator's Guide

GRANT (Storage Location)

Grants privileges to non-superusers or roles to read from or write to an HP Vertica storage location. First, a superuser ***creates*** (page 426) a special class of storage location with the **USER** keyword through the usage parameter. Creating a storage location with a USER type specifies that the the location can be made accessible to non-dbadmin users. The superuser must then grant users or roles the appropriate privileges through the GRANT (Storage Location) statement.

Note: GRANT/REVOKE (Storage Location) statements are applicable only to 'USER' storage locations. If the storage location is dropped, all privileges are revoked automatically.

Syntax

```
GRANT { READ | WRITE | ALL [ PRIVILEGES ] }
... ON LOCATION 'path' [, ON 'node' ]
... TO { username | role | PUBLIC } [, ...]
... ... [ WITH GRANT OPTION ]
```

Parameters

READ	Lets users or roles copy data from files in the storage location into a table.
WRITE	Lets users or roles export data from a table to a storage location. WRITE privileges also lets users export COPY statement exceptions and rejected data files from HP Vertica to the specified storage location.
ALL	Applies to all privileges.

PRIVILEGES	[Optional] For SQL standard compatibility and is ignored.
ON LOCATION '*path*' [, ON '*node*']	▪ *path*—[Required] Specifies where the storage location is mounted ▪ *node*—[Optional] The node on which the location is available. If this parameter is omitted, node defaults to the initiator.
{ *username* | *role* | PUBLIC } [,...]	[Required] The recipient of the privileges, which can be one or more users, one or more roles, or all users (PUBLIC). ▪ *username*–Indicates a specific user ▪ *role*–Specifies a particular role ▪ PUBLIC–Indicates that all users have READ and/or WRITE permissions.
WITH GRANT OPTION	[Optional] Allows the grantee to grant the same READ/WRITE privileges to others.

Notes

Only a superuser can add, alter, retire, drop, and restore a location, as well as set and measure location performance. All other users or roles must be granted READ and/or WRITE privileges on storage locations.

Example

In the following series of commands, a superuser creates a new storage location and grants it to user Bob:

```
dbadmin=> SELECT ADD_LOCATION('/home/dbadmin/UserStorage/BobStore',
          'v_mcdb_node0007', 'USER');
              ADD_LOCATION
-----------------------------------------
 /home/dbadmin/UserStorage/BobStore' added.
(1 row)
```

Now the superuser grants Bob READ/WRITE permissions on the /BobStore location:

```
dbadmin=> GRANT ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore' TO Bob;
GRANT PRIVILEGE
```

Revoke all storage location privileges from Bob:

```
dbadmin=> REVOKE ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore' FROM Bob;
REVOKE PRIVILEGE
```

Grant privileges to Bob on the BobStore location again, this time specifying the node:

```
dbadmin=> GRANT ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore'
          ON v_mcdb_node0007 TO Bob;
GRANT PRIVILEGE
```

Revoke all storage location privileges from Bob:

```
dbadmin=> REVOKE ALL ON LOCATION '/home/dbadmin/UserStorage/BobStore'
          ON v_mcdb_node0007 FROM Bob;
REVOKE PRIVILEGE
```

See Also

Storage Management Functions (page 636)

REVOKE (Storage Location) (page 861)

Granting and Revoking Privileges in the Administrator's Guide

GRANT (Table)

Grants privileges on a table to a user or role. Optionally grants privileges on all tables within one or more schemas.

Note: Granting privileges on all tables within a schema includes all views in the same schema.

Syntax

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES } [ ,... ]
... | ALL [ PRIVILEGES ] }
... ON [ TABLE ] [ [ db-name.]schema.]tablename [ , ... ]
... | ON ALL TABLES IN SCHEMA schema-name [, ...]
... TO { username | role | PUBLIC } [ , ... ]
... [ WITH GRANT OPTION ]
```

Parameters

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 699) command to load the table. **Note:** COPY FROM STDIN is allowed to any user granted the INSERT privilege, while COPY FROM <file> is an admin-only operation.
UPDATE	Allows the user to UPDATE tuples in the specified table.
DELETE	Allows DELETE of a row from the specified table.
REFERENCES	Is necessary to have this privilege on both the referencing and referenced tables in order to create a foreign key constraint. Also need USAGE on schema that contains the table.
ALL	Applies to all privileges.
PRIVILEGES	Is for SQL standard compatibility and is ignored.
[[*db-name.*]*schema.*]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
tablename	Specifies the table on which to grant the privileges. When using more than one schema, specify the schema that contains the table

	on which to grant privileges.
ON ALL TABLES IN SCHEMA	Grants privileges on all tables (and by default all views) within one or more schemas to a user and/or role.
username	Grants the privilege to the specified user.
role	Grants the privilege to the specified role.
PUBLIC	Grants the privilege to all users.
WITH GRANT OPTION	Allows the user to grant the same privileges to other users.

Notes

- The user must also be granted USAGE on the schema that contains the table. See ***GRANT (Schema)*** (page 837).
- To use the ***DELETE*** (page 807) or ***UPDATE*** (page 929) commands with a ***WHERE clause*** (page 901), a user must have both SELECT and UPDATE and DELETE privileges on the table.
- The user can be granted privileges on a global temporary table, but not a local temporary table.

See Also

REVOKE (Table) (page 863)

Granting and Revoking Privileges in the Administrator's Guide

GRANT (User Defined Extension)

Grants privileges on a user-defined extension (UDx) to a database user or role. Optionally grants all privileges on the user-defined extension within one or more schemas. You can grant privileges on the following user-defined extension types:

- User Defined Functions (UDF)
  - User Defined SQL Functions
  - User Defined Scalar Functions (UDSF)
  - User Defined Transform Functions (UDTF)
  - User Defined Aggregate Functions (UDAF)
  - User Defined Analytic Functions (UDAnF)
- User Defined Load Functions (UDL)
  - UDL Filter
  - UDL Parser
  - UDL Source

Syntax

```
GRANT { EXECUTE | ALL }
```

```
... ON FUNCTION [ [ db-name.]schema.]function-name [, ...]
... | ON AGGREGATE FUNCTION [ [ db-name.]schema.]function-name [, ...]
... | ON ANALYTIC FUNCTION [ [ db-name.]schema.]function-name [, ...]
... | ON TRANSFORM FUNCTION [[db-name.]schema.]function-name [, ...]
... | ON FILTER [ [ db-name.]schema.]filter-name [, ...]
... | ON PARSER [ [ db-name.]schema.]parser-name [, ...]
... | ON SOURCE [ [ db-name.]schema.]source-name [, ...]
... | ON ALL FUNCTIONS IN SCHEMA schema-name [, ...]
... ( [ argname ] argtype [, ...] )
... TO { username | role | PUBLIC } [, ...]
```

Parameters

`{ EXECUTE | ALL }`	The type of privilege to grant the function: ▪ `EXECUTE` grants permission to call a user-defined function/extension ▪ `ALL` applies to all privileges on the user-defined function/extension
`[[ db-name.]schema-name.]`	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
function-name *filter-name* *parser-name* *source-name*	The name of the user-defined extension on which to grant the privilege. If you use more than one schema, you must specify the schema that contains the UDx, as noted in the previous row.
`ON ALL FUNCTIONS IN SCHEMA`	Grants privileges on all UDx's within one or more schemas to a user and/or role.
argname	The optional argument name for the user-defined extension.
argtype	The argument data type of the function.
`{ username | role | PUBLIC } [,...]`	The recipient of the function privileges, which can be one or more users, one or more roles, or all users and roles (PUBLIC). ▪ username - Indicates a specific user ▪ role - Specifies a particular role ▪ PUBLIC - Indicates that all users and roles have granted privileges to the function.

Permissions

Only a superuser and owner can grant privileges on a user-defined extension. To grant privileges to a specific schema UDx or to all UDx's within one or more schemas, grantees must have `USAGE` privileges on the schema. See ***GRANT (Schema)*** (page 837).

Examples

The following command grants `EXECUTE` privileges on the `myzeroifnull` SQL function to users Bob and Jules, and to the Operator role. The function takes one integer argument:

```
=> GRANT EXECUTE ON FUNCTION myzeroifnull (x INT) TO Bob, Jules, Operator;
```

The following command grants `EXECUTE` privileges on all functions in the `zero-schema` schema to user Bob:

```
=> GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA zero-schema TO Bob;
```

The following command grants `EXECUTE` privileges on the `tokenize` transform function to user Bob and to the Operator role:

```
=> GRANT EXECUTE ON TRANSFORM FUNCTION tokenize(VARCHAR) TO Bob, Operator;
```

The following command grants `EXECUTE` privileges on the `HCatalogSource()` source to user Alice.

```
=> CREATE USER Alice;
=> GRANT USAGE ON SCHEMA hdfs TO Alice;
=> GRANT EXECUTE ON SOURCE HCatalogSource() TO Alice;
```

The next command grants all privileges on the `HCatalogSource()` source to user Alice:

```
=> GRANT ALL ON SOURCE HCatalogSource() TO Alice;
```

See Also

REVOKE (User Defined Extension) (page 864)

Granting and Revoking Privileges in the Administrator's Guide

Developing and Using User Defined Functions in the Programmer's Guide

GRANT (View)

Grants privileges on a view to a database user or role.

Syntax

```
GRANT
... { SELECT | ALL [ PRIVILEGES ] }
... ON [ [ db-name.]schema.]viewname [, ...]
... TO { username | role | PUBLIC } [, ...]
... [ WITH GRANT OPTION ]
```

Parameters

SELECT	Grants a user or role SELECT operations to a view, and any resources referenced within it.
ALL	Grants a user or role all privileges to a view, and any resources referenced within it.
PRIVILEGES	[Optional] For SQL standard compatibility and is ignored.
[[*db-name.*]*schema.*]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
viewname	Specifies the view on which to grant the privileges. When using more than one schema, specify the schema that contains the view, as noted above.
username	Grants the privilege to the specified user.
role	Grants the privilege to the specified role.
PUBLIC	Grants the privilege to all users.
WITH GRANT OPTION	Permits the user to grant the same privileges to other users.

Note: If `userA` wants to grant `userB` access to a view, `userA` must specify `WITH GRANT OPTION` on the base table, in addition to the view, regardless of whether `userB` (grantee) has access to the base table.

See Also

REVOKE (View) (page 866)

Granting and Revoking Privileges in the Administrator's Guide

INSERT

Inserts values into all projections of a table. You must insert one complete tuple at a time. By default, Insert first uses the WOS. When the WOS is full, the INSERT overflows to the ROS.

Note: If a table has no associated projections, HP Vertica creates a default superprojection for the table in which to insert the data.

HP Vertica does not support subqueries as the target of an INSERT statement.

Syntax

```
INSERT [ /*+ direct */ ] [ /*+ label(label-name)*/ ]
... INTO [[db-name.]schema.]table
... [ ( column [, ...] ) ]
... { DEFAULT VALUES
... | VALUES ( { expression | DEFAULT } [, ...] )
... | SELECT... (page 870) }
```

Parameters

`/*+ direct */`	Writes the data directly to disk (ROS) bypassing memory (WOS).
	HP Vertica accepts optional spaces before and after the plus (+) sign and the `direct` hint. Space characters between the opening `/*` or the closing `*/` are not permitted. The following directives are all acceptable:
	`/*+direct*/`
	`/* + direct*/`
	`/*+ direct*/`
	`/*+direct */`
	Note: If you insert using the `direct` hint, you still need to issue a `COMMIT` or `ROLLBACK` command to finish the transaction.

`/*+ label (label-name)*/`	Passes a user-defined label to a query as a hint, letting you quickly identify labeled queries for profiling and debugging. See Query Labeling in the Administrator's Guide.
`[[db-name.]schema.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
table	Specifies the name of a table in the schema. You cannot INSERT tuples into a projection. When using more than one schema, specify the schema that contains the table, as noted above.
column	Specifies one or more table columns. 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.
`DEFAULT VALUES`	Fills all columns with their default values as specified in ***CREATE TABLE*** (page 770).

VALUES	Specifies a list of values to store in the corresponding columns. If no value is supplied for a column, HP Vertica implicitly adds a DEFAULT value, if present. Otherwise HP Vertica inserts a NULL value. If the column is defined as NOT NULL, INSERT returns an error.
expression	Specifies a value to store in the corresponding column. Do not use meta-functions in INSERT statements.
DEFAULT	Stores the default value in the corresponding column.
SELECT...	Specifies a query (***SELECT*** (page 870) statement) that supplies the rows to insert. 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.

Permissions

Table owner or user with GRANT OPTION is grantor.

- INSERT privilege on table
- USAGE privilege on schema that contains the table

Examples

```
=> INSERT INTO t1 VALUES (101, 102, 103, 104);
=> INSERT INTO customer VALUES (10, 'male', 'DPR', 'MA', 35);
=> INSERT INTO retail.t1 (C0, C1) VALUES (1, 1001);
=> INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

HP Vertica does not support subqueries as the target of an INSERT statement, and the following query returns an error message:

```
INSERT INTO t1 (col1, col2) VALUES ('abc', (SELECT mycolumn FROM mytable));
ERROR 4821:  Subqueries not allowed in target of insert
```

You can rewrite the above query as follows:

```
INSERT INTO t1 (col1, col2) (SELECT 'abc', mycolumn FROM mytable);
 OUTPUT
--------
      0
(1 row)
```

When doing an INSERT /*+ direct */ HP Vertica takes optional spaces before and after the plus sign (e.g., between the /* and the +). Both of the following commands, for example, load data into the ROS:

```
=> CREATE TABLE rostab(x TIMESTAMP);
=> INSERT /*+ direct */ INTO rostab VALUES ('2011-02-10 12:01:00');
=> INSERT /*+ direct */ INTO rostab VALUES ('2011-02-15 12:01:00');
=> SELECT wos_row_count, ros_row_count FROM column_storage
   WHERE anchor_table_name = 'rostab';

 wos_row_count | ros_row_count
---------------+---------------
             0 |             2
             0 |             2
```

```
(2 rows)
```

MERGE

Lets you load a batch of new records while simultaneously updating existing records by internally combining ***INSERT*** (page 846) and ***UPDATE*** (page 929) SQL operations in one statement. In a MERGE operation, HP Vertica replaces the values of the specified columns in all rows of the target table for which a specific condition is true. All other columns and rows in the table are unchanged. By default MERGE uses the WOS, and if the WOS fills up, data overflows to the ROS.

When you write a `MERGE` statement, you specify a target and source table. You also provide a search condition through the `ON` clause, which HP Vertica uses to evaluate each row in the source table in order to update or insert its records into the target table.

You can also use optional `WHEN MATCHED` and `WHEN NOT MATCHED` clauses to further refine results. For example, if you use one or both of:

- **`WHEN MATCHED THEN UPDATE:`** HP Vertica *updates* (replaces) the values of the specified columns in all rows when it finds more than one matching row in the target table for a row in the source table. All other columns and rows in the table are unchanged. If HP Vertica finds more than one matching row in the source table for a row in the target table, it returns a run-time error.
- **`WHEN NOT MATCHED THEN INSERT:`** HP Vertica *inserts* into the target table all rows from the source table that do not match any rows in the target table.

You can help improve the performance of MERGE operations by ensuring projections are designed for optimal use. See Projection Design for Merge Operations in the Administrator's Guide.

Syntax

```
MERGE [/*+ direct */] INTO [[db-name.]schema.]target-table-name [alias]
... USING [[db-name.]schema.]source-table-name [alias ] ON ( condition )
... [ WHEN MATCHED THEN UPDATE SET column1 = value1 [, column2 = value2 ... ] ]
... [ WHEN NOT MATCHED THEN INSERT ( column1 [, column2 ...])
     VALUES ( value1 [, value2 ... ] ) ]
```

Returns

The returned value at the end of a MERGE operation is the number of rows updated plus the number of rows inserted.

Parameters

`/*+ direct */`	Writes the data directly to disk (ROS) bypassing memory (WOS).
	HP Vertica accepts optional spaces before and after the plus (+) sign and the `direct` hint. Space characters between the opening `/*` or the closing `*/` are not permitted. The following directives are all acceptable:
	`/*+direct*/`
	`/* + direct*/`
	`/*+ direct*/`
	`/*+direct */`

`[[db-name.]schema.]`	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
`INTO` *`target-table-name`*	Specifies the target table (with optional alias) that you want to update with records from the source table. You can merge records only into the target table and not, for example, into a projection. MERGE takes an X lock on the target table.
`USING` *`source-table-name`*	Specifies the source table (with optional alias) that contains the data to update or insert into the target table. Source data can come from a table reference only. Subqueries or joins are not allowed.
`ON` *`condition`*	Specifies the search condition that HP Vertica uses to evaluate each row of source table for matching rows in target table.
`WHEN MATCHED THEN UPDATE SET` *`column1 = value1`*	[Optional] Specifies what to update in the target table when the search condition matches. Use this clause when you want to ensure that HP Vertica updates existing rows in the target table with data from the source table. You can use only one WHEN MATCHED clause per statement. HP Vertica updates the target table when it finds more than one matching row in the target table for a row in the source table. If HP Vertica finds more than one matching row in the source table for a row in the target table, you'll see a run-time error.

`WHEN NOT MATCHED THEN INSERT ( column1, ...., columnn ) VALUES ( value1, ..., valuen )`	[Optional] Specifies what to update in the target table when the search condition does not match. HP Vertica will insert into the target table all rows from the source table that do not match any rows in the target table.
	You can use only one WHEN NOT MATCHED clause per statement.
	The columns specified must be columns from the target table.
	The VALUES clause specifies a list of values to store in the corresponding columns. If you do not supply a column value, do not list that column in the WHEN NOT MATCHED clause. For example, in the following syntax, column 2 (c2) is excluded from both the WHEN NOT MATCHED and VALUES clauses:
	`WHEN NOT MATCHED THEN INSERT (c1, c3, c4)`
	`VALUES (c1_value, c3_value, c4_value)`
	HP Vertica inserts a NULL value or uses the DEFAULT value (specified through the ***CREATE TABLE*** (page 770) statement) for column 2 in the above example.
	You cannot qualify table name or alias with the columns; for example, the following is not allowed:
	`WHEN NOT MATCHED THEN INSERT source.x`
	If column names are not listed, MERGE behaves like INSERT-SELECT by assuming that the columns are in the exact same table definition order.

Using named sequences

If you are using named sequences, HP Vertica can perform a MERGE operation if you omit the sequence from the query. For example, if you define column `c1` as follows, HP Vertica can do a merge:

```
CREATE TABLE t (c1 INT DEFAULT s.nextval, c2 INT);
```

HP Vertica can perform the following merge because it omits the sequence:

```
MERGE INTO t USING s ON t.c1 = s.c1
WHEN NOT MATCHED THEN INSERT (c1, c2) VALUES (s.c1, s.c2);
```

However, HP Vertica cannot perform a merge if you use a sequence to update or insert into the query, such as in the following statement, which uses the implicit/default `s.nextval` for c1:

```
MERGE INTO t USING s ON t.c1 = s.c1
WHEN NOT MATCHED THEN INSERT (c2) VALUES (s.c2);
```

HP Vertica cannot perform the following merge because it contains an explicit `s.nextval` condition:

```
MERGE INTO t USING s ON t.c1 = s.c1
WHEN MATCHED THEN UPDATE SET c2 = s.nextval;
```

Restrictions

You cannot run a MERGE operation on identity/auto-increment columns or on columns that have primary key or foreign key referential integrity constraints (as defined in CREATE TABLE ***column-constraint*** (page 783) syntax).

Examples

For examples, see Updating Tables with the MERGE Statement in the Administrator's Guide.

PROFILE

Profiles a single SQL statement.

Syntax

```
PROFILE { SELECT ... }
```

Output

Writes a hint to stderr, as described in the example below.

Permissions

Privileges required to run this command are the same privileges required to run the query being profiled.

Notes

To profile a single statement add the ***PROFILE*** (page 852) keyword to the beginning of the SQL statement, a command that saves profiling information for future analysis:

```
=> PROFILE SELECT customer_name, annual_income
   FROM public.customer_dimension
   WHERE (customer_gender, annual_income) IN (
     SELECT customer_gender, MAX(annual_income)
     FROM public.customer_dimension
     GROUP BY customer_gender);
```

A notice and hint display in the terminal window while the statement is executing. For example, the above query returns the following:

```
NOTICE:  Statement is being profiled.
HINT:  select * from v_monitor.execution_engine_profiles where
transaction_id=45035996273740886 and statement_id=10;
NOTICE:  Initiator memory estimate for query:
[on pool general: 1418047 KB, minimum: 192290 KB]
NOTICE:  Total memory required by query: [1418047 KB]
   customer_name    | annual_income
--------------------+---------------
 Meghan U. Miller   |        999960
 Michael T. Jackson |        999981
(2 rows)
```

Tip: Use the statement returned by the hint as a starting point for reviewing the query's profiling data, such as to see what counters are available.

Real-time profiling example

The following sample statement requests the operators with the largest execution time on each node:

```
=> SELECT node_name, operator_name, counter_valueexecution_time_us
     FROM v_monitor.execution_engine_profiles
     WHERE counter_name='execution time (us)'
     ORDER BY node_name, counter_value DESC;
```

How to use the Linux 'watch' command

You can use the Linux 'watch' command to monitor long-running queries with one-second updates; for example:

```
WATCH -n 1 -d "vsql-c \"select node_name, operator_name,
counter_valueexecution_time_us... \""
```

How to find out which counters are available

To see what counters are available, issue the following command:

```
=> SELECT DISTINCT(counter_name) FROM EXECUTION_ENGINE_PROFILES;
                    counter_name
-----------------------------------------------------
 estimated rows produced
 bytes spilled
 rle rows produced
 join inner current size of temp files (bytes)
 request wait (us)
 start time
 intermediate rows to process
 producer wait (us)
 rows segmented
 consumer stall (us)
 bytes sent
 rows sent
 join inner completed merge phases
 encoded bytes received
 cumulative size of raw temp data (bytes)
 end time
 bytes read from cache
 total merge phases
 rows pruned by valindex
 cumulative size of temp files (bytes)
 output queue wait (us)
 rows to process
 input queue wait (us)
 rows processed
 memory allocated (bytes)
 join inner cumulative size of temp files (bytes)
 current size of temp files (bytes)
```

```
 join inner cumulative size of raw temp data (bytes)
 bytes received
 file handles
 bytes read from disk
 join inner total merge phases
 completed merge phases
 memory reserved (bytes)
 clock time (us)
 response wait (us)
 network wait (us)
 rows received
 encoded bytes sent
 execution time (us)
 producer stall (us)
 buffers spilled
 rows produced
(43 rows)
```

See also

Profiling Query Plan Profiles

RELEASE SAVEPOINT

Destroys a savepoint without undoing the effects of commands executed after the savepoint was established.

Syntax

```
RELEASE [ SAVEPOINT ] savepoint_name
```

Parameters

savepoint_name	Specifies the name of the savepoint to destroy.

Permissions

No special permissions required.

Notes

Once destroyed, the savepoint is unavailable as a rollback point.

Example

The following example establishes and then destroys a savepoint called my_savepoint. The values 101 and 102 are both inserted at commit.

```
=> INSERT INTO product_key VALUES (101);
=> SAVEPOINT my_savepoint;
=> INSERT INTO product_key VALUES (102);
=> RELEASE SAVEPOINT my_savepoint;
=> COMMIT;
```

See Also

SAVEPOINT (page 868) and ***ROLLBACK TO SAVEPOINT*** (page 869)

REVOKE Statements

REVOKE (Database)

Revokes the right for the specified user or role to create schemas in the specified database.

Syntax

```
REVOKE [ GRANT OPTION FOR ]
... { CREATE | { TEMPORARY | TEMP } [ ,... ] }
... | CONNECT
... | ALL [ PRIVILEGES ] }
... ON DATABASE database-name [ , ... ]
... FROM { username | role }[ , ... ]
... [ CASCADE ]
```

Parameters

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	Revokes the right to create schemas in the specified database.
TEMPORARY | TEMP	Revokes the right to create temp tables in the database. **Note:** This privilege is provided by default with ***CREATE USER*** (page 801).
ALL	Applies to all privileges.
PRIVILEGES	Is for SQL standard compatibility and is ignored.
database-name	Identifies the database from which to revoke the privilege.
username	Identifies the user from whom to revoke the privilege.
role	Identifies the role from which to revoke the privilege.
CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The CASCADE keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

Example

The following example revokes Fred's right to create schemas on vmartdb:

```
=> REVOKE CREATE ON DATABASE vmartdb FROM Fred;
```

The following revokes Fred's right to create temporary tables in vmartdb:

```
=> REVOKE TEMPORARY ON DATABASE vmartdb FROM Fred;
```

See Also

GRANT (Database) (page 832)

Granting and Revoking Privileges

REVOKE (Procedure)

Revokes the execute privilege on a procedure from a user or role.

Syntax

```
REVOKE EXECUTE
... ON [ [ db-name.]schema.]procedure-name [ , ... ]
... ( [ argname ] argtype [ ,... ] ] )
... FROM { username | PUBLIC | role } [ , ... ]
...[ CASCADE ]
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
procedure-name	Specifies the procedure on which to revoke the execute privilege. When using more than one schema, specify the schema that contains the procedure, as noted above.
argname	Specifies the argument names used when creating the procedure.
argtype	Specifies the argtypes used when creating the procedure.
username	Specifies the user from whom to revoke the privilege.
PUBLIC	Revokes the privilege from all users.

role	Specifies the role from whom to revoke the privilege.
CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The CASCADE keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

Notes

Only a superuser can revoke USAGE on a procedure.

See Also

GRANT (Procedure) (page 833)

Granting and Revoking Privileges in the Administrator's Guide

REVOKE (Resource Pool)

Revokes a user's or role's access privilege to a resource pool.

Syntax

```
REVOKE USAGE
... ON RESOURCE POOL resource-pool
... FROM { username | PUBLIC | role } [ , ... ]
...[ CASCADE ]
```

Parameters

resource-pool	Specifies the resource pool from which to revoke the usage privilege.
username	Revokes the privilege from the specified user.
PUBLIC	Revokes the privilege from all users.
role	Revokes the privilege from the specified role.
CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The CASCADE keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

Notes

- HP Vertica checks resource pool permissions when a user initially switches to the pool, rather than on each access. Revoking a user's permission to use a resource pool does not affect existing sessions. You need to close the user's open sessions that are accessing the resource pool if you want to prevent them from continuing to use the pool's resources.

- It is an error to revoke a user's access permissions for the resource pool to which they are assigned (their default pool). You must first change the pool they are assigned to using ***ALTER USER ... RESOURCE POOL*** (page 679) (potentially using ***GRANT USAGE ON RESOURCE POOL*** (page 834) first to allow them to access the new pool) before revoking their access.

See Also

GRANT (Resource Pool) (page 834)

Granting and Revoking Privileges in the Administrator's Guide

REVOKE (Role)

Revokes a role (and administrative access, applicable) from a grantee. A user that has administrator access to a role can revoke the role for other users.

If you REVOKE role WITH ADMIN OPTION, HP Vertica revokes only the ADMIN OPTION from the grantee, not the role itself.

You can also remove a role's access to another role.

Syntax

```
REVOKE [ ADMIN OPTION FOR ] role [, ...]
... FROM { user | role | PUBLIC } [, ...]
...[ CASCADE ];
```

Parameters

ADMIN OPTION FOR	Revokes just the user's or role's administration access to the role.
role	The name of one or more roles from which you want to revoke access.
user | *role* | PUBLIC	The name of a user or role whose permission you want to revoke. You can use the PUBLIC option to revoke access to a role that was previously made public.
CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The CASCADE keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

Notes

If the role you are trying to revoke was not already granted to the grantee, HP Vertica returns a NOTICE:

```
=> REVOKE commentor FROM Sue;
```

```
NOTICE 2022:  Role "commentor" was not already granted to user "Sue"
REVOKE ROLE
```

See Also

GRANT (Role) (page 835)

Granting and Revoking Privileges in the Administrator's Guide

REVOKE (Schema)

Revokes privileges on a schema from a user or role.

Note: In a database with trust authentication, the GRANT and REVOKE statements appear to work as expected but have no actual effect on the security of the database.

Syntax

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

Parameters

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	Revokes the user read access to the schema and the right to create tables and views within the schema.
USAGE	Revokes user access to the objects contained within the schema. Note that the user can also have access to the individual objects revoked. See the ***GRANT TABLE*** (page 842) and ***GRANT VIEW*** (page 845) statements.
ALL	Revokes all privileges previously granted.
PRIVILEGES	Is for SQL standard compatibility and is ignored.
[*db-name.*]	[Optional] Specifies the current database name. Using a database name prefix is optional, and does not affect the command in any way. You must be connected to the specified database.
schema	Identifies the schema from which to revoke privileges.
username	Revokes the privilege to a specific user.
PUBLIC	Revokes the privilege to all users.
role	Revokes the privilege to a specific role.

CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The CASCADE keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

See Also

GRANT (Schema) (page 837)

Granting and Revoking Privileges in the Administrator's Guide

REVOKE (Sequence)

Revokes privileges on a sequence generator from a user or role. Optionally revokes privileges on all sequences within one or more schemas.

Syntax

```
REVOKE [ GRANT OPTION FOR ]
... { SELECT | ALL [ PRIVILEGES ] }
... ON SEQUENCE [ [ db-name.]schema.]sequence-name [ , ... ]
... | ON ALL SEQUENCES IN SCHEMA schema-name [, ...]
... FROM { username | PUBLIC | role } [ , ... ]
...[ CASCADE ]
```

Parameters

SELECT	Revokes the right to use both the ***CURRVAL()*** (page 353) and ***NEXTVAL()*** (page 351) functions on the specified sequence.
ALL	Applies to all privileges.
PRIVILEGES	Is for SQL standard compatibility and is ignored.
[[*db-name.*]*schema.*]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (mytable.column1), a schema, table, and column (myschema.mytable.column1), and as full qualification, a database, schema, table, and column (mydb.myschema.mytable.column1).

sequence-name	Specifies the sequence from which to revoke privileges. When using more than one schema, specify the schema that contains the sequence from which to revoke privileges.
ON ALL SEQUENCES IN SCHEMA	Revokes privileges on all sequences within one or more schemas from a user and/or role.
username	Revokes the privilege from the specified user.
PUBLIC	Revokes the privilege from all users.
role	Revokes the privilege from the specified role.
CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The CASCADE keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

See Also

GRANT (Sequence) (page 838)

Granting and Revoking Privileges in the Administrator's Guide

REVOKE (Storage Location)

Revokes privileges from a user or role to read from or write to a storage location.

Note: The REVOKE (Storage Location) statement is applicable only to 'USER' storage locations. See ***GRANT (Storage Location)*** (page 839) for more information. If the storage location is dropped, all user privileges are removed as part of that.

Syntax

```
REVOKE [ GRANT OPTION FOR ]
... { READ | WRITE | ALL [ PRIVILEGES ] }
... ON LOCATION [ 'path' , [ ON 'node' ] ]
... FROM { username | role | PUBLIC } [, ...]
... [ CASCADE ]
```

Parameters

GRANT OPTION FOR	Revokes GRANT privileges from the grantee.
READ	Revokes privileges to copy data from files in a storage locations into a table.

WRITE	Revokes privileges to export HP Vertica data from a table to a storage location. Also revokes permissions to export the COPY statement's exceptions/rejections files HP Vertica to a storage location.
ALL	Applies to all privileges.
PRIVILEGES	For SQL standard compatibility and is ignored.
{ *username* | *role* | PUBLIC } [,...]	The recipient of the privileges, which can be one or more users, one or more roles, or all users (PUBLIC). ▪ *username*–Indicates a specific user ▪ *role*–Specifies a particular role ▪ PUBLIC–Indicates that all users have READ and/or WRITE permissions.
ON LOCATION ('path' , [ON 'node'])	▪ *path*—specifies where the storage location is mounted ▪ *node*—the HP Vertica node where the location is available. If this parameter is omitted, node defaults to the initiator.
CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles; the CASCADE keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

Examples

For examples, see ***GRANT (Storage Location)*** (page 839)

See Also

Granting and Revoking Privileges in the Administrator's Guide

REVOKE (Table)

Revokes privileges on a table from a user or role. Optionally revokes privileges on all tables within one or more schemas.

Note: Revoking privileges on all tables within a schema includes all views in the same schema.

In a database with trust authentication, the GRANT and REVOKE statements appear to work as expected but have no actual effect on the security of the database.

Syntax

```
REVOKE [ GRANT OPTION FOR ]
... {
......{ SELECT | INSERT | UPDATE | DELETE | REFERENCES } [ ,... ]
......| ALL [ PRIVILEGES ]
... }
... ON [ TABLE ] [ [ db-name.]schema.]tablename [ , ... ]
... | ON ALL TABLES IN SCHEMA schema-name [, ...]
... FROM { username | PUBLIC | role } [ , ... ]
... [ CASCADE ]
```

Parameters

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	Revokes the user's ability to SELECT from any column of the specified table.
INSERT	Revokes the user from being able to INSERT tuples into the specified table and to use the ***COPY*** (page 699) command to load the table. **Note:** COPY FROM STDIN is allowed to any user granted the INSERT privilege, while COPY FROM <file> is an admin-only operation.
UPDATE	Revokes user from being allowed to UPDATE tuples in the specified table.
DELETE	Revokes user from being able to DELETE a row from the specified table.
REFERENCES	Revokes the user's privilege on both the referencing and referenced tables for creating a foreign key constraint.
ALL	Revokes all previously granted privileges.
PRIVILEGES	Is for SQL standard compatibility and is ignored.
[[*db-name.*]*schema.*]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name,

	but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
tablename	Specifies the table from which to remove privileges. When using more than one schema, specify the schema that contains the table on which to revoke privileges, as noted above.
`ON ALL TABLES IN SCHEMA`	Revokes privileges on all tables (and by default views) within one or more schemas from a user and/or role.
username	Revokes the privilege from the specified user.
`PUBLIC`	Revokes the privilege from all users.
role	Revokes the privilege from the specified role.
`CASCADE`	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The `CASCADE` keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

See Also

GRANT (Table) (page 842)

Granting and Revoking Privileges in the Administrator's Guide

REVOKE (User Defined Extension)

Revokes the `EXECUTE` privilege on a user-defined extension (UDx) from a database user or role. Optionally revokes privileges on all user-defined extensions within one or more schemas. You can revoke privileges on the following user-defined extension types:

- User Defined Functions (UDF)
  - User Defined SQL Functions
  - User Defined Scalar Functions (UDSF)
  - User Defined Transform Functions (UDTF)
  - User Defined Aggregate Functions (UDAF)

  - User Defined Analytic Functions (UDAnF)
- User Defined Load Functions (UDL)
  - UDL Filter
  - UDL Parser
  - UDL Source

Syntax

```
REVOKE EXECUTE
... ON FUNCTION [ [ db-name.]schema.]function-name [, ...]
... | ON AGGREGATE FUNCTION [ [ db-name.]schema.]function-name [, ...]
... | ON ANALYTIC FUNCTION [ [ db-name.]schema.]function-name [, ...]
... | ON TRANSFORM FUNCTION [[db-name.]schema.]function-name [, ...]
... | ON FILTER [ [ db-name.]schema.]filter-name [, ...]
... | ON PARSER [ [ db-name.]schema.]parser-name [, ...]
... | ON SOURCE [ [ db-name.]schema.]source-name [, ...]
... | ON ALL FUNCTIONS IN SCHEMA schema-name [, ...]
... FROM { username | PUBLIC | role } [ , ... ]
... [ CASCADE ]
```

Parameters

[[*db-name.*]*schema.*]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
function-name *filter-name* *parser-name* *source-name*	Specifies the name of the user-defined extension (UDx) from which to revoke the `EXECUTE` privilege. If you use more than one schema, you must specify the schema that contains the user-defined function/extension, as noted in the previous row.
`ON ALL FUNCTIONS IN SCHEMA`	Revokes `EXECUTE` privileges on all UDx's within one or more schemas from a user and/or role.
argname	Specifies the optional argument name or names for the UDx.
argtype	Specifies the argument data type or types for the UDx.
{ *username* | *role* | PUBLIC } [,...]	Revokes the privilege from the specified user, role or from all users and roles that had been granted privileges.

CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The CASCADE keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

Permissions

Only a superuser and owner can revoke EXECUTE privilege on a user defined extension.

Example

The following command revokes EXECUTE privileges from user Bob on the myzeroifnull function:

```
=> REVOKE EXECUTE ON FUNCTION myzeroifnull (x INT) FROM Bob;
```

The following command revokes EXECUTE privileges on all functions in the zero-schema schema from user Bob:

```
=> REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA zero-schema FROM Bob;
```

The following command revokes EXECUTE privileges from user Bob on the tokenize function:

```
=> REVOKE EXECUTE ON TRANSFORM FUNCTION tokenize(VARCHAR) FROM Bob;
```

The following command revokes all privileges on the HCatalogSource() source from user Alice:

```
=> REVOKE ALL ON SOURCE HCatalogSource() FROM Alice;
```

See Also

GRANT (User Defined Extension) (page 843)

Granting and Revoking Privileges in the Administrator's Guide

Developing and Using User Defined Functions in the Programmer's Guide

REVOKE (View)

Revokes user privileges on a view.

Note: In a database with trust authentication, the GRANT and REVOKE statements appear to work as expected but have no actual effect on the security of the database.

Syntax

```
REVOKE [ GRANT OPTION FOR ]
... { SELECT | ALL [ PRIVILEGES ] }
... ON [ VIEW ] [ [ db-name.]schema.]viewname [ , ... ]
... FROM { username | PUBLIC } [ , ... ]
... [ CASCADE ]
```

Parameters

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	Allows the user to perform SELECT operations on a view and the resources referenced within it.
PRIVILEGES	Is for SQL standard compatibility and is ignored.
[[*db-name.*]*schema.*]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).
viewname	Specifies the view on which to revoke the privileges. When using more than one schema, specify the schema that contains the view, as noted above.
username	Revokes the privilege from the specified user.
PUBLIC	Revokes the privilege from all users.
CASCADE	Revokes the privilege from the specified user or role and then from others. After a user or role has been granted a privilege, the user can grant that privilege to other users and roles. The `CASCADE` keyword first revokes the privilege from the initial user or role, and then from other grantees extended the privilege.

See Also

GRANT (View) (page 845)

Granting and Revoking Privileges

ROLLBACK

Ends the current transaction and discards all changes that occurred during the transaction.

Syntax

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

Parameters

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

Permissions

No special permissions required.

Notes

When an operation is rolled back, any locks that are acquired by the operation are also rolled back.

ABORT is a synonym for ROLLBACK.

See Also

- Transactions
- Creating and Rolling Back Transactions
- ***BEGIN*** (page 682)
- ***COMMIT*** (page 697)
- ***END*** (page 827)
- ***START TRANSACTION*** (page 926)

SAVEPOINT

Creates a special mark, called a savepoint, inside a transaction. A savepoint allows all commands that are executed after it was established to be rolled back, restoring the transaction to the state it was in at the point in which the savepoint was established.

Tip: Savepoints are useful when creating nested transactions. For example, a savepoint could be created at the beginning of a subroutine. That way, the result of the subroutine could be rolled back if necessary.

Syntax

```
SAVEPOINT savepoint_name
```

Parameters

savepoint_name	Specifies the name of the savepoint to create.

Permissions

No special permissions required.

Notes

- Savepoints are local to a transaction and can only be established when inside a transaction block.

- Multiple savepoints can be defined within a transaction.
- If a savepoint with the same name already exists, it is replaced with the new savepoint.

Example

The following example illustrates how a savepoint determines which values within a transaction can be rolled back. The values 102 and 103 that were entered after the savepoint, my_savepoint, was established are rolled back. Only the values 101 and 104 are inserted at commit.

```
=> INSERT INTO T1 (product_key) VALUES (101);
=> SAVEPOINT my_savepoint;
=> INSERT INTO T1 (product_key) VALUES (102);
=> INSERT INTO T1 (product_key) VALUES (103);
=> ROLLBACK TO SAVEPOINT my_savepoint;
=> INSERT INTO T1 (product_key) VALUES (104);
=> COMMIT;

=> SELECT product_key FROM T1;
--
101
104
(2 rows)
```

See Also

RELEASE SAVEPOINT (page 854) and ***ROLLBACK TO SAVEPOINT*** (page 869)

ROLLBACK TO SAVEPOINT

Rolls back all commands that have been entered within the transaction since the given savepoint was established.

Syntax

```
ROLLBACK TO [SAVEPOINT] savepoint_name
```

Parameters

savepoint_name	Specifies the name of the savepoint to roll back to.

Permissions

No special permissions required.

Notes

- The savepoint remains valid and can be rolled back to again later if needed.
- When an operation is rolled back, any locks that are acquired by the operation are also rolled back.
- ROLLBACK TO SAVEPOINT implicitly destroys all savepoints that were established after the named savepoint.

Example

The following example rolls back the values 102 and 103 that were entered after the savepoint, my_savepoint, was established. Only the values 101 and 104 are inserted at commit.

```
=> INSERT INTO product_key VALUES (101);
=> SAVEPOINT my_savepoint;
=> INSERT INTO product_key VALUES (102);
=> INSERT INTO product_key VALUES (103);
=> ROLLBACK TO SAVEPOINT my_savepoint;
=> INSERT INTO product_key VALUES (104);
=> COMMIT;
```

See Also

RELEASE SAVEPOINT (page 854) and ***SAVEPOINT*** (page 868)

SELECT

Retrieves a result set from one or more tables.

Syntax

```
[ AT EPOCH LATEST ] | [ AT TIME 'timestamp' ]
SELECT [ /*+ label(label-name)*/ ] [ ALL | DISTINCT ]
... ( expression [, ...] ) ] ]
... *
... | expression [ AS ] output_name ] [ , ... ]
... [ INTO (page 884) ]
... [ FROM (page 876) [, ...] ]
... [ WHERE (page 901) condition ]
... [ TIMESERIES (page 894) slice_time ]
... [ GROUP BY (page 878) expression [ , ... ] ]
... [ HAVING (page 880) condition [ , ... ] ]
... [ WINDOW (page 902) window_name AS ( window_definition_clause ) [ ,... ] ]
... [ MATCH (page 887) ]
... [ UNION (page 896) { ALL | DISTINCT } ]
... [ EXCEPT (page 872) ]
... [ INTERSECT (page 880) ]
... [ ORDER BY (page 893) expression { ASC | DESC } [ ,... ] ]
... [ LIMIT (page 886) { count | ALL } ]
... [ OFFSET (page 891) start ]
... [ FOR UPDATE [ OF table_name [ , ... ] ] ]
```

Parameters

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 run under the READ COMMITTED isolation level, which means:

	▪ AT EPOCH LATEST includes data from the latest committed DML transaction. ▪ Each epoch contains exactly one transaction—the one that modified the data. ▪ The Tuple Mover can perform moveout and mergeout operations on committed data immediately.
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 a historical query. AT TIME is ignored when applied to temporary tables (all rows are returned).
/*+ label (*label-name*)*/	Passes a user-defined label to a query as a hint, letting you quickly identify labeled queries for profiling and debugging. See Query Labeling in the Administrator's Guide.
*	Is equivalent to listing all columns of the tables in the FROM Clause. HP 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*** (page 54) to columns computed in the FROM clause ▪ ***Literals*** (page 24) (constants) ▪ ***Mathematical operators*** (page 47) ▪ ***String concatenation operators*** (page 49) ▪ ***Aggregate expressions*** (page 51) ▪ ***CASE expressions*** (page 52) ▪ ***SQL functions*** (page 117)
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 and GROUP BY clauses, but not in the WHERE or HAVING clauses.
FOR UPDATE	Is most often used from READ COMMITTED isolation. When specified, the SELECT statement takes an X lock on all tables specified in the query. The FOR UPDATE keywords require update/delete permissions on the tables involved and cannot be issued from a read-only transaction.

Permissions

Table owner or user with GRANT OPTION is grantor.

- SELECT privilege on table
- USAGE privilege on schema that contains the table

Privileges required on base objects for the view owner must be directly granted, not through roles:

- If a non-owner runs a SELECT query on the view, the view owner must also have SELECT ... WITH GRANT OPTION privileges on the view's base tables or views. This privilege must be directly granted to the owner, rather than through a role.
- If a view owner runs a SELECT query on the view, the owner must also have SELECT privilege directly granted (not through a role) on a view's base objects (table or view).

Example

When multiple clients run transactions like in the following example query, deadlocks can occur if `FOR UPDATE` is not used. Two transactions acquire an S lock, and when both attempt to upgrade to an X lock, they encounter deadlocks:

```
=> SELECT balance FROM accounts WHERE account_id=3476 FOR UPDATE; ...
=> UPDATE accounts SET balance = balance+10 WHERE account_id=3476;
=> COMMIT;
```

See Also

LOCKS (page 1037)

Analytic Functions (page 141)

Using SQL Analytics, Using Time Series Analytics, and Event Series Pattern Matching in the Programmer's Guide

Subqueries and Joins in the Programmer's Guide

EXCEPT Clause

Combines two or more SELECT queries, returning the results of the left-hand query that are not returned by the right-hand SELECT query.

Note: MINUS is an alias for EXCEPT.

Syntax

```
SELECT
... EXCEPT select
... [ EXCEPT select ]...
... [ ORDER BY { column-name
... | ordinal-number }
... [ ASC | DESC ] [ , ... ] ]
... [ LIMIT { integer | ALL } ]
... [ OFFSET integer ]
```

Notes

- Use the EXCEPT clause to filter out specific results from a SELECT statement. The EXCEPT query operates on the results of two or more SELECT queries; it returns only those rows in the left-hand query that are not returned by the right-hand query.
- HP Vertica evaluates multiple EXCEPT clauses in the same SELECT query from left to right, unless parentheses indicate otherwise.
- You cannot use the ALL keyword with an EXCEPT query.

- The results of each SELECT statement must be union compatible; they must return the same number of columns, and the corresponding columns must have compatible data types. For example, you cannot use the EXCEPT clause on a column of type INTEGER and a column of type VARCHAR. If they do not meet these criteria, HP Vertica returns an error.

  Note: The ***data type coercion chart*** (page 115) lists the data types that can be cast to other data types. If one data type can be cast to the other, those two data types are compatible.
- You can use EXCEPT in FROM, WHERE, and HAVING clauses.
- You can order the results of an EXCEPT operation by adding an ORDER BY operation. In the ORDER BY list, specify the column names from the leftmost SELECT statement or specify integers that indicate the position of the columns by which to sort.
- The rightmost ORDER BY, LIMIT, or OFFSET clauses in an EXCEPT query do not need to be enclosed in parentheses because the rightmost query specifies that HP Vertica perform the operation on the results of the EXCEPT operation. Any ORDER BY, LIMIT, or OFFSET clauses contained in SELECT queries that appear earlier in the EXCEPT query must be enclosed in parentheses.
- HP Vertica supports EXCEPT noncorrelated subquery predicates:

```
=> SELECT * FROM T1
   WHERE T1.x IN
      (SELECT MAX(c1) FROM T2
       EXCEPT
          SELECT MAX(cc1) FROM T3
       EXCEPT
          SELECT MAX(d1) FROM T4);
```

Examples

Consider the following three tables:

Company_A

```
  Id  | emp_lname |      dept      | sales
------+-----------+----------------+-------
 1234 | Vincent   | auto parts     |  1000
 5678 | Butch     | auto parts     |  2500
 9012 | Marcellus | floral         |   500
 3214 | Smithson  | sporting goods |  1500
(4 rows)
```

Company_B

```
  Id  | emp_lname |    dept     | sales
------+-----------+-------------+-------
 4321 | Marvin    | home goods  |   250
 8765 | Zed       | electronics | 20000
 9012 | Marcellus | home goods  |   500
 3214 | Smithson  | home goods  |  1500
(4 rows)
```

Company_C

```
  Id  | emp_lname |      dept      | sales
------+-----------+----------------+-------
 3214 | Smithson  | sporting goods |  1500
```

```
 5432 | Madison   | sporting goods |   400
 7865 | Jefferson | outdoor        |  1500
 1234 | Vincent   | floral         |  1000
(4 rows)
```

The following query returns the IDs and last names of employees that exist in Company_A, but not in Company_B:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT id, emp_lname FROM Company_B;

 id  | emp_lname
-----+-----------
 1234 | Vincent
 5678 | Butch
(2 rows)
```

The following query sorts the results of the previous query by employee last name:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT id, emp_lname FROM Company_B
   ORDER BY emp_lname ASC;

  id  | emp_lname
------+-----------
 5678 | Butch
 1234 | Vincent
(2 rows)
```

If you order by the column position, the query returns the same results:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT id, emp_lname FROM Company_B
   ORDER BY 2 ASC;

  id  | emp_lname
------+-----------
 5678 | Butch
 1234 | Vincent
(2 rows)
```

The following query returns the IDs and last names of employees that exist in Company_A, but not in Company_B or Company_C:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT id, emp_lname FROM Company_B
   EXCEPT
```

```
   SELECT id, emp_lname FROM Company_C;

  id   | emp_lname
------+-----------
 5678 | Butch
(1 row)
```

The following query shows the results of mismatched data types:

```
=> SELECT id, emp_lname FROM Company_A
   EXCEPT
   SELECT emp_lname, id FROM Company_B;
ERROR 3429:  For 'EXCEPT', types int and varchar are inconsistent
DETAIL:  Columns: id and emp_lname
```

Using the VMart example database, the following query returns information about all Connecticut-based customers who bought items through stores and whose purchases amounted to more than $500, except for those customers who paid cash:

```
=> SELECT customer_key, customer_name FROM public.customer_dimension
      WHERE customer_key IN (SELECT customer_key FROM store.store_sales_fact
         WHERE sales_dollar_amount > 500
         EXCEPT
         SELECT customer_key FROM store.store_sales_fact
         WHERE tender_type = 'Cash')
      AND customer_state = 'CT';
 customer_key |    customer_name
--------------+---------------------
        15084 | Doug V. Lampert
        21730 | Juanita F. Peterson
        24412 | Mary U. Garnett
        25840 | Ben Z. Taylor
        29940 | Brian B. Dobisz
        32225 | Ruth T. McNulty
        33127 | Darlene Y. Rodriguez
        40000 | Steve L. Lewis
        44383 | Amy G. Jones
        46495 | Kevin H. Taylor
(10 rows)
```

See Also

INTERSECT Clause (page 880)

SELECT (page 870)

Subqueries in the Programmer's Guide

UNION Clause (page 896)

FROM Clause

Specifies one or more source tables from which to retrieve rows.

Syntax

```
FROM table-reference (on page 876) [ , ... ]
... [ subquery ] [AS] name ...
```

Parameters

table-reference	Is a ***table-primary*** (on page 876) or a ***joined-table*** (on page 877).

Example

The following example returns all records from the `customer_dimension` table:

```
=> SELECT * FROM customer_dimension
```

table-reference

Syntax

```
table-primary (on page 876) | joined-table (on page 877)
```

Parameters

table-primary	Specifies an optionally qualified table name with optional table aliases, column aliases, and outer joins.
joined-table	Specifies an outer join.

table-primary

Syntax

```
{ table-name [ AS ] alias
    [ ( column-alias [ , ...] ) ] [ , ...] ]
  | ( joined-table (on page 877) ) }
```

Parameters

table-name	Specifies a table in the logical schema. HP Vertica selects a suitable projection to use.
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.

joined-table	Specifies an outer join.

joined-table

Syntax

```
table-reference join-type table-reference
ON join-predicate (on page 64)
```

Parameters

table-reference	Is a *table-primary* (page 876) or another *joined-table*.
join-type	Is one of the following: INNER JOIN LEFT [OUTER] JOIN RIGHT [OUTER] JOIN FULL [OUTER] JOIN
join-predicate	An equi-join based on one or more columns in the joined tables.

Notes

A query that uses INNER JOIN syntax in the FROM clause produces the same result set as a query that uses the WHERE clause to state the join-predicate. See Joins in the Programmer's Guide for more information.

GROUP BY Clause

Divides a query result set into sets of rows that match an expression.

Syntax

```
GROUP BY expression [ ,... ]
```

Parameters

expression	Is any expression including constants and ***references to columns*** (see "***Column References***" on page 54) in the tables specified in the FROM clause. For example: `column1, ..., column_n, aggregate_function (expression)`

Notes

- The *expression* cannot include ***aggregate functions*** (page 118); however, the GROUP BY clause is often used with ***aggregate functions*** (page 118) to return summary values for each group.
- The GROUP BY clause without aggregates is similar to using SELECT DISTINCT. For example, the following two queries are equal:

  ```
  SELECT DISTINCT household_id from customer_dimension;
   SELECT household_id from customer_dimension GROUP BY household_id;
  ```
- All non-aggregated columns in the SELECT list must be included in the GROUP BY clause.
- Using the WHERE clause with the GROUP BY clause is useful in that all rows that do not satisfy the WHERE clause conditions are eliminated before any grouping operations are computed.
- The GROUP BY clause does not order data. If you want to sort data a particular way, place the ***ORDER BY clause*** (page 893) after the GROUP BY clause.

Examples

In the following example, the WHERE clause filters out all employees whose last name does not begin with S. The GROUP BY clause returns the groups of last names that begin with S, and the SUM aggregate function computes the total vacation days for each group.

```
=> SELECT employee_last_name, SUM(vacation_days)
   FROM employee_dimension
   WHERE employee_last_name ILIKE 'S%'
   GROUP BY employee_last_name;
 employee_last_name | SUM
--------------------+------
 Sanchez            | 2892
 Smith              | 2672
 Stein              | 2660
(3 rows)

=> SELECT vendor_region, MAX(deal_size) as "Biggest Deal"
```

```
   FROM vendor_dimension
   GROUP BY vendor_region;
 vendor_region | Biggest Deal
---------------+--------------
 East          |       990889
 MidWest       |       699163
 NorthWest     |        76101
 South         |       854136
 SouthWest     |       609807
 West          |       964005
(6 rows)
```

The only difference between the following query and the one before it is the HAVING clause filters the groups to deal sizes greater than $900,000:

```
=> SELECT vendor_region, MAX(deal_size) as "Biggest Deal"
   FROM vendor_dimension
   GROUP BY vendor_region
   HAVING MAX(deal_size) > 900000;
 vendor_region | Biggest Deal
---------------+--------------
 East          |       990889
 West          |       964005
(2 rows)
```

HAVING Clause

Restricts the results of a ***GROUP BY clause*** (page 878).

Syntax

```
HAVING condition [, ...]
```

Parameters

condition	Must unambiguously reference a grouping column, unless the reference appears within an aggregate function

Notes

- Semantically the having clause occurs after the group by operation.
- You can use expressions in the HAVING clause.
- The HAVING clause was added to the SQL standard because you cannot use WHERE with ***aggregate functions*** (page 118).

Example

The following example returns the employees with salaries greater than $50,000:

```
=> SELECT employee_last_name, MAX(annual_salary) as "highest_salary"
   FROM employee_dimension
   GROUP BY employee_last_name
   HAVING MAX(annual_salary) > 50000;
 employee_last_name | highest_salary
--------------------+----------------
 Bauer              |         920149
 Brown              |         569079
 Campbell           |         649998
 Carcetti           |         195175
 Dobisz             |         840902
 Farmer             |         804890
 Fortin             |         481490
 Garcia             |         811231
 Garnett            |         963104
 Gauthier           |         927335
(10 rows)
```

INTERSECT Clause

Calculates the intersection of the results of two or more SELECT queries; returns all elements that exist in all the results.

Syntax

```
SELECT
... INTERSECT select
```

```
... [ INTERSECT select ]...
... [ ORDER BY { column-name
... | ordinal-number }
... [ ASC | DESC ] [ , ... ] ]
... [ LIMIT { integer | ALL } ]
... [ OFFSET integer ]
```

Notes

- Use the INTERSECT clause to return all elements that are common to the results of all the SELECT queries. The INTERSECT query operates on the results of two or more SELECT queries; it returns only the rows that are returned by all the specified queries. You cannot use the ALL keyword with an INTERSECT query.
- The results of each SELECT query must be union compatible; they must return the same number of columns, and the corresponding columns must have compatible data types. For example, you cannot use the INTERSECT clause on a column of type INTEGER and a column of type VARCHAR. If the SELECT queries do not meet these criteria, HP Vertica returns an error.

  Note: The ***data type coercion chart*** (page 115) lists the data types that can be cast to other data types. If one data type can be cast to the other, those two data types are compatible.
- You can order the results of an INTERSECT operation with an ORDER BY clause. In the ORDER BY list, specify the column names from the leftmost SELECT statement or specify integers that indicate the position of the columns by which to sort.
- You can use INTERSECT in FROM, WHERE, and HAVING clauses.
- The rightmost ORDER BY, LIMIT, or OFFSET clauses in an INTERSECT query do not need to be enclosed in parentheses because the rightmost query specifies that HP Vertica perform the operation on the results of the INTERSECT operation. Any ORDER BY, LIMIT, or OFFSET clauses contained in SELECT queries that appear earlier in the INTERSECT query must be enclosed in parentheses.
- HP Vertica supports INTERSECT noncorrelated subquery predicates:

```
=> SELECT * FROM T1
   WHERE T1.x IN
      (SELECT MAX(c1) FROM T2
       INTERSECT
          SELECT MAX(cc1) FROM T3
       INTERSECT
          SELECT MAX(d1) FROM T4);
```

Examples

Consider the following three tables:

Company_A

```
id       emp_lname     dept           sales
------+------------+----------------+-------
1234  | Vincent    | auto parts     | 1000
5678  | Butch      | auto parts     | 2500
9012  | Marcellus  | floral         |  500
3214  | Smithson   | sporting goods | 1500
```

Company_B

```
id       emp_lname     dept        sales
------+------------+-------------+-------
4321  | Marvin     | home goods  |   250
9012  | Marcellus  | home goods  |   500
8765  | Zed        | electronics | 20000
3214  | Smithson   | home goods  |  1500
```

Company_C

```
  id  | emp_lname |      dept      | sales
------+-----------+----------------+-------
 3214 | Smithson  | sporting goods |  1500
 5432 | Madison   | sporting goods |   400
 7865 | Jefferson | outdoor        |  1500
 1234 | Vincent   | floral         |  1000
```

The following query returns the IDs and last names of employees that exist in both Company_A and Company_B:

```
=> SELECT id, emp_lname FROM Company_A
   INTERSECT
   SELECT id, emp_lname FROM Company_B;

 id   | emp_lname
------+-----------
 3214 | Smithson
 9012 | Marcellus
(2 rows)
```

The following query returns the same two employees in descending order of sales:

```
=> SELECT id, emp_lname, sales FROM Company_A
   INTERSECT
   SELECT id, emp_lname, sales FROM Company_B
   ORDER BY sales DESC;

  id  | emp_lname | sales
------+-----------+-------
 3214 | Smithson  |  1500
 9012 | Marcellus |   500
(2 rows)
```

You can also use the integer that represents the position of the sales column (3) to return the same result:

```
=> SELECT id, emp_lname, sales FROM Company_A
```

```
   INTERSECT
   SELECT id, emp_lname, sales FROM Company_B
   ORDER BY 3 DESC;

  id  | emp_lname | sales
------+-----------+-------
 3214 | Smithson  |  1500
 9012 | Marcellus |   500
(2 rows)
```

The following query returns the employee who works for both companies whose sales in Company_B are greater than 1000:

```
=> SELECT id, emp_lname, sales FROM Company_A
   INTERSECT
   (SELECT id, emp_lname, sales FROM company_B WHERE sales > 1000)
   ORDER BY sales DESC;
  id  | emp_lname | sales
------+-----------+-------
 3214 | Smithson  |  1500
(1 row)
```

In the following query returns the ID and last name of the employee who works for all three companies:

```
=> SELECT id, emp_lname FROM Company_A
   INTERSECT
   SELECT id, emp_lname FROM Company_B
   INTERSECT
   SELECT id, emp_lname FROM Company_C;

  id  | emp_lname
------+-----------
 3214 | Smithson
(1 row)
```

The following query shows the results of a mismatched data types; these two queries are not union compatible:

```
=> SELECT id, emp_lname FROM Company_A
   INTERSECT
   SELECT emp_lname, id FROM Company_B;
ERROR 3429:  For 'INTERSECT', types int and varchar are inconsistent
DETAIL:  Columns: id and emp_lname
```

Using the VMart example database, the following query returns information about all Connecticut-based customers who bought items online and whose purchase amounts were between $400 and $500:

```
=> SELECT customer_key, customer_name from public.customer_dimension
```

```
      WHERE customer_key IN (SELECT customer_key
         FROM online_sales.online_sales_fact
         WHERE sales_dollar_amount > 400
         INTERSECT
         SELECT customer_key FROM online_sales.online_sales_fact
         WHERE sales_dollar_amount < 500)
      AND customer_state = 'CT';
 customer_key |     customer_name
--------------+-----------------------
           39 | Sarah S. Winkler
           44 | Meghan H. Overstreet
           70 | Jack X. Jefferson
          103 | Alexandra I. Vu
          110 | Matt . Farmer
          173 | Mary R. Reyes
          188 | Steve G. Williams
          233 | Theodore V. McNulty
          250 | Marcus E. Williams
          294 | Samantha V. Young
          313 | Meghan P. Pavlov
          375 | Sally N. Vu
          384 | Emily R. Smith
          387 | Emily L. Garcia
...
```

The previous query returns the same data as:

```
=> SELECT customer_key,customer_name FROM public.customer_dimension
      WHERE customer_key IN (SELECT customer_key
      FROM online_sales.online_sales_fact
         WHERE sales_dollar_amount > 400
         AND sales_dollar_amount < 500)
   AND customer_state = 'CT';
```

See Also

EXCEPT Clause (page 872)

SELECT (page 870)

Subqueries in the Programmer's Guide

UNION Clause (page 896)

INTO Clause

Creates a new table from the results of a query and fills it with data from the query.

Syntax

```
INTO [ { GLOBAL | LOCAL } { TEMPORARY | TEMP } ]
... [ TABLE ] table-name
... [ON COMMIT { PRESERVE | DELETE } ROWS ]
```

Parameters

GLOBAL	[Optional] Specifies that the table definition is visible to all sessions.
LOCAL	[Optional] Specifies that the table is visible only to the user who creates it for the duration of the session. When the session ends, the table definition is automatically dropped from the database catalogs.
TABLE	[Optional] Specifies that a table is to be created.
table-name	Specifies the name of the table to be created.
ON COMMIT { PRESERVE | DELETE } ROWS	[Optional] Specifies whether data is transaction- or session-scoped: ▪ DELETE marks a temporary table for transaction-scoped data. HP Vertica truncates the table (delete all its rows) after each commit. DELETE ROWS is the default. ▪ PRESERVE marks a temporary table for session-scoped data, which is preserved beyond the lifetime of a single transaction. HP Vertica truncates the table (delete all its rows) when you terminate a session.

Example

The following statement creates a table called newtable and fills it with the data from customer_dimension:

```
=> SELECT * INTO newtable FROM customer_dimension;
```

The following statement creates a temporary table called newtable and fills it with the data from customer_dimension:

```
=> SELECT * INTO temp TABLE newtable FROM customer_dimension;
```

The following example creates a local temporary table and inserts the contents from mytable into it:

```
=> SELECT * INTO LOCAL TEMP TABLE ltt FROM mytable;
   WARNING:  No rows are inserted into table "v_temp_schema"."ltt" because ON
COMMIT DELETE ROWS
   is the default for create temporary table
   HINT:  Use "ON COMMIT PRESERVE ROWS" to preserve the data in temporary table
   CREATE TABLE
```

See Also

Creating Temporary Tables in the Administrator's Guide

LIMIT Clause

Specifies the maximum number of result set rows to return.

Syntax

```
LIMIT { rows | ALL }
```

Parameters

rows	Specifies the maximum number of rows to return
ALL	Returns all rows (same as omitting LIMIT)

Notes

When both LIMIT and ***OFFSET*** (page 891) are used, HP Vertica skips the specified number of rows before it starts to count the rows to be returned.

You can use LIMIT without an ***ORDER BY clause*** (page 893) that includes all columns in the select list, but the query could produce nondeterministic results.

Nondeterministic: Omits the ORDER BY clause and returns *any* five records from the customer_dimension table:

```
=> SELECT customer_city
   FROM customer_dimension
   LIMIT 5;
 customer_city
---------------
 Baltimore
 Nashville
 Allentown
 Clarksville
 Baltimore
(5 rows)
```

Deterministic: Specifies the ORDER BY clause:

```
=> SELECT customer_city
   FROM customer_dimension
   ORDER BY customer_city
   LIMIT 5;
 customer_city
---------------
 Abilene
 Abilene
 Abilene
 Abilene
 Abilene
(5 rows)
```

Examples

The following examples illustrate the LIMIT clause queries that HP Vertica supports:

```
=> SELECT * FROM (SELECT * FROM t1 ORDER BY x ) alias LIMIT 3;
=> SELECT * FROM (SELECT * FROM t1 ORDER BY x LIMIT 5) alias LIMIT 3;
=> SELECT * FROM (SELECT * FROM t1 ORDER BY x) alias LIMIT 4 OFFSET 3;
=> SELECT * FROM t1 UNION SELECT * FROM t2 LIMIT 3;
=> SELECT * FROM fact JOIN dim using (x) LIMIT 3;
=> SELECT * FROM t1 JOIN t2 USING (x) LIMIT 3;
```

MATCH Clause

A SQL extension that lets you screen large amounts of historical data in search of event patterns, the MATCH clause provides subclasses for analytic partitioning and ordering and matches rows from the result table based on a pattern you define.

You specify a pattern as a regular expression, which is composed of event types defined in the DEFINE subclause, where each event corresponds to a row in the input table. Then you can search for the pattern within a sequence of input events. Pattern matching returns the contiguous sequence of rows that conforms to PATTERN subclause. For example, pattern `P (A B* C)` consist of three event types: A, B, and C. When HP Vertica finds a match in the input table, the associated pattern instance must be an event of type A followed by 0 or more events of type B, and an event of type C.

Pattern matching is particularly useful for clickstream analysis where you might want to identify users' actions based on their Web browsing behavior (page clicks). A typical online clickstream funnel is:

Company home page -> product home page -> search -> results -> purchase online

Using the above clickstream funnel, you can search for a match on the user's sequence of web clicks and identify that the user:

- landed on the company home page
- navigated to the product page
- ran a search
- clicked a link from the search results
- made a purchase

For examples that use this clickstream model, see Event Series Pattern Matching in the Programmer's Guide.

Syntax

```
MATCH ( [ PARTITION BY table_column ] ORDER BY table_column
... DEFINE event_name AS boolean_expr [, ...]
... PATTERN pattern_name AS ( regexp )
... [ ROWS MATCH ( ALL EVENTS | FIRST EVENT ) ] )
```

Parameters

PARTITION BY	[Optional] Defines the window data scope in which the pattern, defined in the PATTERN subclause, is matched. The partition clause partitions the data by matched patterns defined in the PATTERN subclause. For each partition, data is sorted by the ORDER BY clause. If the partition clause is omitted, the entire data set is considered a single partition.
ORDER BY	Defines the window data scope in which the pattern, defined in the PATTERN subclause, is matched. For each partition, the order clause specifies how the input data is ordered for pattern matching. **Note:** The ORDER BY clause is mandatory.

DEFINE	Defines the ***boolean*** (page 60) expressions that make up the event types in the regular expressions. For example:
	`DEFINE`
	`Entry AS RefURL NOT ILIKE '%website2.com%' AND PageURL ILIKE '%website2.com%',`
	`Onsite AS PageURL ILIKE '%website2.com%' AND Action='V',`
	`Purchase AS PageURL ILIKE '%website2.com%' AND Action = 'P'`
	The DEFINE subclause accepts a maximum of 52 events. See Event Series Pattern Matching in the Programmer's Guide for examples.

event_name	Is the name of the event to evaluate for each row; for example, `Entry, Onsite, Purchase`. **Note:** Event names are case insensitive and follow the same naming conventions as those used for tables and columns.
boolean_expr	Is an expression that returns true or false. boolean_expr can include ***Boolean operators*** (on page 44) and relational ***(comparison)*** (see "***Comparison Operators***" on page 44) operators. For example: `Purchase AS PageURL ILIKE '%website2.com%' AND Action = 'P'`
PATTERN *pattern_name*	Is the name of the pattern, which you define in the PATTERN subclause; for example, P is the pattern name defined below: `PATTERN P AS (...)` PATTERN defines the regular expression composed of event types that you specify in the DEFINE subclause. **Note:** HP Vertica supports one pattern per query.
regexp	Is the pattern that is composed of the event types defined in the DEFINE subclause and which returns the contiguous sequence of rows that conforms to the PATTERN subclause. `*` Match 0 or more times `*?` Match 0 or more times, not greedily `+` Match 1 or more times `+?` Match 1 or more times, not greedily `?` Match 0 or 1 time `??` Match 0 or 1 time, not greedily `*+` Match 0 or more times, possessive `++` Match 1 or more times, possessive `?+` Match 0 or 1 time, possessive **Note:** Syntax for regular expressions is compatible with the Perl 5 regular expression syntax. See the ***Perl Regular Expressions Documentation*** (***http://perldoc.perl.org/perlre.html***) for details.

ROWS MATCH	[Optional] Defines how to resolve more than one event evaluating to true for a single row. ▪ If you use ROWS MATCH ALL EVENTS (default), HP Vertica returns the following run-time error if more than one event evaluates to true for a single row: ▪ `ERROR: pattern events must be mutually exclusive` ▪ `HINT: try using ROWS MATCH FIRST EVENT` ▪ For ROWS MATCH FIRST EVENT, if more than one event evaluates to true for a single row, HP Vertica chooses the event defined first in the SQL statement to be the event it uses for the row.

Pattern Semantic Evaluation

- The semantic evaluating ordering of the SQL clauses is: FROM -> WHERE -> PATTERN MATCH -> SELECT.
- Data is partitioned as specified in the the PARTITION BY clause. If the partition clause is omitted, the entire data set is considered a single partition.
- For each partition, the order clause specifies how the input data is ordered for pattern matching.
- Events are evaluated for each row. A row could have 0, 1, or *N* events evaluate to true. If more than one event evaluates to true for the same row, HP Vertica returns a run-time error unless you specify ROWS MATCH FIRST EVENT. If you specify ROWS MATCH FIRST EVENT and more than one event evaluates to TRUE for a single row, HP Vertica chooses the event that was defined first in the SQL statement to be the event it uses for the row.
- HP Vertica performs pattern matching by finding the contiguous sequence of rows that conforms to the pattern defined in the PATTERN subclause.

  For each match, HP Vertica outputs the rows that contribute to the match. Rows not part of the match (do not satisfy one or more predicates) are not output.
- HP Vertica reports only non-overlapping matches. If an overlap occurs, HP Vertica chooses the first match found in the input stream. After finding the match, HP Vertica looks for the next match, starting at the end of the previous match.
- HP Vertica reports the longest possible match, not a subset of a match. For example, consider pattern: A*B with input: AAAB. Because A uses the greedy regular expression quantifier (*), HP Vertica reports all A inputs (AAAB), not AAB, AB, or B.

Notes and Restrictions

- DISTINCT and GROUP BY/HAVING clauses are not allowed in pattern match queries.
- The following expressions are not allowed in the DEFINE subclause:
  - Subqueries, such as `DEFINE X AS c IN (SELECT c FROM table1)`
  - Analytic functions, such as `DEFINE X AS c < LEAD(1) OVER (ORDER BY 1)`
  - Aggregate functions, such as `DEFINE X AS c < MAX(1)`
- You cannot use the same pattern name to define a different event; for example, the following is not allowed for X:

```
DEFINE
  X AS c1 <  3
  X AS c1  >= 3
```

- Used with MATCH clause, HP Vertica ***pattern matching functions*** (page 331) provide additional data about the patterns it finds. For example, you can use the functions to return values representing the name of the event that matched the input row, the sequential number of the match, or a partition-wide unique identifier for the instance of the pattern that matched.

Example

See Event Series Pattern Matching in the Programmer's Guide.

See Also

Pattern matching functions (page 331): ***EVENT_NAME*** (page 331), ***MATCH_ID*** (page 332), and ***PATTERN_ID*** (page 334)

Perl Regular Expressions Documentation (***http://perldoc.perl.org/perlre.html***)

MINUS Clause

MINUS is an alias for ***EXCEPT*** (page 872).

OFFSET Clause

Omits a specified number of rows from the beginning of the result set.

Syntax

```
OFFSET rows
```

Parameters

rows	specifies the number of result set rows to omit.

Notes

- When both ***LIMIT*** (page 886) 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 893). Otherwise the query returns an undefined subset of the result set.

Example

The following example is similar to the the example used in the ***LIMIT clause*** (page 886). If you want to see just records 6-10, however, use the OFFSET clause to skip over the first five cities:

```
=> SELECT customer_city
   FROM customer_dimension
   WHERE customer_name = 'Metamedia'
   ORDER BY customer_city
   OFFSET 5;
  customer_city
------------------
 El Monte
 Fontana
 Hartford
 Joliet
 Peoria
 Rancho Cucamonga
 Ventura
 Waco
 Wichita Falls
(9 rows)
```

The following are the results without the OFFSET clause:

```
  customer_city
------------------
 Arvada
 Arvada
 Athens
 Beaumont
 Coral Springs
 El Monte
 Fontana
 Hartford
```

```
 Joliet
 Peoria
 Rancho Cucamonga
 Ventura
 Waco
 Wichita Falls
(14 rows)
```

ORDER BY Clause

Sorts a query result set on one or more columns.

Syntax

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

Parameters

expression	Can be: ▪ The name or ***ordinal number*** (***http://en.wikipedia.org/wiki/Ordinal_number***) of a SELECT list item ▪ An arbitrary expression formed from columns that do not appear in the SELECT list ▪ A ***CASE*** (page 52) 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.)
- While the user's current locale and collation sequence are used to compare strings and determine the results of the ORDER BY clause of a query, data in HP Vertica projections is always stored sorted by the ASCII (binary) collating sequence.
- For INTEGER, INT, 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.

Example

The follow example returns all the city and deal size for customer Metamedia, sorted by deal size in descending order.

```
=> SELECT customer_city, deal_size
   FROM customer_dimension
   WHERE customer_name = 'Metamedia'
   ORDER BY deal_size DESC;
  customer_city   | deal_size
------------------+-----------
 El Monte         |   4479561
 Athens           |   3815416
 Ventura          |   3792937
 Peoria           |   3227765
 Arvada           |   2671849
 Coral Springs    |   2643674
 Fontana          |   2374465
 Rancho Cucamonga |   2214002
 Wichita Falls    |   2117962
```

```
 Beaumont          |   1898295
 Arvada            |   1321897
 Waco              |   1026854
 Joliet            |    945404
 Hartford          |    445795
(14 rows)
```

TIMESERIES Clause

Provides gap-filling and interpolation (GFI) computation, an important component of time series analytics computation. See Using Time Series Analytics in the Programmer's Guide for details and examples.

Syntax

```
TIMESERIES slice_time AS 'length_and_time_unit_expression' OVER (
... [ window_partition_clause (page 143) [ , ... ] ]
... ORDER BY time_expression )
... [ ORDER BY table_column [ , ... ] ]
```

Parameters

slice_time	A time column produced by the TIMESERIES clause, which stores the time slice start times generated from gap filling. **Note:** This parameter is an alias, so you can use any name that an alias would take.
'length_and_time_unit_expression'	Is INTERVAL (DAY TO SECOND) that represents the length of time unit of time slice computation; for example, `TIMESERIES slice_time AS '3 seconds' ...`
OVER()	Specifies partitioning and ordering for the function. OVER() also specifies that the time series function operates on a query result set (the rows that are returned after the FROM, WHERE, GROUP BY, and HAVING clauses have been evaluated).
PARTITION BY	Partitions the data by expressions (*column1 ..., column_n, slice_time*).
expression	Expressions on which to partition the data, where each partition is sorted by *time_expression*. Gap filling and interpolation is performed on each partition separately.
ORDER BY	Sorts the data by *time_expression*. **Note:** The TIMESERIES clause requires an ORDER BY operation on the timestamp column.
time_expression	An expression that computes the time information of the time series data. The *time_expression* can be TIMESTAMP data type only.

Notes

If the `window_partition_clause` is not specified in TIMESERIES OVER(), for each defined time slice, exactly one output record is produced; otherwise, one output record is produced per partition per time slice. Interpolation is computed there.

Given a query block that contains a TIMESERIES clause, the following are the semantic phases of execution (after evaluating the FROM and the optional WHERE clauses):

1. Compute *time_expression.*
2. Perform the same computation as the TIME_SLICE() function on each input record based on the result of *time_expression* and '*length_and_time_unit_expression*'.
   1. Perform gap filling to generate time slices missing from the input.
   2. Name the result of this computation as *slice_time*, which represents the generated "time series" column (alias) after gap filling.
3. Partition the data by *expression*, *slice_time*. For each partition, do step 4.
4. Sort the data by *time_expression*. Interpolation is computed here.

There is semantic overlap between the TIMESERIES clause and the ***TIME_SLICE*** (page 240) function with the following key differences:

- TIMESERIES only supports the DAY TO SECOND ***interval-qualifier*** (on page 40) (does not allow YEAR TO MONTH).
- Unlike TIME_SLICE, the time slice length and time unit expressed in *length_and_time_unit_expr* must be constants so gaps in the time slices are well-defined.
- TIMESERIES performs gap filling; the TIME_SLICE function does not.
- TIME_SLICE can return the start or end time of a time slice, depending on the value of its fourth input parameter (*start_or_end*). TIMESERIES, on the other hand, always returns the start time of each time slice. To output the end time of each time slice, you can write a SELECT statement like the following:

  ```
  SELECT slice_time + <slice_length>;
  ```

Restrictions

- When the TIMESERIES clause occurs in a SQL query block, only SELECT, FROM, WHERE, and ORDER BY clauses can be used in that same query block. GROUP BY and HAVING clauses are not allowed.

  If a GROUP BY operation is needed before or after gap-filling and interpolation (GFI), use a subquery and place the GROUP BY In the outer query. For example:

  ```
  => SELECT symbol, AVG(first_bid) as avg_bid FROM (
          SELECT symbol, slice_time, TS_FIRST_VALUE(bid1) AS first_bid
          FROM Tickstore
          WHERE symbol IN ('MSFT', 'IBM')
          TIMESERIES slice_time AS '5 seconds' OVER (PARTITION BY symbol
  ORDER BY ts)
          ) AS resultOfGFI
  GROUP BY symbol;
  ```

- When the TIMESERIES clause is present in the SQL query block, only time series aggregate functions (such as ***TS_FIRST_VALUE*** (page 421) and ***TS_LAST_VALUE*** (page 422)), the slice_time column, PARTITION BY expressions, and ***TIME_SLICE*** (page 240) are allowed in the SELECT list. For example, the following two queries would return a syntax error because bid1 was not a PARTITION BY or GROUP BY column:

  ```
  => SELECT bid, symbol, TS_FIRST_VALUE(bid) FROM Tickstore
  ```

```
   TIMESERIES slice_time AS '5 seconds' OVER (PARTITION BY symbol ORDER
BY ts);
   ERROR:  column "Tickstore.bid" must appear in the PARTITION BY list
of Timeseries clause or be used in a Timeseries Output function
=> SELECT bid, symbol, AVG(bid) FROM Tickstore
   GROUP BY symbol;
   ERROR:  column "Tickstore.bid" must appear in the GROUP BY clause or
be used in an aggregate function
```

Examples

See Gap Filling and Interpolation (GFI) in the Programmer's Guide.

See Also

TIME_SLICE (page 240), ***TS_FIRST_VALUE*** (page 421), and ***TS_LAST_VALUE*** (page 422)

Using Time Series Analytics in the Programmer's Guide

UNION Clause

Combines the results of two or more SELECT statements.

Syntax

```
SELECT
... UNION { ALL | DISTINCT } select
... [ UNION { ALL | DISTINCT } select ]...
... [ ORDER BY { column-name | ordinal-number }
... [ ASC | DESC ] [ , ... ] ]
... [ LIMIT { integer | ALL } ]
... [ OFFSET integer ]
```

Notes

- The UNION operation combines the results of several SELECT statements into a single result. Specifically, all rows in the result of a UNION operation must exist in the results of at least one of the SELECT statements.
- The results of each SELECT statement must be union compatible; they must return the same number of columns, and the corresponding columns must have compatible data types. For example, you cannot use the UNION clause on a column of type INTEGER and a column of type VARCHAR. If they do not meet these criteria, HP Vertica returns an error.

  Note: The ***data type coercion chart*** (page 115) lists the data types that can be cast to other data types. If one data type can be cast to the other, those two data types are compatible.
- The results of a UNION contain only distinct rows. UNION ALL keeps all duplicate rows, and so results in better performance than UNION. Therefore, unless you must eliminate duplicates, use UNION ALL for best performance.
- You can use UNION [ALL] in FROM, WHERE, and HAVING clauses.
- You can order the results of an UNION operation with an ORDER BY clause. In the ORDER BY list, specify the column names from the leftmost SELECT statement or specify integers that indicate the position of the columns by which to sort.

- The rightmost ORDER BY, LIMIT, or OFFSET clauses in a UNION query do not need to be enclosed in parentheses because the rightmost query specifies that HP Vertica perform the operation on the results of the UNION operation. Any ORDER BY, LIMIT, or OFFSET clauses contained in SELECT queries that appear earlier in the UNION query must be enclosed in parentheses.
- HP Vertica supports UNION noncorrelated subquery predicates:

```
=> SELECT * FROM T1
   WHERE T1.x IN
       (SELECT MAX(c1) FROM T2
        UNION { ALL | DISTINCT }
           SELECT MAX(cc1) FROM T3
        UNION { ALL | DISTINCT }
           SELECT MAX(d1) FROM T4);
```

Examples

Consider the following two tables:

Company_A

```
Id       emp_lname     dept         sales
------+-----------+-------------+-------
1234  | Vincent    | auto parts  | 1000
5678  | Butch      | auto parts  | 2500
9012  | Marcellus  | floral      |  500
```

Company_B

```
Id       emp_lname     dept         sales
------+-----------+-------------+-------
4321  | Marvin     | home goods  |   250
9012  | Marcellus  | home goods  |   500
8765  | Zed        | electronics | 20000
```

The following query lists all *distinct* IDs and last names of employees; Marcellus works for both companies, so he only appears once in the results. The DISTINCT keyword is the default; if you omit the DISTINCT keyword, the query returns the same results:

```
=> SELECT id, emp_lname FROM Company_A
   UNION DISTINCT
   SELECT id, emp_lname FROM Company_B;
  id  | emp_lname
------+-----------
 1234 | Vincent
 4321 | Marvin
 5678 | Butch
 8765 | Zed
 9012 | Marcellus
(5 rows)
```

The following query lists *all* IDs and surnames of employees. Marcellus works for both companies, so both his records appear in the results:

```
=> SELECT id, emp_lname FROM Company_A
   UNION ALL
   SELECT id, emp_lname FROM Company_B;
  id  | emp_lname
------+-----------
 1234 | Vincent
 5678 | Butch
 9012 | Marcellus
 4321 | Marvin
 9012 | Marcellus
 8765 | Zed
(6 rows)
```

The next example returns the top two performing salespeople in each company and orders the full results in descending order of sales:

```
=> (SELECT id, emp_lname, sales FROM Company_A
      ORDER BY sales DESC LIMIT 2)
   UNION ALL
   (SELECT id, emp_lname, sales FROM Company_b
      ORDER BY sales DESC LIMIT 2)
   ORDER BY sales DESC;
  id  | emp_lname | sales
------+-----------+-------
 8765 | Zed       | 20000
 5678 | Butch     |  2500
 1234 | Vincent   |  1000
 9012 | Marcellus |   500
(4 rows)
```

The following query returns all employee orders by sales. The rightmost clause (ORDER BY) is applied to the entire result, listing the sales in ascending order:

```
=> SELECT id, emp_lname, sales FROM Company_A
   UNION
   SELECT id, emp_lname, sales FROM Company_B
   ORDER BY sales;

  id  | emp_lname | sales
------+-----------+-------
 4321 | Marvin    |   250
 9012 | Marcellus |   500
 1234 | Vincent   |  1000
 5678 | Butch     |  2500
 8765 | Zed       | 20000
(5 rows)
```

To calculate the sum of the sales for each company, grouped by department, use this query:

```
=> (SELECT 'Company A' as company, dept, SUM(sales) FROM Company_A
    GROUP BY dept)
    UNION
   (SELECT 'Company B' as company, dept, SUM(sales) FROM Company_B
    GROUP BY dept)
    ORDER BY 1;

 company   |    dept     |  sum
-----------+-------------+-------
 Company A | auto parts  |  3500
 Company A | floral      |   500
 Company B | electronics | 20000
 Company B | home goods  |   750
(4 rows)
```

The following query shows the results of mismatched data types:

```
=> SELECT id, emp_lname FROM Company_A
   UNION
   SELECT emp_lname, id FROM Company_B;
ERROR 3429:  For 'UNION', types int and varchar are inconsistent
DETAIL:  Columns: id and emp_lname
```

Using the VMart example database, the following query returns information about all Connecticut-based customers who bought items through either stores or online sales channels and whose purchases totaled more than $500:

```
=> SELECT DISTINCT customer_key, customer_name FROM public.customer_dimension
   WHERE customer_key IN (SELECT customer_key FROM store.store_sales_fact
      WHERE sales_dollar_amount > 500
      UNION ALL
      SELECT customer_key FROM online_sales.online_sales_fact
      WHERE sales_dollar_amount > 500)
   AND customer_state = 'CT';
 customer_key |     customer_name
--------------+-----------------------
          955 | Darlene N. Vu
        48916 | Tanya H. Wilson
        43123 | Luigi I. Fortin
        33780 | Ben T. Nielson
        24827 | Luigi . Kramer
        21631 | Jack R. Perkins
        31493 | Matt E. Miller
        12438 | Samantha Q. Campbell
         7021 | Luigi T. Dobisz
         5229 | Julie V. Garcia
         1971 | Betty V. Dobisz
...
```

See Also

EXCEPT Clause (page 872)

INTERSECT Clause (page 880)

SELECT (page 870)

Subqueries in the Programmer's Guide

WHERE Clause

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

Syntax

```
WHERE boolean-expression
     [ subquery ] ...
```

Parameters

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 44) and the following elements:

- ***BETWEEN-predicate*** (on page 58)
- ***Boolean-predicate*** (on page 60)
- ***Column-value-predicate*** (on page 60)
- ***IN-predicate*** (on page 61)
- ***Join-predicate*** (on page 64)
- ***LIKE-predicate*** (on page 66)
- ***NULL-predicate*** (on page 69)

Notes

You can use parentheses to group expressions, predicates, and boolean operators. For example:

```
=> ... WHERE NOT (A=1 AND B=2) OR C=3;
```

Example

The following example returns the names of all customers in the Eastern region whose name starts with 'Amer'. Without the WHERE clause filter, the query returns *all* customer names in the customer_dimension table.

```
=> SELECT DISTINCT customer_name
   FROM customer_dimension
   WHERE customer_region = 'East'
   AND customer_name ILIKE 'Amer%';
 customer_name
---------------
 Americare
 Americom
 Americore
 Americorp
 Ameridata
 Amerigen
 Amerihope
 Amerimedia
 Amerishop
```

```
 Ameristar
 Ameritech
(11 rows)
```

WINDOW Clause

Creates a named window for an analytics query so you can avoid typing long `OVER()` clause syntax.

Syntax

```
WINDOW window_name AS ( window_definition_clause );
```

Parameters

`WINDOW window_name`	Specifies that window name.
`AS window_definition_clause`	Defines the ***window_partition_clause*** (on page 143) and ***window_order_clause*** (on page 144)

See Also

Analytic Functions (page 141)

Named Windows in the Programmer's Guide

WITH Clause

WITH clauses are individually-evaluated SELECT statements for use in a larger, container query. Each WITH clause is evaluated once while processing the main query, though it can be used in subsequent WITH clauses to create combinatorial results. HP Vertica maintains the results of each concomitant WITH statement as if it were a temporary table, stored only for the duration of the container query's execution. Each WITH clause query must have a unique name. Attempting to use same-name aliases for WITH clause query names causes an error.

Syntax

This syntax statement is illustrative, rather than syntactically exact, to show the possibility of numerous successive WITH queries in use with others:

```
WITH
... with_query_1 [(col_name[,...])]AS (SELECT ...),
... with_query_2 [(col_name[,...])]AS (SELECT ...[with_query_1]),
.
.
.
... with_query_n [(col_name[,...])]AS (SELECT ...[with_query1, with_query_2,
with_query_n [,...]])
SELECT
.
```

.
.

Restrictions for WITH Clauses

There are two restrictions when using WITH clauses:

- Do not support INSERT, UPDATE, or DELETE statements.
- Cannot be used recursively, only in succession.

Examples

Consider the following example:

```
-- Begin WITH clauses,

-- First WITH clause,regional_sales
WITH
   regional_sales AS (
      SELECT region, SUM(amount) AS total_sales
      FROM orders
      GROUP BY region),

-- Second WITH clause top_regions
   top_regions AS (
       SELECT region
       FROM regional_sales
       WHERE total_sales > (SELECT SUM (total_sales)/10 FROM regional_sales) )

-- End defining WITH clause statement

-- Begin main primary query
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
```

GROUP BY region, product;

See Also

SELECT (page 870)

Subqueries and WITH Clauses in SELECT in the Programmer's Guide

SET DATESTYLE

Specifies the format of date/time output for the current session.

Syntax

```
SET DATESTYLE TO { value | 'value' } [ ,... ]
```

Parameters

The DATESTYLE parameter can have multiple, non-conflicting values:

Value	Interpretation	Example
MDY	month-day-year	12/16/2011
DMY	day-month-year	16/12/2011
YMD	year-month-day	2011-12-16
ISO	ISO 8601/SQL standard (default)	2011-12-16 07:37:16-08
POSTGRES	verbose style	Fri Dec 16 07:37:16 2012 PST
SQL	traditional style	12/16/2011 07:37:16.00 PST
GERMAN	regional style	16.12.2011 07:37:16.00 PST

In the SQL style, if DMY field ordering has been specified, the day appears before the month. Otherwise, the month appears before the day. (To see how this setting also affects interpretation of input values, see ***Date/Time Literals*** (page 35).) The following table shows an example.

DATESTYLE	Input Ordering	Example Output
SQL, DMY	day/month/year	17/12/2007 15:37:16.00 CET
SQL, MDY	month/day/year	12/17/2007 07:37:16.00 PST

Permissions

No special permissions required.

Notes

- HP Vertica ISO output for DATESTYLE is ISO long form, but several input styles are accepted. If the year appears first in the input, YMD is used for input and output, regardless of the DATESTYLE value.
- The SQL standard requires the use of the ISO 8601 format.
- 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 the appropriate sign. In ISO mode, the output looks like
  `[ quantity unit [ ... ] ] [ days ] [ hours:minutes:seconds ]`
- The SHOW (page 923) command displays the run-time parameters.

Example

The following examples show how the DATESTYLE parameter affects the output of the SELECT INTERVAL command:

```
=>SHOW DATESTYLE;
   name    | setting
-----------+----------
 datestyle | ISO, MDY

=> SELECT INTERVALYM '-12-11', INTERVAL '-10 15:05:1.234567';
 ?column? |      ?column?
----------+--------------------
 -12-11   | -10 15:05:01.234567

=> SET INTERVALSTYLE TO UNITS;
SET
=> SELECT INTERVALYM '-12-11', INTERVAL '-10 15:05:1.234567';
      ?column?       |         ?column?
---------------------+-------------------------
 -12 years 11 months | -10 days 15:05:01.234567

=> SET DATESTYLE TO SQL;;
SET
=> SELECT INTERVALYM '-12-11', INTERVAL '-10 15:05:1.234567';
 ?column? |      ?column?
----------+--------------------
 -12-11   | -10 15:05:01.234567

=> SET DATESTYLE TO POSTGRES;
SET
=> SELECT INTERVALYM '-12-11', INTERVAL '-10 15:05:1.234567';
         ?column?         |                  ?column?
--------------------------+---------------------------------------------
 @ 12 years 11 months ago | @ 10 days 15 hours 5 mins 1.234567 secs ago


=> SET DATESTYLE TO GERMAN;
SET
=> SELECT INTERVALYM '-12-11', INTERVAL '-10 15:05:1.234567';
      ?column?       |         ?column?
---------------------+-------------------------
 -12 years 11 months | -10 days 15:05:01.234567
```

SET ESCAPE_STRING_WARNING

Issues a warning when a backslash is used in a string literal during the current session.

Syntax

```
SET ESCAPE_STRING_WARNING TO { ON | OFF }
```

Parameters

ON	[Default] Issues a warning when a back slash is used in a string literal. **Tip:** Organizations that have upgraded from earlier versions of HP Vertica can use this as a debugging tool for locating backslashes that used to be treated as escape characters, but are now treated as literals.
OFF	Ignores back slashes within string literals.

Permissions

No special permissions required.

Notes

- This statement works under vsql only.
- Turn off standard conforming strings before you turn on this parameter.

Tip: To set escape string warnings across all sessions, use the EscapeStringWarnings configuration parameter. See the Internationalization Parameters in the Administrator's Guide.

Examples

The following example shows how to turn OFF escape string warnings for the session.

```
=> SET ESCAPE_STRING_WARNING TO OFF;
```

See Also

STANDARD_CONFORMING_STRINGS (page 920)

SET INTERVALSTYLE

Specifies whether to include units in interval output for the current session.

Syntax

```
SET INTERVALSTYLE TO [ plain | units (see "interval-literal" on page 38) ]
```

Parameters

plain	Sets the default interval output to omit units. PLAIN is the default value.
units	Enables interval output to include units. When you enable interval units, the ***DATESTYLE*** (page 903) parameter controls the output. If you enable units and they do not display in the output, check the ***DATESTYLE*** (page 903) parameter value, which must be set to ISO or POSTGRES for interval units to display.

Permissions

No special permissions required.

Output Intervals with Units

The following statement sets the INTERVALSTYLE output to show units:

```
=> SET INTERVALSTYLE TO UNITS;
 SET
=> SELECT INTERVAL '3 2' DAY TO HOUR;
   ?column?
--------------
 3 days 2 hours
(1 row)
```

Output Intervals Without Units

This statement sets the INTERVALSTYLE to plain (no units) on output:

```
=> SET INTERVALSTYLE TO PLAIN;
 SET
=> SELECT INTERVAL '3 2' DAY TO HOUR;
 ?column?
----------
 3 2
(1 row)
```

Displaying the Current Interval OUTPUT Style

Use the ***SHOW*** (page 923) command to display the `INTERVALSTYLE` runtime parameter:

```
=> SHOW INTERVALSTYLE;
     name      | setting
---------------+---------
 intervalstyle | plain
(1 row)
```

See Also

INTERVAL (page 81)

SET LOCALE

Specifies the locale for the current session.

Syntax

```
SET LOCALE TO < ICU-locale-identifier >
```

Parameters

< ICU-locale-identifier >	Specifies the ICU locale identifier to use. By default, the locale for the database is en_US@collation=binary (English as in the United States of America). ICU Locales were developed by the ICU Project. See the ICU User Guide (http://userguide.icu-project.org/locale) for a complete list of

	parameters that can be used to specify a locale. **Note:** The only keyword HP Vertica supports is the COLLATION keyword. **Note:** Single quotes are mandatory to specify the collation.

Permissions

No special permissions required.

Notes

Though not inclusive, the following are some commonly-used locales:

- German (Germany) `de_DE`
- English (Great Britain) `en_GB`
- Spanish (Spain) `es_ES`
- French (France) `fr_FR`
- Portuguese (Brazil) `pt_BR`
- Portuguese (Portugal) `pt_PT`
- Russian (Russia) `ru_RU`
- Japanese (Japan) `ja_JP`
- Chinese (China, simplified Han) `zh_CN`
- Chinese (Taiwan, traditional Han) `zh_Hant_TW`

Session related:

- The locale setting is session scoped and applies to queries only (no DML/DDL) run in that session. You cannot specify a locale for an individual query.
- The default locale for new sessions can be set using a configuration parameter

Query related:

The following restrictions apply when queries are run with locale other than the default en_US@collation=binary:

- Multicolumn NOT IN subqueries are not supported when one or more of the left-side NOT IN columns is of CHAR or VARCHAR data type. For example:

```
=> CREATE TABLE test (x VARCHAR(10), y INT);
=> SELECT ... FROM test WHERE (x,y) NOT IN (SELECT ...);
   ERROR: Multi-expression NOT IN subquery is not supported because a
left hand expression could be NULL
```

  Note: An error is reported even if columns `test.x` and `test.y` have a "NOT NULL" constraint.

- Correlated HAVING clause subqueries are not supported if the outer query contains a GROUP BY on a CHAR or a VARCHAR column. In the following example, the `GROUP BY x` in the outer query causes the error:

```
=> DROP TABLE test CASCADE;
=> CREATE TABLE test (x VARCHAR(10));
```

```
=> SELECT COUNT(*) FROM test t GROUP BY x HAVING x
     IN (SELECT x FROM test WHERE t.x||'a' = test.x||'a' );
   ERROR: subquery uses ungrouped column "t.x" from outer query
```

- Subqueries that use analytic functions in the HAVING clause are not supported. For example:

```
=> DROP TABLE test CASCADE;
=> CREATE TABLE test (x VARCHAR(10));
=> SELECT MAX(x)OVER(PARTITION BY 1 ORDER BY 1)
   FROM test GROUP BY x HAVING x IN (
     SELECT MAX(x) FROM test);
   ERROR: Analytics query with having clause expression that involves
aggregates
   and subquery is not supported
```

DML/DDL related:

- SQL identifiers (such as table names, column names, and so on) can use UTF-8 Unicode characters. For example, the following CREATE TABLE statement uses the ß (German eszett) in the table name:

```
=> CREATE TABLE straße(x int, y int);
   CREATE TABLE
```

- Projection sort orders are made according to the default en_US@collation=binary collation. Thus, regardless of the session setting, issuing the following command creates a projection sorted by `col1` according to the binary collation:

```
=> CREATE PROJECTION p1 AS SELECT * FROM table1 ORDER BY col1;
```

  Note that in such cases, `straße` and strasse would not be near each other on disk.

  Sorting by binary collation also means that sort optimizations do not work in locales other than binary. HP Vertica returns the following warning if you create tables or projections in a non-binary locale:

```
WARNING:  Projections are always created and persisted in the default
HP Vertica locale. The current locale is de_DE
```

- When creating pre-join projections, the projection definition query does not respect the locale or collation setting. This means that when you insert data into the fact table of a pre-join projection, referential integrity checks are not locale or collation aware.

  For example:

```
\locale LDE_S1    -- German
=> CREATE TABLE dim (col1 varchar(20) primary key);
=> CREATE TABLE fact (col1 varchar(20) references dim(col1));
=> CREATE PROJECTION pj AS SELECT * FROM fact JOIN dim
   ON fact.col1 = dim.col1 UNSEGMENTED ALL NODES;
=> INSERT INTO dim VALUES('ß');
=> COMMIT;
```

  The following INSERT statement fails with a "nonexistent FK" error even though 'ß' is in the dim table, and in the German locale 'SS' and 'ß' refer to the same character.

```
=> INSERT INTO fact VALUES('SS');
   ERROR:  Nonexistent foreign key value detected in FK-PK join (fact
x dim)
   using subquery and dim_node0001; value SS
```

```
=> => ROLLBACK;
=> DROP TABLE dim, fact CASCADE;
```

- When the locale is non-binary, the collation function is used to transform the input to a binary string which sorts in the proper order.

  This transformation increases the number of bytes required for the input according to this formula:

  ```
  result_column_width = input_octet_width * CollationExpansion + 4
  ```

  CollationExpansion defaults to 5.
- CHAR fields are displayed as fixed length, including any trailing spaces. When CHAR fields are processed internally, they are first stripped of trailing spaces. For VARCHAR fields, trailing spaces are usually treated as significant characters; however, trailing spaces are ignored when sorting or comparing either type of character string field using a non-BINARY locale.

Examples

This example sets the locale for the session to en_GB (English as in Great Britain).

```
SET LOCALE TO en_GB;
INFO:  Locale: 'en_GB'
INFO:    English (United Kingdom)
INFO:  Short form: 'LEN'
```

You can also use the short form of a locale in this command:

```
SET LOCALE TO LEN;
INFO:  Locale: 'en'
INFO:    English
INFO:  Short form: 'LEN'
```

Single quotes are mandatory to specify the collation:

```
SET LOCALE TO 'tr_tr@collation=standard';
INFO: Locale: 'tr_TR@collation=standard'
Standard collation: 'LTR'
Turkish (Turkey, collation=standard)  Türkçe (Türkiye, Sıralama=standard)
SET
```

See Also

Implement Locales for International Data Sets and Appendix: Locales in the Administrator's Guide

SET ROLE

Enables a role for the current user's current session. The user will gain the permissions that have been granted to the role.

Syntax

```
SET ROLE { role [, ...] | NONE | ALL | DEFAULT }
```

Parameters

role [, ...] | NONE | ALL | DEFAULT	The name of one or more roles to set as the current role, or one of the following keywords: ▪ NONE disables all roles for the current session. ▪ ALL enables all of the roles to which the user has access. ▪ DEFAULT sets the current role to the user's default role.

Permissions

You can only set a role that has been granted to you. Run the SHOW AVAILABLE_ROLES command to get a list of the roles available to the user.

Notes

- The default role is the first role that was assigned to the user or set by the superuser via the ***ALTER USER*** (page 679) statement's DEFAULT ROLE argument.
- Enabling a role does not affect any other roles that are currently enabled. A user session can have more than one role enabled at a time. The user's permissions are the union of all the roles that are currently active, plus any permissions granted directly to the user.

Example

```
=> SHOW ENABLED_ROLES; -- Which roles are active now.
     name      |          setting
---------------+---------------------------
 enabled roles |
(1 row)

=> SHOW AVAILABLE_ROLES; -- All roles the user can access
      name       |          setting
-----------------+---------------------------
 available roles | applogs, appadmin, appuser
(1 row)

=> SET ROLE applogs;
SET
=> SHOW ENABLED_ROLES;
     name      | setting
---------------+---------
 enabled roles | applogs
(1 row)

=> SET ROLE appuser;
SET
=> SHOW ENABLED_ROLES;
     name      |     setting
---------------+-----------------
 enabled roles | applogs, appuser
```

```
(1 row)

=> SET ROLE NONE; -- disable all roles
SET
dbadmin=> SHOW ENABLED_ROLES;
     name      | setting
---------------+---------
 enabled roles |
(1 row)
```

SET SEARCH_PATH

Specifies the order in which HP Vertica searches schemas when a SQL statement contains an unqualified table name.

HP Vertica provides the SET search_path statement instead of the CURRENT_SCHEMA statement found in some other databases.

Syntax 1

```
SET SEARCH_PATH [ TO | = ] schemaname [ , ... ]
```

Syntax 2

```
SET SEARCH_PATH [ TO | = ] default
```

Parameters

schemaname	A comma-delimited list of schemas that indicates the order in which HP Vertica searches schemas when a SQL statement contains an unqualified table name. The default value for this parameter is '"$user", public' Where: ▪ $user is the schema with the same name as the current user. If the schema does not exist, $user is ignored. ▪ public is the public database. Public is ignored if there is no schema named 'public'.
default	Returns the search path to the default value of '"$user", public'.

Permissions

No special permissions required.

Notes

The first schema named in the search path is called the current schema. The current schema is the first schema that HP Vertica searches. It is also the schema in which HP Vertica creates new tables if the ***CREATE TABLE*** (page 770) command does not specify a schema name.

Examples

The following example shows the current search path settings:

```
=> SHOW SEARCH_PATH;
    name     |                       setting
-------------+-------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

This command sets the order in which HP Vertica searches schemas to T1, U1, and V1:

```
=> SET SEARCH_PATH TO T1, U1, V1;
```

This command returns the search path to the default of '"$user", public':

```
=> SET SEARCH_PATH = default;
```

SET SESSION AUTOCOMMIT

Sets whether statements automatically commit their transactions on completion. This statement is primarily used by the client drivers to enable and disable autocommit, you should never have to directly call it.

Syntax

```
SET SESSION AUTOCOMMIT TO { ON | OFF }
```

Parameters

ON	Enable autocommit. Statements automatically commit their transactions when they complete. This is the default setting for connections made using the HP Vertica client libraries.
OFF	Disable autocommit. Transactions are not automatically committed. This is the default for interactive sessions (connections made through vsql).

Permissions

No special permissions required.

See Also

HP Vertica Client Library Overview in the Programmer's Guide.

SET SESSION CHARACTERISTICS

Sets the transaction characteristics for the isolation level and access mode (read/write or read-only). Setting the transaction mode affects subsequent transactions of a user session.

Syntax

```
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
where transaction_mode is one of:
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ { ONLY | WRITE }
```

Parameters

Isolation level, described in the following table, determines what data the transaction can access when other transactions are running concurrently. The isolation level cannot be changed after the first query (`SELECT`) or DML statement (`INSERT`, `DELETE`, `UPDATE`) if a transaction has run. A transaction retains its isolation level until it completes, even if the session's transaction isolation level changes mid-transaction. HP Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations are always run at `SERIALIZABLE` isolation level to ensure consistency.

`SERIALIZABLE`	Sets the strictest level of SQL transaction isolation. This level emulates transactions serially, rather than concurrently. It holds locks and blocks write operations until the transaction completes. Not recommended for normal query operations.
`REPEATABLE READ`	Automatically converted to SERIALIZABLE by HP Vertica.
`READ COMMITTED`	(Default) Allows concurrent transactions. Use READ COMMITTED isolation or Snapshot Isolation for normal query operations, but be aware that there is a subtle difference between them. (See section below this table.)
`READ UNCOMMITTED`	Automatically converted to READ COMMITTED by HP Vertica.
`READ {WRITE | ONLY}`	Determines whether the transaction is read/write or read-only. Read/write is the default. Setting the transaction session mode to read-only disallows the following SQL commands, but does not prevent all disk write operations: ▪ 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 run is among those listed.

Permissions

No special permissions required.

Understanding READ COMMITTED and 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 sees the transaction's own changes regardless of isolation level.

Using SERIALIZABLE Transaction Isolation

Setting the transaction isolation level to SERIALIZABLE does not apply to temporary tables. Temporary tables are isolated by their transaction scope.

Applications using SERIALIZABLE must be prepared to retry transactions due to serialization failures.

Setting READ ONLY Transaction Mode

This example sets a Read-only transaction level:

```
=> SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED READ
ONLY;
SET
```

SET SESSION MEMORYCAP

Specifies a limit on the amount of memory that any request issued by the session can consume.

Syntax

```
SET SESSION MEMORYCAP 'memory-limit' | = default
```

Parameters

memory-limit | = default	The maximum amount of memory the session can use. To set a value, supply number followed by a unit. Units can be one of the following: ▪ % percentage of total memory available to the Resource Manager. (In this case, size must be 0-100). ▪ K Kilobytes ▪ M Megabytes ▪ G Gigabytes ▪ T Terabytes If you use the value = default the session's MEMORYCAP is set to the user's MEMORYCAP value.

Permissions

- This command requires superuser privileges if the MEMORYCAP is being increased over the user's MEMORYCAP limit (see ***CREATE USER*** (page 801) for details).

- Non-superusers can change this value to anything below or equal to their MEMORYCAP limit.

Notes

The MEMORYCAP limit is per user session, not an overall cap on the user's memory usage. A user could spawn multiple sessions, each of which could use up to the limit set by the MEMORYCAP.

Example

The following command sets a memorycap of 4 gigabytes on the session:

```
=> SET SESSION MEMORYCAP '4G';
```

To return the memorycap to the previous setting:

```
=> SET SESSION MEMORYCAP NONE;
=> SHOW MEMORYCAP;
   name     |  setting
-----------+-----------
 memorycap | UNLIMITED
(1 row)
```

See Also

ALTER RESOURCE POOL (page 663)

CREATE RESOURCE POOL (page 753)

CREATE USER (page 801)

DROP RESOURCE POOL (page 819)

SET SESSION RESOURCE POOL (page 916)

Managing Workloads in the Administrator's Guide

SET SESSION RESOURCE_POOL

Associates the user session with the specified resource pool.

Syntax

```
SET SESSION RESOURCE_POOL = { pool-name | default }
```

Parameters

pool-name | default	Specifies the name of the resource pool to be associated with session. If you use the value `default`, then the session's resource pool is set to the default resource pool for the user.

Permissions

- This command requires non-superusers to have USAGE privileges for the resource pool.
- Superusers can assign their session to any resource pool they want.

- To set the resource pool session, the user needs USAGE privilege on the resource pool.

Notes

The resource pool must exist.

See Also

ALTER RESOURCE POOL (page 663)

CREATE RESOURCE POOL (page 753)

CREATE USER (page 801)

DROP RESOURCE POOL (page 819)

GRANT (Resource Pool) (page 834)

SET SESSION MEMORYCAP (page 915)

Managing Workloads in the Administrator's Guide

SET SESSION RUNTIMECAP

Sets the maximum amount of time a session's query can run.

Syntax

```
SET SESSION RUNTIMECAP [ 'duration' | NONE | = default ]
```

Parameters

'duration' | NONE | DEFAULT	One of three values: ▪ An interval such as `'1 minute'` or `'100 seconds'` (see ***Interval Values*** (page 37) for a full explanation) setting the maximum amount of time this session's queries should be allowed to run. ▪ `NONE` which eliminates any limit on the amount of time the session's queries can run (the default value). ▪ `= default` which sets the session's RUNTIMECAP to the user's RUNTIMECAP value. You must include the equals (=) sign before the `default` keyword.

Notes

- The largest allowable RUNTIMECAP value is 1 year (365 days).
- If RUNTIMECAP is also set for the user or the resource pool, HP Vertica uses the shortest limit.

- This command requires superuser privileges if the RUNTIMECAP is being increased over the user's RUNTIMECAP limit.
- Normal users can change the RUNTIMECAP of their own sessions to any value below their own RUNTIMECAP. They cannot increase the RUNTIMECAP beyond any limit set for them by the superuser.
- The timeout is not precise, so a query may run a little longer than the value set in RUNTIMECAP.
- Queries that violate the RUNTIMECAP are terminated with the error message `Execution time exceeded run time cap of <cap_duraation>.`
- `SHOW RUNTIMECAP` (below) shows only the session RUNTIMECAP; it does not show the user or resource pool RUNTIMECAP.

Example

The following command sets the session's RUNTIMECAP to 10 minutes:

```
=> SET SESSION RUNTIMECAP '10 minutes';
```

To return the RUNTIMECAP to the user's default setting:

```
=> SET SESSION RUNTIMECAP =DEFAULT;
SET
=> SHOW RUNTIMECAP;
    name     |  setting
-------------+-----------
 runtimecap | UNLIMITED
(1 row)
```

See Also

CREATE USER (page 801)

ALTER USER (page 679)

Managing Workloads in the Administrator's Guide

SET SESSION TEMPSPACECAP

Sets the maximum amount of temporary file storage space that any request issued by the session can consume.

Syntax

```
SET SESSION TEMPSPACECAP 'space-limit' | = default | NONE
```

Parameters

'space-limit'	The maximum amount of temporary file space the session can use. To set a limit, use a numeric value followed by a unit (for example: '10G'). The unit can be one of the following: ▪ % percentage of total temporary storage space

	available. (In this case, the numeric value must be 0-100). ▪ K Kilobytes ▪ M Megabytes ▪ G Gigabytes ▪ T Terabytes Setting this value to `= default` sets the session's TEMPSPACECAP to the user's TEMPSPACECAP value. Setting this value to `NONE` results in the session having unlimited temporary storage space. This is the default value.

Permissions

- This command requires superuser privileges to increase the TEMPSPACECAP over the user's TEMPSPACECAP limit.
- Regular users can change the TEMPSPACECAP associated with their own sessions to any value less than or equal to their own TEMPSPACECAP. They cannot increase its value beyond their own TEMPSPACECAP value.

Notes

- This limit is per session, not per user. A user could open multiple sessions, each of which could use up to the TEMPSPACECAP.
- Any execution plan that exceeds its TEMPSPACECAP usage results in the error:
  `ERROR: Exceeded temp space cap.`

Example

The following command sets a TEMPSPACECAP of 20gigabytes on the session:

```
=> SET SESSION TEMPSPACECAP '20G';
SET
=> SHOW TEMPSPACECAP;
     name      | setting
---------------+----------
 tempspacecap | 20971520
(1 row)
```

Note: SHOW displays the TEMPSPACECAP in kilobytes.

To return the memorycap to the previous setting:

```
=> SET SESSION TEMPSPACECAP NONE;
SET
=> SHOW TEMPSPACECAP;
     name      |  setting
---------------+-----------
 tempspacecap | UNLIMITED
(1 row)
```

See Also

ALTER USER (page 679)

CREATE USER (page 801)

Managing Workloads in the Administrator's Guide

SET STANDARD_CONFORMING_STRINGS

Treats backslashes as escape characters for the current session.

Syntax

```
SET STANDARD_CONFORMING_STRINGS TO { ON | OFF }
```

Parameters

ON	Makes ordinary string literals ('...') treat back slashes (\) literally. This means that back slashes are treated as string literals, not escape characters. (This is the default.)
OFF	Treats back slashes as escape characters.

Permissions

No special permissions required.

Notes

- This statement works under vsql only.
- When standard conforming strings are on, HP Vertica supports SQL:2008 string literals within Unicode escapes.
- Standard conforming strings must be ON to use Unicode-style string literals (`U&'\nnnn'`).

TIP: To set conforming strings across all sessions (permanently), use the `StandardConformingStrings` as described in Internationalization Parameters in the Administrator's Guide.

Examples

The following example shows how to turn off conforming strings for the session.

```
=> SET STANDARD_CONFORMING_STRINGS TO OFF;
```

The following command lets you verify the settings:

```
=> SHOW STANDARD_CONFORMING_STRINGS;
            name             | setting
-----------------------------+---------
 standard_conforming_strings | off
(1 row)
```

The following example shows how to turn on conforming strings for the session.

```
=> SET STANDARD_CONFORMING_STRINGS TO ON;
```

See Also

ESCAPE_STRING_WARNING (page 905)

SET TIME ZONE

Changes the TIME ZONE run-time parameter for the current session.

Syntax

```
SET TIME ZONE TO { value | 'value' }
```

Parameters

value	Is one of the following: ▪ One of the time zone names specified in the tz database, as described in ***Sources for Time Zone and Daylight Saving Time Data http://www.twinsun.com/tz/tz-link.htm***. ***Time Zone Names for Setting TIME ZONE*** (page 922) listed in the next section are for convenience only and could be out of date. ▪ A signed integer representing an offset from UTC in hours ▪ An ***interval value*** (page 37)

Permissions

No special permissions required.

Notes

- TIME ZONE is a synonym for TIMEZONE. Both are allowed in HP Vertica syntax.
- The built-in constants LOCAL and DEFAULT, which set the time zone to the one specified in the TZ environment variable or, if TZ is undefined, from the operating system time zone. See Set the Default Time Zone and Using Time Zones with HP Vertica in the Installation Guide.
- When using a Country/City name, do not omit the country or the city. For example:
  ```
  SET TIME ZONE TO 'Africa/Cairo'; -- valid
  SET TIME ZONE TO 'Cairo'; -- invalid
  ```
- Include the required keyword TO.
- Positive integer values represent an offset east from UTC.
- The ***SHOW*** (page 923) command displays the run-time parameters.
- If you set the timezone using POSIX format, the timezone abbreviation you use overrides the default timezone abbreviation. If the DATESTYLE is set to POSTGRES, the timezone abbreviation you use is also used when converting a timestamp to a string.

Examples

```
=> SET TIME ZONE TO DEFAULT;
=> SET TIME ZONE TO 'PST8PDT'; -- Berkeley, California
```

```
=> SET TIME ZONE TO 'Europe/Rome'; -- Italy
=> SET TIME ZONE TO '-7'; -- UDT offset equivalent to PDT
=> SET TIME ZONE TO INTERVAL '-08:00 HOURS';
```

See Also

Using Time Zones with HP Vertica in the Installation Guide

Time Zone Names for Setting TIME ZONE

The following time zone names are recognized by HP Vertica as valid settings for the SQL time zone (the TIME ZONE run-time parameter).

Note: The names listed here are for convenience only and could be out of date. Refer to the ***Sources for Time Zone and Daylight Saving Time Data http://www.twinsun.com/tz/tz-link.htm*** page for precise information.

These names are not the same as the names shown in `/opt/vertica/share/timezonesets`, which are recognized by HP Vertica in date/time input values. The TIME ZONE names shown below imply a local daylight-savings time rule, where date/time input names represent 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.

In addition to the names listed in the table, HP Vertica accepts 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. 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 TIME ZONE TO FOOBANKO` works, leaving the system effectively using a rather peculiar abbreviation for GMT.

Time Zone
Africa
America
Antarctica
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

SHOW

Displays run-time parameters for the current session.

Syntax

```
SHOW { name | ALL }
```

Parameters

name	AUTOCOMMIT	Displays whether statements automatically commit their transactions when they complete.
	AVAILABLE_ROLES	Lists all roles available to the user.
	DATESTYLE	Displays the current style of date values. See ***SET DATESTYLE*** (page 903).

	ENABLED_ROLES	Displays the roles enabled for the current session. See ***SET ROLE*** (page 910).
	ESCAPE_STRING_WARNING	Displays whether warnings are issued when backslash escapes are found in strings. See ***SET ESCAPE_STRING_WARNING*** (page 905).
	INTERVALSTYLE	Displays whether units are output when printing intervals. See ***SET INTERVALSTYLE*** (page 906).
	LOCALE	Displays the current locale. See ***SET LOCALE***.
	MEMORYCAP	Displays the maximum amount of memory that any request use. See ***SET MEMORYCAP*** (page 915).
	RESOURCE_POOL	Displays the resource pool that the session is using. See ***SET RESOURCE POOL*** (page 916).
	RUNTIMECAP	Displays the maximum amount of time that queries can run in the session. See ***SET RUNTIMECAP*** (page 917)
	SEARCH_PATH	Displays the order in which HP Vertica searches schemas. See ***SET SEARCH_PATH*** (page 912).
	SESSION_CHARACTERISTICS	Displays the transaction characteristics. See ***SET SESSION CHARACTERISTICS*** (page 914).
	STANDARD_CONFORMING_STRINGS	Displays whether backslash escapes are enabled for the session. See ***SET STANDARD_CONFORMING_STRINGS*** (page 920).
	TEMPSPACECAP	Displays the maximum amount of temporary file space that queries can use in the session. See ***SET TEMPSPACECAP*** (page 918).
	TIMEZONE	Displays the timezone set in the current session. See ***SET TIMEZONE*** (page 921).
	TRANSACTION_ISOLATION	Displays the current transaction isolation setting, as described in ***SET SESSION CHARACTERISTICS*** (page 914).
	TRANSACTION_READ_ONLY	Displays the current setting, as described in ***SET SESSION CHARACTERISTICS*** (page 914).
ALL		Shows all run-time parameters.

Permissions

No special permissions required.

Displaying all current run-time parameter settings

The following command returns all the run-time parameter settings:

```
=> SHOW ALL;
           name             |                      setting
----------------------------+-------------------------------------------------
 locale                     | en_US@collation=binary (LEN_KBINARY)
 autocommit                 | off
 standard_conforming_strings | on
```

```
 escape_string_warning       | on
 datestyle                   | ISO, MDY
 intervalstyle               | plain
 timezone                    | US/Eastern
 search_path                 | "$user", public, v_catalog, v_monitor, v_internal
 transaction_isolation       | READ COMMITTED
 transaction_read_only       | false
 resource_pool               | general
 memorycap                   | UNLIMITED
 tempspacecap                | UNLIMITED
 runtimecap                  | UNLIMITED
 enabled roles               |
 available roles             | applogs, appadmin
(15 rows)
```

Displaying current search path settings

The following command returns the search path settings:

```
=> SHOW SEARCH_PATH;
    name     |                        setting
-------------+-----------------------------------------------------
 search_path | "$user", public, v_catalog, v_monitor, v_internal
(1 row)
```

Displaying the Transaction Isolation Level

The following command shows the session transaction isolation level:

```
=> SHOW TRANSACTION ISOLATION LEVEL;
         name          |    setting
-----------------------+----------------
 transaction_isolation | READ COMMITTED
(1 row)
```

The next command returns the setting for SESSION CHARACTERISTICS AS TRANSACTION. False indicates that the default read/write is the current setting:

```
=> SHOW transaction_read_only;
         name          | setting
-----------------------+---------
 transaction_read_only | false
(1 row)
```

To change to read only, you'd need to enter:

```
=> SET SESSION CHARACTERISTICS AS TRANSACTION READ ONLY;
```

Now the same SHOW command returns true:

```
=> SHOW transaction_read_only;
         name          | setting
-----------------------+---------
 transaction_read_only | true
(1 row)
```

START TRANSACTION

Starts a transaction block.

Syntax

```
START TRANSACTION [ isolation_level ]
```

where *isolation_level* is one of:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
}
READ { ONLY | WRITE }
```

Parameters

Isolation level, described in the following table, determines what data the transaction can access when other transactions are running concurrently. The isolation level cannot be changed after the first query (`SELECT`) or DML statement (`INSERT`, `DELETE`, `UPDATE`) has run. A transaction retains its isolation level until it completes, even if the session's transaction isolation level changes mid-transaction. HP Vertica internal processes (such as the Tuple Mover and refresh operations) and DDL operations are always run at `SERIALIZABLE` isolation level to ensure consistency.

`WORK | TRANSACTION`	Have no effect; they are optional keywords for readability.
`ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }`	▪ SERIALIZABLE—Sets the strictest level of SQL transaction isolation. This level emulates transactions serially, rather than concurrently. It holds locks and blocks write operations until the transaction completes. Not recommended for normal query operations. ▪ REPEATABLE READ—Automatically converted to SERIALIZABLE by HP Vertica. ▪ READ COMMITTED (Default)—Allows concurrent transactions. Use READ COMMITTED isolation or Snapshot Isolation for normal query operations, but be aware that there is a subtle difference between them. (See section below this table.) ▪ READ UNCOMMITTED—Automatically converted to READ COMMITTED by HP Vertica.
`READ {WRITE | ONLY}`	Determines whether the transaction is read/write or read-only. Read/write is the default. Setting the transaction session mode to read-only disallows the following SQL commands, but does not prevent all disk write operations: ▪ 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 run is among those listed.

Permissions

No special permissions required.

Notes

BEGIN (page 682) performs the same function as START TRANSACTION.

See Also

- Transactions
- Creating and Rolling Back Transactions
- ***COMMIT*** (page 697)
- ***END*** (page 827)
- ***ROLLBACK*** (page 867)

TRUNCATE TABLE

Removes all storage associated with a table, while preserving the table definitions. TRUNCATE TABLE auto-commits the current transaction after statement execution and cannot be rolled back.

Syntax

```
TRUNCATE TABLE [[db-name.]schema.]table
```

Parameters

[[db-name.]schema.]	[Optional] Specifies the database name and optional schema name. Using a database name identifies objects that are not unique within the current search path (see Setting Search Paths). You must be connected to the database you specify, and you cannot change objects in other databases. Specifying different database objects lets you qualify database objects as explicitly as required. For example, you can use a database and a schema name (`mydb.myschema`).
table	Specifies the name of a base table or temporary table. Cannot truncate an external table.

Permissions

Superuser or table owner. A schema owner can drop a table but cannot truncate a table.

Notes

- To truncate an `ON COMMIT DELETE ROWS` temporary table without ending the transaction, use ***DELETE FROM temp_table*** (page 807) syntax.

  Note: The effect of DELETE FROM depends on the table type. If the table is specified as ON COMMIT DELETE ROWS, then DELETE FROM works like TRUNCATE TABLE; otherwise it behaves like a normal delete in that it does not truncate the table.

- After truncate operations complete, the data recovers from that current epoch onward. Because TRUNCATE TABLE removes table history, AT EPOCH queries return nothing. TRUNCATE TABLE behaves the same when you have data in WOS, ROS, or both, as well as for unsegmented/segmented projections.
- If the operation cannot obtain an ***O Lock*** (page 1037) on the table(s), HP Vertica attempts to close any internal Tuple Mover (TM) sessions running on the same table(s) so that the operation can proceed. Explicit TM operations that are running in user sessions are not closed. If an explicit TM operation is running on the table, then the operation cannot proceed until the explicit TM operation completes.

Examples

For examples about how to use TRUNCATE, see Dropping and Truncating Tables in the Administrator's Guide.

See Also

DELETE (page 807), ***DROP TABLE*** (page 823), and ***LOCKS*** (page 1037)

Transactions in the Concepts Guide

Deleting Data and Best Practices for DELETE and UPDATE in the Administrator's Guide

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. By default, UPDATE uses the WOS and if the WOS fills up, overflows to the ROS.

Syntax

```
UPDATE [ /*+ direct */ ] [ /*+ label(label-name)*/ ]
... [[db-name.]schema.]table-reference (on page 876) [AS] alias
... SET column =
... { expression | DEFAULT } [ , ... ]
... [ FROM from-list ]
... [ WHERE clause (page 901) ]
```

Parameters

`/*+ direct */`	Writes the data directly to disk (ROS) bypassing memory (WOS).
	HP Vertica accepts optional spaces before and after the plus (+) sign and the `direct` hint. Space characters between the opening `/*` or the closing `*/` are not permitted. The following directives are all acceptable:
	`/*+direct*/`
	`/* + direct*/`
	`/*+ direct*/`
	`/*+direct */`
	Note: If you update using the `direct` hint, you still need to issue a `COMMIT` or `ROLLBACK` command to finish the transaction.

`/*+ label (label-name)*/`	Passes a user-defined label to a query as a hint, letting you quickly identify labeled queries for profiling and debugging. See Query Labeling in the Administrator's Guide.
[[db-name.]schema.]	[Optional] Specifies the schema name. Using a schema identifies objects that are not unique within the current search path (see Setting Schema Search Paths). You can optionally precede a schema with a database name, but you must be connected to the database you specify. You cannot make changes to objects in other databases. The ability to specify different database objects (from database and schemas to tables and columns) lets you qualify database objects as explicitly as required. For example, you can specify a table and column (`mytable.column1`), a schema, table, and column (`myschema.mytable.column1`), and as full qualification, a database, schema, table, and column (`mydb.myschema.mytable.column1`).

table-reference	*table-primary* or *joined-table*: *table-primary* — Specifies an optionally qualified table name with optional table aliases, column aliases, and outer joins. *joined-table* — Specifies an outer join. You cannot update a projection.
alias	Specifies a temporary name to be used for references to the table.
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*** (page 876) of a `SELECT` command. Note that the target table must not appear in the *from-list*.

Permissions

Table owner or user with GRANT OPTION is grantor.

- UPDATE privilege on table
- USAGE privilege on schema that contains the table
- SELECT privilege on the table when executing an UPDATE statement that references table column values in a WHERE or SET clause

Notes

- Subqueries and joins are permitted in UPDATE statements, which is useful for updating values in a table based on values that are stored in other tables. UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need to be specified in the SET clause. Columns that are not explicitly modified retain their previous values. On successful completion, an UPDATE operation returns a count, which represents the number of rows updated. A count of 0 is not an error; it means that no rows matched the condition.

  See Subqueries in UPDATE and DELETE Statements in the Programmer's Guide.
- The table you specify in the UPDATE list cannot also appear in the FROM list (no self joins); for example, the following UPDATE statement is not allowed:

```
=> BEGIN;
=> UPDATE result_table
   SET address='new' || r2.address
   FROM result_table r2
   WHERE r2.cust_id = result_table.cust_id + 10;
   ERROR:  Self joins in UPDATE statements are not allowed
   DETAIL:  Target relation result_table also appears in the FROM list
```

- If the joins specified in the WHERE predicate produce more than one copy of the row in the table to be updated, the new value of the row in the table is chosen arbitrarily.
- UPDATE inserts new records into the WOS and marks the old records for deletion.
- You cannot UPDATE columns that have primary key or foreign key referential integrity constraints.
- To use the ***DELETE*** (page 807) or UPDATE commands with a ***WHERE clause*** (page 901), you must have both SELECT and DELETE privileges on the table.

Examples

In the FACT table, modify the PRICE column value for all rows where the COST column value is greater than 100:

```
=> UPDATE FACT SET PRICE = PRICE - COST * 80 WHERE COST > 100;
```

In the Retail.CUSTOMER table, set the STATE column to 'NH' when the CID column value is greater than 100:

```
=> UPDATE Retail.CUSTOMER SET STATE = 'NH' WHERE CID > 100;
```

To use table aliases in UPDATE queries, consider the following two tables:

```
=> SELECT * FROM Result_Table;
 cust_id |      address
---------+--------------------
      20 | Lincoln Street
      30 | Beach Avenue
      30 | Booth Hill Road
      40 | Mt. Vernon Street
      50 | Hillside Avenue
(5 rows)

=> SELECT * FROM New_Addresses;
 new_cust_id | new_address
-------------+---------------
          20 | Infinite Loop
          30 | Loop Infinite
          60 | New Addresses
(3 rows)
```

The following query and subquery use table aliases to update the `address` column in Result_Table (alias `r`) with the new address from the corresponding column in the New_Addresses table (alias `n`):

```
=> UPDATE Result_Table r
   SET address=n.new_address
   FROM New_Addresses n
   WHERE r.cust_id = n.new_cust_id;
```

The Result_Table table reflects the `address` field updates made for customer IDs 20 and 30:

```
=> SELECT * FROM Result_Table ORDER BY cust_id;
 cust_id |     address
---------+-----------------
      20 | Infinite Loop
      30 | Loop Infinite
```

```
      30 | Loop Infinite
      40 | Mt. Vernon Street
      50 | Hillside Avenue
(5 rows)
```

For more information about nesting subqueries within an UPDATE statement, see Subqueries in UPDATE and DELETE in the Programmer's Guide.

HP Vertica System Tables

HP Vertica provides system tables that let you monitor your database. Query these tables the same way you perform query operations on base or temporary tables—by using SELECT statements.

For more information, see the following sections in the Administrator's Guide:

- Using System Tables
- Monitoring Vertica

V_CATALOG Schema

The system tables in this section reside in the `v_catalog` schema. These tables provide information (metadata) about the objects in a database; for example, tables, constraints, users, projections, and so on.

ALL_TABLES

Provides summary information about the tables in HP Vertica.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	The name of the schema that contains the table.
TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog that identifies the table.
TABLE_NAME	VARCHAR	The table name.
TABLE_TYPE	VARCHAR	The type of table, which can be one of the following: ▪ TABLE ▪ SYSTEM TABLE ▪ VIEW ▪ GLOBAL TEMPORARY ▪ LOCAL TEMPORARY
REMARKS	VARCHAR	A brief comment about the table. You define this field by using the ***COMMENT ON TABLE*** (page 693) and ***COMMENT ON VIEW*** (page 695) commands.

Example

```
onenode=> SELECT DISTINCT table_name, table_type FROM all_tables
          WHERE table_name ILIKE 't%';
```

```
       table_name       |  table_type
------------------------+--------------
 types                  | SYSTEM TABLE
 trades                 | TABLE
 tuple_mover_operations | SYSTEM TABLE
 tables                 | SYSTEM TABLE
 tuning_recommendations | SYSTEM TABLE
 testid                 | TABLE
 table_constraints      | SYSTEM TABLE
 transactions           | SYSTEM TABLE
(8 rows)

onenode=> SELECT table_name, table_type FROM all_tables
          WHERE table_name ILIKE 'my%';

 table_name | table_type
------------+------------
 mystocks   | VIEW
(1 row)


=> SELECT * FROM all_tables LIMIT 4;
-[ RECORD 1 ]-----------------------------------------
schema_name | v_catalog
table_id    | 10206
table_name  | all_tables
table_type  | SYSTEM TABLE
remarks     | A complete listing of all tables and views
-[ RECORD 2 ]-----------------------------------------
schema_name | v_catalog
table_id    | 10000
table_name  | columns
table_type  | SYSTEM TABLE
remarks     | Table column information
-[ RECORD 3 ]-----------------------------------------
schema_name | v_catalog
table_id    | 10054
table_name  | comments
table_type  | SYSTEM TABLE
remarks     | User comments on catalog objects
-[ RECORD 4 ]-----------------------------------------
schema_name | v_catalog
table_id    | 10134
table_name  | constraint_columns
table_type  | SYSTEM TABLE
remarks     | Table column constraint information
```

COLUMNS

Provides table column information.

Column Name	Data Type	Description
TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog that identifies the table.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed in the database.
TABLE_NAME	VARCHAR	The table name for which information is listed in the database.
IS_SYSTEM_TABLE	BOOLEAN	Indicates if the table is a system table, where *t* is true and *f* is false.
COLUMN_ID	VARCHAR	A unique VARCHAR ID, assigned by the HP Vertica catalog, that identifies a column in a table.
COLUMN_NAME	VARCHAR	The column name for which information is listed in the database.
DATA_TYPE	VARCHAR	The data type assigned to the column; for example `VARCHAR(16), INT, FLOAT`.
DATA_TYPE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the data type.
DATA_TYPE_LENGTH	INTEGER	The maximum allowable length of the data type.
CHARACTER_MAXIMUM_LENGTH	VARCHAR	The maximum allowable length of the column.
NUMERIC_PRECISION	INTEGER	The number of significant decimal digits.
NUMERIC_SCALE	INTEGER	The number of fractional digits.
DATETIME_PRECISION	INTEGER	For `TIMESTAMP` data type, returns the declared precision; returns NULL if no precision was declared.
INTERVAL_PRECISION	INTEGER	The number of fractional digits retained in the seconds field.
ORDINAL_POSITION	INTEGER	The position of the column respective to other columns in the table.
IS_NULLABLE	BOOLEAN	Indicates whether the column can contain NULL values, where *t* is true and *f* is false.
COLUMN_DEFAULT	VARCHAR	The default value of a column, such as empty or expression.
IS_IDENTITY	BOOLEAN	True if the column is an identity column. See ***column-constraint*** (page 783).

Example

Retrieve table and column information from the COLUMNS table:

```
=> SELECT table_schema, table_name, column_name, data_type, is_nullable
   FROM columns WHERE table_schema = 'store'  AND data_type = 'Date';

 table_schema |    table_name     |      column_name       | data_type |
is_nullable
--------------+-------------------+------------------------+-----------+------
-------
 store        | store_dimension   | first_open_date        | Date      | f
 store        | store_dimension   | last_remodel_date      | Date      | f
 store        | store_orders_fact | date_ordered           | Date      | f
 store        | store_orders_fact | date_shipped           | Date      | f
 store        | store_orders_fact | expected_delivery_date | Date      | f
 store        | store_orders_fact | date_delivered         | Date      | f
6 rows)
```

NULL results indicate that those columns were not defined. For example, given the following table, the result for the DATETIME_PRECISION column is NULL because no precision was declared:

```
=> CREATE TABLE c (c TIMESTAMP);
CREATE TABLE
=> SELECT table_name, column_name, datetime_precision FROM columns WHERE
table_name = 'c';
 table_name | column_name | datetime_precision
------------+-------------+--------------------
 c          | c           |
(1 row)
```

In this example, the DATETIME_PRECISION column returns 4 because the precision was declared as 4 in the CREATE TABLE statement:

```
=> DROP TABLE c;
=> CREATE TABLE c (c TIMESTAMP(4));
CREATE TABLE
=> SELECT table_name, column_name, datetime_precision FROM columns WHERE
table_name = 'c';
 table_name | column_name | datetime_precision
------------+-------------+--------------------
 c          | c           |                  4
```

An identity column is a sequence available only for numeric column types. To identify what column in a table, if any, is an identity column, search the COLUMNS table to find the identity column in a table testid:

```
=> CREATE TABLE testid (c1 IDENTITY(1, 1, 1000), c2 INT);
=> \x
Expanded display is on.
=> SELECT * FROM COLUMNS WHERE is_identity='t' AND table_name='testid';
-[ RECORD 1 ]------------+-------------------
table_id                 | 45035996273719486
table_schema             | public
table_name               | testid
is_system_table          | f
```

```
column_id                | 45035996273719486-1
column_name              | c1
data_type                | int
data_type_id             | 6
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 1
is_nullable              | f
column_default           |
is_identity              | t
```

Use the SEQUENCES table to get detailed information about the sequence in testid:

```
=> SELECT * FROM sequences WHERE identity_table_name='testid';
-[ RECORD 1 ]-------+--------------------
sequence_schema     | public
sequence_name       | testid_c1_seq
owner_name          | dbadmin
identity_table_name | testid
session_cache_count | 1000
allow_cycle         | f
output_ordered      | f
increment_by        | 1
minimum             | 1
maximum             | 9223372036854775807
current_value       | 0
sequence_schema_id  | 45035996273704976
sequence_id         | 45035996273719488
owner_id            | 45035996273704962
identity_table_id   | 45035996273719486
```

For more information about sequences and identity columns, see Using Named Sequences.

COMMENTS

Returns information about comments associated with objects in the database.

Column Name	Data Type	Description
COMMENT_ID	INTEGER	The comment's internal ID number
OBJECT_ID	INTEGER	The internal ID number of the object associated with the comment

OBJECT_TYPE	VARCHAR	The type of object associated with the comment. Possible values are: ▪ COLUMN ▪ CONSTRAINT ▪ FUNCTION ▪ LIBRARY ▪ NODE ▪ PROJECTION ▪ SCHEMA ▪ SEQUENCE ▪ TABLE ▪ VIEW
OBJECT_SCHEMA	VARCHAR	The schema containing the object.
OBJECT_NAME	VARCHAR	The name of the object associated with the comment.
OWNER_ID	VARCHAR	The internal ID of the owner of the object.
OWNER_NAME	VARCHAR	The object owner's name.
CREATION_TIME	TIMESTAMPTZ	When the comment was created.
LAST_MODIFIED_TIME	TIMESTAMPTZ	When the comment was last modified.
COMMENT	VARCHAR	The text of the comments.

CONSTRAINT_COLUMNS

Records information about table column constraints.

Column Name	Data Type	Description
CONSTRAINT_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the constraint.
TABLE_SCHEMA	VARCHAR	Name of the schema that contains this table.
TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog that identifies the table.
TABLE_NAME	VARCHAR	Name of the table in which the column resides.
COLUMN_NAME	VARCHAR	Name of the column that is constrained.
CONSTRAINT_NAME	VARCHAR	Constraint name for which information is listed.

CONSTRAINT_TYPE	CHAR	Is one of: ▪ c — check is reserved, but not supported ▪ f — foreign ▪ n — not null ▪ p — primary ▪ u — unique ▪ d — determines
REFERENCE_TABLE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the referenced table
REFERENCE_TABLE_SCHEMA	VARCHAR	Schema name for which information is listed.
REFERENCE_TABLE_NAME	VARCHAR	References the TABLE_NAME column in the PRIMARY_KEY table.
REFERENCE_COLUMN_NAME	VARCHAR	References the COLUMN_NAME column in the PRIMARY_KEY table.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

DATABASES

Provides information about the databases in this HP Vertica installation.

Column Name	Data Type	Description
DATABASE_ID	INTEGER	The database's internal ID number
DATABASE_NAME	VARCHAR	The database's name
OWNER_ID	INTEGER	The database owner's ID
OWNER_NAME	INTEGER	The database owner's name
START_TIME	TIMESTAMP TZ	The date and time the database last started
COMPLIANCE_MESSAGE	VARCHAR	Message describing the current state of the database's license compliance.
EXPORT_SUBNET	VARCHAR	The subnet (on the public network) used by the database for import/export.

DUAL

DUAL is a single-column "dummy" table with one record whose value is X; for example:

```
=> SELECT * FROM DUAL;
 dummy
-------
 X
(1 row)
```

You can now write the following types of queries:

```
=> SELECT 1 FROM dual;
 ?column?
----------
        1
(1 row)
=> SELECT current_timestamp, current_user FROM dual;
           ?column?            | current_user
-------------------------------+--------------
 2010-03-08 12:57:32.065841-05 | release
(1 row)


=> CREATE TABLE t1(col1 VARCHAR(20), col2 VARCHAR(2));


=> INSERT INTO T1(SELECT 'hello' AS col1, 1 AS col2 FROM dual);)


=> SELECT * FROM t1;
 col1  | col2
-------+------
 hello | 1
(1 row
```

Because DUAL is a system table, you cannot create projections for it. You also cannot use it in pre-join projections for table objects. For example, assuming the following table schema (`CREATE TABLE t1 (col1 varchar(20), col2 varchar(2));`) both of the following statements are not permitted and will return errors:

```
=> CREATE PROJECTION t1_prejoin AS SELECT * FROM t1 JOIN dual
      ON t1.col1 = dual.dummy;
   ERROR:  Virtual tables are not allowed in FROM clause of projection


=> CREATE PROJECTION dual_proj AS SELECT * FROM dual;
   ERROR:  Virtual tables are not allowed in FROM clause of projection
```

ELASTIC_CLUSTER

Returns information about cluster elasticity, such as whether Elastic Cluster is running.

Column Name	Data Type	Description
SCALING_FACTOR	INTEGER	This value is only meaningful when you enable local segments. SCALING_FACTOR influences

		the number of local segments on each node. Initially—before a rebalance runs—there are *scaling_factor* number of local segments per node. A large SCALING_FACTOR is good for rebalancing a potentially wide range of cluster configurations quickly. However, too large a value could lead to ROS pushback, particularly in a database with a table with a large number of partitions. See ***SET_SCALING_FACTOR*** (page 533) for more details.
MAXIMUM_SKEW_PERCENT	INTEGER	This value is only meaningful when you enable local segments. MAXIMUM_SKEW_PERCENT is the maximum amount of skew a rebalance operation tolerates, which preferentially redistributes local segments; however, if after doing so the segment ranges of any two nodes differs by more than this amount, rebalance will separate and distribute storage to even the distribution.
SEGMENT_LAYOUT	VARCHAR	Current, offset=0, segment layout. New segmented projections will be created with this layout, with segments rotated by the corresponding offset. Existing segmented projections will be rebalanced into an offset of this layout.
LOCAL_SEGMENT_LAYOUT	VARCHAR	Similar to SEGMENT_LAYOUT but includes details that indicate the number of local segments, their relative size and node assignment.
VERSION	INTEGER	Number that gets incremented each time the cluster topology changes (nodes added, marked ephemeral, marked permanent, etc). Useful for monitoring active and past rebalance operations.
IS_ENABLED	BOOLEAN	True if Elastic Cluster is enabled, otherwise false.
IS_LOCAL_SEGMENT_ENABLED	BOOLEAN	True if local segments are enabled, otherwise false.
IS_REBALANCE_RUNNING	BOOLEAN	True if rebalance is currently running, otherwise false.

Permissions

Must be a superuser.

Example

```
=> SELECT * FROM elastic_cluster;

-[ RECORD 1 ]-----------+-------------------------------------------------
scaling_factor          | 4
maximum_skew_percent    | 15
```

```
segment_layout           | v_myvdb_node0004[33.3%] v_myvdb_node0005[33.3%]
                         | v_myvdb_node0006[33.3%]
local_segment_layout     | v_myvdb_node0004[8.3%] v_myvdb_node0004[8.3%]
                         | v_myvdb_node0004[8.3%] v_myvdb_node0004[8.3%]
                         | v_myvdb_node0005[8.3%] v_myvdb_node0005[8.3%]
                         | v_myvdb_node0005[8.3%] v_myvdb_node0005[8.3%]
                         | v_myvdb_node0006[8.3%] v_myvdb_node0006[8.3%]
                         | v_myvdb_node0006[8.3%] v_myvdb_node0006[8.3%]
version                  | 1
is_enabled               | t
is_local_segment_enabled | f
is_rebalance_running     | f
```

See Also

ENABLE_ELASTIC_CLUSTER (page 482)

DISABLE_ELASTIC_CLUSTER (page 468)

Elastic Cluster in the Administrator's Guide

EPOCHS

For all epochs, provides the date and time of the close and the corresponding epoch number of the closed epoch. This information lets you determine which time periods pertain to which epochs.

Column Name	Data Type	Description
EPOCH_CLOSE_TIME	DATETIME	The date and time of the close of the epoch.
EPOCH_NUMBER	INTEGER	The corresponding epoch number of the closed epoch.

Example

```
=> SELECT * FROM epochs;
       epoch_close_time        | epoch_number
-------------------------------+--------------
 2012-11-14 09:26:10.696296-05 |            0
 2012-11-14 10:04:55.668457-05 |            1
 2012-11-14 10:08:05.963606-05 |            2
(3 rows)
```

See Also

Epoch Management Parameters

Epoch Management Functions (page 574)

FOREIGN_KEYS

Provides foreign key information.

Column Name	Data Type	Description
CONSTRAINT_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the constraint.
CONSTRAINT_NAME	VARCHAR	The constraint name for which information is listed.
COLUMN_NAME	VARCHAR	The name of the column that is constrained.
ORDINAL_POSITION	VARCHAR	The position of the column respective to other columns in the table.
TABLE_NAME	VARCHAR	The table name for which information is listed.
REFERENCE_TABLE_NAME	VARCHAR	References the TABLE_NAME column in the PRIMARY_KEY table.
CONSTRAINT_TYPE	VARCHAR	The constraint type, `f`, for foreign key.
REFERENCE_COLUMN_NAME	VARCHAR	References the COLUMN_NAME column in the PRIMARY_KEY table.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
REFERENCE_TABLE_SCHEMA	VARCHAR	References the TABLE_SCHEMA column in the PRIMARY_KEY table.

Example

```
mydb=> SELECT constraint_name, table_name, ordinal_position, reference_table_name
       FROM foreign_keys ORDER BY 3;
      constraint_name      |    table_name     | ordinal_position | reference_table_name
---------------------------+-------------------+------------------+----------------------
 fk_store_sales_date       | store_sales_fact  |                1 | date_dimension
 fk_online_sales_saledate  | online_sales_fact |                1 | date_dimension
 fk_store_orders_product   | store_orders_fact |                1 | product_dimension
 fk_inventory_date         | inventory_fact    |                1 | date_dimension
 fk_inventory_product      | inventory_fact    |                2 | product_dimension
 fk_store_sales_product    | store_sales_fact  |                2 | product_dimension
 fk_online_sales_shipdate  | online_sales_fact |                2 | date_dimension
 fk_store_orders_product   | store_orders_fact |                2 | product_dimension
 fk_inventory_product      | inventory_fact    |                3 | product_dimension
 fk_store_sales_product    | store_sales_fact  |                3 | product_dimension
 fk_online_sales_product   | online_sales_fact |                3 | product_dimension
 fk_store_orders_store     | store_orders_fact |                3 | store_dimension
 fk_online_sales_product   | online_sales_fact |                4 | product_dimension
 fk_inventory_warehouse    | inventory_fact    |                4 | warehouse_dimension
 fk_store_orders_vendor    | store_orders_fact |                4 | vendor_dimension
 fk_store_sales_store      | store_sales_fact  |                4 | store_dimension
 fk_store_orders_employee  | store_orders_fact |                5 | employee_dimension
 fk_store_sales_promotion  | store_sales_fact  |                5 | promotion_dimension
 fk_online_sales_customer  | online_sales_fact |                5 | customer_dimension
 fk_store_sales_customer   | store_sales_fact  |                6 | customer_dimension
 fk_online_sales_cc        | online_sales_fact |                6 | call_center_dimension
 fk_store_sales_employee   | store_sales_fact  |                7 | employee_dimension
 fk_online_sales_op        | online_sales_fact |                7 | online_page_dimension
 fk_online_sales_shipping  | online_sales_fact |                8 | shipping_dimension
 fk_online_sales_warehouse | online_sales_fact |                9 | warehouse_dimension
 fk_online_sales_promotion | online_sales_fact |               10 | promotion_dimension
(26 rows)
```

GRANTS

Provides information about privileges granted on various objects, the granting user, and grantee user. The order of columns in the table corresponds to the order in which they appear in the GRANT command. The GRANTS table does not retain the role grantor.

Column Name	Data Type	Description
GRANT_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the grant.
GRANTOR_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the user who performed the grant operation.
GRANTOR	VARCHAR	The user granting the permission.
PRIVILEGES_DESCRIPTION	VARCHAR	A readable description of the privileges being granted; for example INSERT, SELECT. An asterisk in PRIVILEGES_DESCRIPTION output indicates a privilege WITH GRANT OPTION.
OBJECT_SCHEMA	VARCHAR	The name of the schema that is being granted privileges.
OBJECT_NAME	VARCHAR	The name of the object that is being granted privileges. Note that for schema privileges, the schemaname appears in the OBJECT_NAME column instead of the OBJECT_SCHEMA column.
OBJECT_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the object granted.
OBJECT_TYPE	VARCHAR	The object type on which the grant was applied; for example, ROLE, SCHEMA, DATABASE, RESOURCEPOOL. Output from this column is useful in cases where a schema, resource pool, or user share the same name.
GRANTEE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the user granted permissions.
GRANTEE	VARCHAR	The user being granted permission.

Notes

The vsql commands \dp and \z both include the schema name in the output; for example:

```
=> \dp

         Access privileges for database "vmartdb"
```

```
 Grantee | Grantor | Privileges | Schema |      Name
---------+---------+------------+--------+-----------------
         | dbadmin | USAGE      |        | public
         | dbadmin | USAGE      |        | v_internal
         | dbadmin | USAGE      |        | v_catalog
         | dbadmin | USAGE      |        | v_monitor
         | dbadmin | USAGE      |        | v_internal
         | dbadmin | USAGE      |        | v_catalog
         | dbadmin | USAGE      |        | v_monitor
         | dbadmin | USAGE      |        | v_internal
         | dbadmin | USAGE      |        | designer_system
(9 rows)
```

The vsql command `\dp *.tablename` displays table names in all schemas. This command lets you distinguish grants for same-named tables in different schemas:

```
=> \dp *.events;
                    Access privileges for database "dbadmin"
 Grantee | Grantor |                Privileges                | Schema  | Name
---------+---------+------------------------------------------+---------+-------
 user2   | dbadmin | INSERT, SELECT, UPDATE, DELETE, REFERENCES | schema1 | events
 user1   | dbadmin | SELECT                                     | schema1 | events
 user2   | dbadmin | INSERT, SELECT, UPDATE, DELETE, REFERENCES | schema2 | events
 user1   | dbadmin | INSERT, SELECT                             | schema2 | events
(4 rows)
```

The vsql command `\dp schemaname.*` displays all tables in the named schema:

```
=> \dp schema1.*
                    Access privileges for database "dbadmin"
 Grantee | Grantor |           Privileges                     | Schema       | Name
---------+---------+------------------------------------------+--------------+------------
 user2   | dbadmin | INSERT, SELECT, UPDATE, DELETE, REFERENCES | schema1      | events
 user1   | dbadmin | SELECT                                     | schema1      | events
(2 rows)
```

Examples

This example shows CREATE and USAGE privileges granted to Bob in the fictitious apps database:

```
=> SELECT grantor, privileges_description, object_schema, object_name, grantee
    FROM grants;

 grantor | privileges_description | object_schema | object_name | grantee
---------+------------------------+---------------+-------------+---------
 dbadmin | USAGE                  |               | general     | Bob
 dbadmin | CREATE                 |               | schema2     | Bob
(2 rows)
```

This next query looks for privileges granted to a particular set of grantees. The asterisk in privileges_description column for User1 means that user has WITH GRANT OPTION privileges.

```
=> SELECT grantor, privileges_description, object_schema, object_name, grantee
    FROM grants WHERE grantee ILIKE 'User%';

 grantor |            privileges_description             | object_schema | object_name | grantee
---------+-----------------------------------------------+---------------+-------------+-------
--
```

```
 release | USAGE                                          |              | general     | User1
 release | USAGE                                          |              | general     | User2
 release | USAGE                                          |              | general     | User3
 release | USAGE                                          |              | s1          | User1
 release | USAGE                                          |              | s1          | User2
 release | USAGE                                          |              | s1          | User3
 User1   | INSERT*, SELECT*, UPDATE*, DELETE*, REFERENCES* | s1           | t1          | User1
(7 rows)
```

In the following example, online_sales is the schema that first gets privileges, and then inside that schema the anchor table gets SELECT privileges:

```
=> SELECT grantee, grantor, privileges_description, object_schema, object_name
   FROM grants WHERE grantee='u1' ORDER BY object_name;
```

```
 grantee | grantor | privileges_description | object_schema | object_name
---------+---------+------------------------+-------- ------+-----------------
 u1      | dbadmin | CREATE                 |               | online_sales
 u1      | dbadmin | SELECT                 | online_sales  | online_sales_fact
```

The following statement shows all grants for user Bob:

```
-[ RECORD 1 ]----------+--------------------
grant_id               | 45035996273749244
grantor_id             | 45035996273704962
grantor                | dbadmin
privileges_description | USAGE
object_schema          |
object_name            | general
object_id              | 45035996273718666
object_type            | RESOURCEPOOL
grantee_id             | 45035996273749242
grantee                | Bob
-[ RECORD 2 ]----------+--------------------
grant_id               | 45035996273749598
grantor_id             | 45035996273704962
grantor                | dbadmin
privileges_description |
object_schema          |
object_name            | dbadmin
object_id              | 45035996273704968
object_type            | ROLE
grantee_id             | 45035996273749242
grantee                | Bob
-[ RECORD 3 ]----------+--------------------
grant_id               | 45035996273749716
grantor_id             | 45035996273704962
grantor                | dbadmin
privileges_description |
object_schema          |
object_name            | dbadmin
object_id              | 45035996273704968
object_type            | ROLE
grantee_id             | 45035996273749242
grantee                | Bob
-[ RECORD 4 ]----------+--------------------
grant_id               | 45035996273755986
grantor_id             | 45035996273704962
```

```
grantor                | dbadmin
privileges_description |
object_schema          |
object_name            | pseudosuperuser
object_id              | 45035996273704970
object_type            | ROLE
grantee_id             | 45035996273749242
grantee                | Bob
-[ RECORD 5 ]----------+--------------------
grant_id               | 45035996273756986
grantor_id             | 45035996273704962
grantor                | dbadmin
privileges_description | CREATE, CREATE TEMP
object_schema          |
object_name            | mcdb
object_id              | 45035996273704974
object_type            | DATABASE
grantee_id             | 45035996273749242
grantee                | Bob
-[ RECORD 5 ]----------+--------------------
...
```

See also

HAS_ROLE (page 501)

ROLES (page 967)

USERS (page 985)

Managing Users and Privileges in the Administrator's Guide

LICENSE_AUDITS

Lists the results of HP Vertica's license automatic compliance audits. See How HP Vertica Calculates Database Size in the Administrator's Guide.

Column Name	Data Type	Description
DATABASE_SIZE_BYTES	INTEGER	The estimated raw data size of the database
LICENSE_SIZE_BYTES	INTEGER	The licensed data allowance
USAGE_PERCENT	FLOAT	Percentage of the licensed allowance used
AUDIT_START_TIMESTAMP	TIMESTAMPTZ	When the audit started
AUDIT_END_TIMESTAMP	TIMESTAMPTZ	When the audit finished
CONFIDENCE_LEVEL_PERCENT	FLOAT	The confidence level of the size estimate
ERROR_TOLERANCE_PERCENT	FLOAT	The error tolerance used for the size estimate

USED_SAMPLING	BOOLEAN	Whether data was randomly sampled (if false, all of the data was analyzed)
CONFIDENCE_INTERVAL_LOWER_BOUND_BYTES	INTEGER	The lower bound of the data size estimate within the confidence level
CONFIDENCE_INTERVAL_UPPER_BOUND_BYTES	INTEGER	The upper bound of the data size estimate within the confidence level
SAMPLE_COUNT	INTEGER	The number of data samples used to generate the estimate
CELL_COUNT	INTEGER	The number of cells in the database

NODES

Lists details about the nodes in the database.

Column Name	Date Type	Description
NODE_NAME	VARCHAR(128)	The name of the node.
NODE_ID	INT	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the node.
NODE_STATE	VARCHAR(128)	The node's current state (up, down, recovering, etc.).
NODE_ADDRESS	VARCHAR(80)	The host address of the node.
EXPORT_ADDRESS	VARCHAR	The IP address of the node (on the public network) used for import/export operations.
CATALOG_PATH	VARCHAR(8192)	The absolute path to the catalog on the node.
IS_EPHEMERAL	BOOLEAN	True if this node has been marked as ephemeral (in preparation of removing it from the cluster).

Example

```
dbadmin=> \x
Expanded display is on.
-[ RECORD 1 ]--+---------------------------------------------------
node_name      | v_mcdb_node0001
node_id        | 45035996273704980
node_state     | UP
node_address   | XX.XX.XXX.X2
export_address | XX.XX.XXX.X2
catalog_path   | /home/dbadmin/mcdb/v_vmart_node0001_catalog/Catalog
is_ephemeral   | f
-[ RECORD 2 ]--+---------------------------------------------------
node_name      | v_mcdb_node0002
```

```
node_id         | 45035996273718764
node_state      | UP
node_address    | XX.XX.XXX.X3
export_address | XX.XX.XXX.X3
catalog_path    | /home/dbadmin/mcdb/v_vmart_node0002_catalog/Catalog
is_ephemeral    | f
-[ RECORD 3 ]--+---------------------------------------------------
node_name       | v_mcdb_node0003
node_id         | 45035996273718768
node_state      | UP
node_address    | XX.XX.XXX.X4
export_address | XX.XX.XXX.X4
catalog_path    | /home/dbadmin/mcdb/v_vmart_node0003_catalog/Catalog
is_ephemeral    | f
```

ODBC_COLUMNS

Provides table column information. The format is defined by the ODBC standard for the ODBC SQLColumns metadata. Details on the ODBC SQLColumns format are available in the ODBC specification: ***http://msdn.microsoft.com/en-us/library/windows/desktop/ms711683%28v=vs.85%29.aspx http://www.***.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	The name of the schema in which the column resides. If the column does not reside in a schema, this field is empty.
TABLE_NAME	VARCHAR	The name of the table in which the column resides.
COLUMN_NAME	VARCHAR	The name of the column.
DATA_TYPE	INTEGER	The data type of the column. This can be an ODBC SQL data type or a driver-specific SQL data type. This column corresponds to the ODBC_TYPE column in the ***TYPES*** (page 980) table.
DATA_TYPE_NAME	VARCHAR	The driver-specific data type name.
COLUMN_SIZE	INTEGER	The ODBC-defined data size of the column.
BUFFER_LENGTH	INTEGER	The transfer octet length of a column is the maximum number of bytes returned to the application when data is transferred to its default C data type. See ***http://msdn.microsoft.com/en-us/library/windows/desktop/ms713979%28v=vs.85%29.aspx http://msdn.microsoft.com/en-us/library/windows/desktop/ms713979%28v=vs.85%29.aspx***
DECIMAL_DIGITS	INTEGER	The total number of significant digits to the right of the decimal point. This value has no meaning for

		non-decimal data types.
NUM_PREC_RADIX	INTEGER	The radix HP Vertica reports decimal_digits and columns_size as. This value is always 10, because it refers to a number of decimal digits, rather than a number of bits.
NULLABLE	BOOLEAN	Indicates whether the column can contain null values. Values are 0 or 1.
REMARKS	VARCHAR	The textual remarks for the column.
COLUMN_DEFAULT	VARCHAR	The default value of the column.
SQL_TYPE_ID	INTEGER	The SQL data type of the column.
SQL_DATETIME_SUB	VARCHAR	The subtype for a datetime data type. This value has no meaning for non-datetime data types.
CHAR_OCTET_LENGTH	INTEGER	The maximum length of a string or binary data column.
ORDINAL_POSITION	INTEGER	Indicates the position of the column in the table definition.
IS_NULLABLE	VARCHAR	Values can be YES or NO, determined by the value of the NULLABLE column.
IS_IDENTITY	BOOLEAN	Indicates whether the column is a sequence, for example, an auto increment column.

Example

```
dbadmin=> select schema_name, table_name, data_type_name from odbc_columns limit
2;

 schema_name |         table_name          | data_type_name
-------------+-----------------------------+----------------
 v_monitor   | execution_engine_profiles | Boolean
 v_monitor   | query_profiles            | Boolean
(2 rows)
```

PASSWORDS

Contains user passwords information. This table stores not only current passwords, but also past passwords if any profiles have PASSWORD_REUSE_TIME or PASSWORD_REUSE_MAX parameters set. See ***CREATE PROFILE*** (page 739) for details.

Column Name	Data Type	Description
USER_ID	INTEGER	The ID of the user who owns the password.
USER_NAME	VARCHAR	The name of the user who owns the password.
PASSWORD	VARCHAR	The encrypted password.

PASSWORD_CREATE_TIME	DATETIME	The date and time when the password was created.
IS_CURRENT_PASSWORD	BOOLEAN	Denotes whether this is the user's current password. Non-current passwords are retained to enforce password reuse limitations.
PROFILE_ID	INTEGER	The ID number of the profile to which the user is assigned.
PROFILE_NAME	VARCHAR	The name of the profile to which the user is assigned.
PASSWORD_REUSE_MAX	VARCHAR	The number password changes that must take place before an old password can be reused.
PASSWORD_REUSE_TIME	VARCHAR	The amount of time that must pass before an old password can be reused.

PRIMARY_KEYS

Provides primary key information.

Column Name	Data Type	Description
CONSTRAINT_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the constraint.
CONSTRAINT_NAME	VARCHAR	The constraint name for which information is listed.
COLUMN_NAME	VARCHAR	The column name for which information is listed.
ORDINAL_POSITION	VARCHAR	The position of the column respective to other columns in the table.
TABLE_NAME	VARCHAR	The table name for which information is listed.
CONSTRAINT_TYPE	VARCHAR	The constraint type, p, for primary key.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.

Example

Request specific columns from the PRIMARY_KEYS table:

```
=> SELECT constraint_name, table_name, ordinal_position, table_schema
   FROM primary_keys ORDER BY 3;

 constraint_name |      table_name       | ordinal_position | table_schema
-----------------+-----------------------+------------------+--------------
 C_PRIMARY       | customer_dimension    |                1 | public
 C_PRIMARY       | product_dimension     |                1 | public
 C_PRIMARY       | store_dimension       |                1 | store
 C_PRIMARY       | promotion_dimension   |                1 | public
 C_PRIMARY       | date_dimension        |                1 | public
 C_PRIMARY       | vendor_dimension      |                1 | public
 C_PRIMARY       | employee_dimension    |                1 | public
```

```
 C_PRIMARY       | shipping_dimension    |                1 | public
 C_PRIMARY       | warehouse_dimension   |                1 | public
 C_PRIMARY       | online_page_dimension |                1 | online_sales
 C_PRIMARY       | call_center_dimension |                1 | online_sales
 C_PRIMARY       | product_dimension     |                2 | public
(12 rows)
```

PROFILE_PARAMETERS

Defines what information is stored in profiles.

Column Name	Data Type	Description
PROFILE_ID	INTEGER	The ID of the profile to which this parameter belongs.
PROFILE_NAME	VARCHAR	The name of the profile to which this parameter belongs.
PARAMETER_TYPE	VARCHAR	The policy type of this parameter (password_complexity, password_security, etc.)
PARAMETER_NAME	VARCHAR	The name of the parameter.
PARAMETER_LIMIT	VARCHAR	The parameter's value.

PROFILES

Provides information about password policies that you set using the ***CREATE PROFILE*** (page 739) statement.

Column Name	Data Type	Description
PROFILE_ID	INTEGER	The unique identifier for the profile.
PROFILE_NAME	VARCHAR	The profile's name.
PASSWORD_LIFE_TIME	VARCHAR	The number of days before the user's password expires. After expiration, the user is forced to change passwords during login or warned that their password has expired if password_grace_time is set to a value other than zero or unlimited.
PASSWORD_GRACE_TIME	VARCHAR	The number of days users are allowed to log in after their passwords expire. During the grace time, users are warned about their expired passwords when they log in. After the grace period, the user is forced to change passwords if he or she hasn't already.
PASSWORD_REUSE_MAX	VARCHAR	The number of password changes that must occur before the current password can be reused.

PASSWORD_REUSE_TIME	VARCHAR	The number of days that must pass after setting a password before it can be used again.
FAILED_LOGIN_ATTEMPTS	VARCHAR	The number of consecutive failed login attempts that triggers HP Vertica to lock the account.
PASSWORD_LOCK_TIME	VARCHAR	The number of days an account is locked after being locked due to too many failed login attempts.
PASSWORD_MAX_LENGTH	VARCHAR	The maximum number of characters allowed in a password.
PASSWORD_MIN_LENGTH	VARCHAR	The minimum number of characters required in a password.
PASSWORD_MIN_LETTERS	VARCHAR	The minimum number of letters (either uppercase or lowercase) required in a password.
PASSWORD_MIN_LOWERCASE_LETTERS	VARCHAR	The minimum number of lowercase.
PASSWORD_MIN_UPPERCASE_LETTERS	VARCHAR	The minimum number of uppercase letters required in a password.
PASSWORD_MIN_DIGITS	VARCHAR	The minimum number of digits required in a password.
PASSWORD_MIN_SYMBOLS	VARCHAR	The minimum of symbols (for example, !, #, $, etc.) required in a password.

Notes

Non-superusers querying this table see only the information for the profile to which they are assigned.

See Also

CREATE PROFILE (page 739)

ALTER PROFILE (page 660)

Profiles in the Administrator's Guide

PROJECTION_CHECKPOINT_EPOCHS

Records when each projection checkpoint epoch changes.

Column Name	Data Type	Description
NODE_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog,

		which identifies the node for which information is listed
NODE_NAME	VARCHAR	Node name for which information is listed.
PROJECTION_SCHEMA_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the specific schema that contains the projection.
PROJECTION_SCHEMA	VARCHAR	Schema containing the projection.
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed.
IS_UP_TO_DATE	BOOLEAN	Indicates whether the projection is up to date, where *t* is true and *f* is false. Projections must be up to date for queries to use them.
CHECKPOINT_EPOCH	INTEGER	Checkpoint epoch of the projection on the corresponding node. Data up to and including this epoch is in persistent storage. If the node were to fail, without moving more data out of the WOS, data after this epoch would need to be recovered.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> SELECT epoch FROM t;
 epoch
-------
    52
    52
    53
(3 rows)

=> SELECT * FROM projection_checkpoint_epochs;

      node_id      | node_name | projection_schema_id | projection_schema |   projection_id   |
projection_name | is_up_to_date | checkpoint_epoch
-------------------+-----------+----------------------+-------------------+-------------------+--
---------------+---------------+------------------
 45035996273704970 | node01    | 45035996273704966    | public            | 45035996273724430 |
t_super         | t             | 51
 45035996273704970 | node01    | 45035996273704966    | public            | 45035996273724452 |
p_super         | t             | 51
(2 rows)


=> SELECT DO_TM_TASK('moveout','');

                                                  do_tm_task
-----------------------------------------------------------------------------------------------
-------------
 Task: moveout
(Table: public.t) (Projection: public.t_super)
(Table: public.p) (Projection: public.p_super)
```

```
(1 row)

=> SELECT * FROM projection_checkpoint_epochs;

      node_id      | node_name | projection_schema_id | projection_schema |   projection_id   |
projection_name | is_up_to_date | checkpoint_epoch
-------------------+-----------+----------------------+-------------------+-------------------+--
---------------+---------------+------------------
 45035996273704970 | node01    | 45035996273704966    | public            | 45035996273724430 |
t_super         | t             | 53
 45035996273704970 | node01    | 45035996273704966    | public            | 45035996273724452 |
p_super         | t             | 53
(2 rows)
```

PROJECTION_COLUMNS

Provides information about projection columns, such as encoding type, sort order, type of statistics, and the time at which columns statistics were last updated.

Column Name	Data Type	Description
PROJECTION_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed.
PROJECTION_COLUMN_NAME	VARCHAR	The projection column name.
COLUMN_POSITION	INTEGER	The ordinal position of a projection's column used in the CREATE PROJECTION (page 742) statement.
SORT_POSITION	INTEGER	The projection's column sort specification, as specified in CREATE PROJECTION .. ORDER BY clause. SORT_POSITION output is NULL if the column is not included in the projection's sort order.
COLUMN_ID	INTEGER	A unique numeric object ID (OID) assigned by the HP Vertica catalog, COLUMN_ID is the OID of the associated projection column object. This field is helpful as a key to other system tables.
DATA_TYPE	VARCHAR	Matches the corresponding table column data type (see ***V_CATALOG.COLUMNS*** (page 935)). DATA_TYPE is provided as a complement to ENCODING_TYPE.
ENCODING_TYPE	VARCHAR	The encoding type defined on the projection column.
ACCESS_RANK	INTEGER	The access rank of the projection column. See the ACCESSRANK parameter in the ***CREATE***

		PROJECTION (page 742) statement for more information.
GROUP_ID	INTEGER	A unique numeric ID (OID) assigned by the HP Vertica catalog that identifies the group.
TABLE_SCHEMA	VARCHAR	The schema name in which the projection resides.
TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog that identifies the table.
TABLE_NAME	VARCHAR	The table name that contains the projection.
TABLE_COLUMN_ID	VARCHAR	A unique VARCHAR ID, assigned by the HP Vertica catalog, that identifies a column in a table.
TABLE_COLUMN_NAME	VARCHAR	The projection's corresponding table column name.
STATISTICS_TYPE	VARCHAR	The type of statistics the column has, which can be one of: ▪ NONE—No statistics ▪ ROWCOUNT—Created from existing catalog metadata, which HP Vertica automatically and periodically updates ▪ FULL—Created by running ***ANALYZE_STATISTICS()*** (page 440)
STATISTICS_UPDATED_TIMESTAMP	TIMESTAMPTZ	The time at which the columns statistics were last updated. Querying this column, along with STATISTICS_TYPE and PROJECTION_COLUMN_NAME, lets you quickly identify projection columns whose statistics need updating. See also ***V_CATALOG.PROJECTIONS.HAS_STATISTICS*** (page 961).

Example

On a single-node cluster, the following sample schema defines a table named trades, which groups the highly-correlated columns bid and ask and stores the stock column separately:

```
=> CREATE TABLE trades (stock CHAR(5), bid INT, ask INT);

=> CREATE PROJECTION trades_p (stock ENCODING RLE, GROUPED(bid ENCODING
     DELTAVAL, ask)) AS (SELECT * FROM trades) ORDER BY stock, bid;

=> INSERT INTO trades VALUES('acme', 10, 20);

=> COMMIT;
```

Query the PROJECTION_COLUMNS table for table trades:

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

```
=> SELECT * FROM PROJECTION_COLUMNS WHERE table_name = 'trades';
```

Notice that the `statistics_type column` returns `NONE` for all three columns in the `trades` table. Also, there is no value in the `statistics_updated_timestamp` field because statistics have not yet been run on this table.

```
-[ RECORD 1 ]----------------+--------------------
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | stock
column_position              | 0
sort_position                | 0
column_id                    | 45035996273718840
data_type                    | char(5)
encoding_type                | RLE
access_rank                  | 0
group_id                     | 0
table_schema                 | public
table_id                     | 45035996273718836
table_name                   | trades
table_column_id              | 45035996273718836-1
table_column_name            | stock
statistics_type              | NONE
statistics_updated_timestamp |
-[ RECORD 2 ]----------------+--------------------
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | bid
column_position              | 1
sort_position                | 1
column_id                    | 45035996273718842
data_type                    | int
encoding_type                | DELTAVAL
access_rank                  | 0
group_id                     | 45035996273718844
table_schema                 | public
table_id                     | 45035996273718836
table_name                   | trades
table_column_id              | 45035996273718836-2
table_column_name            | bid
statistics_type              | NONE
statistics_updated_timestamp |
-[ RECORD 3 ]----------------+--------------------
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | ask
column_position              | 2
sort_position                |
column_id                    | 45035996273718846
data_type                    | int
encoding_type                | AUTO
access_rank                  | 0
group_id                     | 45035996273718844
table_schema                 | public
table_id                     | 45035996273718836
```

```
table_name                    | trades
table_column_id               | 45035996273718836-3
table_column_name             | ask
statistics_type               | NONE
statistics_updated_timestamp |
```

Now run statistics on the `stock` column:

```
=> SELECT ANALYZE_STATISTICS('trades.stock');
```

The system returns 0 for success:

```
-[ RECORD 1 ]------+--
ANALYZE_STATISTICS | 0
```

Now query PROJECTION_COLUMNS again:

```
=> SELECT * FROM PROJECTION_COLUMNS where table_name = 'trades';
```

This time, `statistics_type` changes to FULL for the `trades.stock` column (representing full statistics were run), and the `statistics_updated_timestamp` column returns the time the `stock` columns statistics were updated. Note that the timestamp for the `bid` and `ask` columns have not changed because statistics were not run on those columns. Also, the `bid` and `ask` columns changed from NONE to ROWCOUNT. This is because HP Vertica automatically updates ROWCOUNT statistics from time to time. The statistics are created by looking at existing catalog metadata.

```
-[ RECORD 1 ]----------------+-----------------------------
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | stock
column_position              | 0
sort_position                | 0
column_id                    | 45035996273718840
data_type                    | char(5)
encoding_type                | RLE
access_rank                  | 0
group_id                     | 0
table_schema                 | public
table_id                     | 45035996273718836
table_name                   | trades
table_column_id              | 45035996273718836-1
table_column_name            | stock
statistics_type              | FULL
statistics_updated_timestamp | 2012-12-08 13:52:04.178294-05
-[ RECORD 2 ]----------------+-----------------------------
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | bid
column_position              | 1
sort_position                | 1
column_id                    | 45035996273718842
data_type                    | int
encoding_type                | DELTAVAL
access_rank                  | 0
group_id                     | 45035996273718844
table_schema                 | public
```

```
table_id                     | 45035996273718836
table_name                   | trades
table_column_id              | 45035996273718836-2
table_column_name            | bid
statistics_type              | ROWCOUNT
statistics_updated_timestamp | 2012-12-08 13:51:20.016465-05
-[ RECORD 3 ]----------------+-----------------------------
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | ask
column_position              | 2
sort_position                |
column_id                    | 45035996273718846
data_type                    | int
encoding_type                | AUTO
access_rank                  | 0
group_id                     | 45035996273718844
table_schema                 | public
table_id                     | 45035996273718836
table_name                   | trades
table_column_id              | 45035996273718836-3
table_column_name            | ask
statistics_type              | ROWCOUNT
statistics_updated_timestamp | 2012-12-08 13:51:20.016475-05
```

If you run statistics on the bid column and then query this system table again, only RECORD 2 is updated:

```
=> SELECT ANALYZE_STATISTICS('trades.bid');
-[ RECORD 1 ]------+--
ANALYZE_STATISTICS | 0

=> SELECT * FROM PROJECTION_COLUMNS where table_name = 'trades';

-[ RECORD 1 ]----------------+-----------------------------
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | stock
column_position              | 0
sort_position                | 0
column_id                    | 45035996273718840
data_type                    | char(5)
encoding_type                | RLE
access_rank                  | 0
group_id                     | 0
table_schema                 | public
table_id                     | 45035996273718836
table_name                   | trades
table_column_id              | 45035996273718836-1
table_column_name            | stock
statistics_type              | FULL
statistics_updated_timestamp | 2012-12-08 13:52:04.178294-05
-[ RECORD 2 ]----------------+-----------------------------
```

```
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | bid
column_position              | 1
sort_position                | 1
column_id                    | 45035996273718842
data_type                    | int
encoding_type                | DELTAVAL
access_rank                  | 0
group_id                     | 45035996273718844
table_schema                 | public
table_id                     | 45035996273718836
table_name                   | trades
table_column_id              | 45035996273718836-2
table_column_name            | bid
statistics_type              | FULL
statistics_updated_timestamp | 2012-12-08 13:53:23.438447-05
-[ RECORD 3 ]----------------+------------------------------
projection_id                | 45035996273718838
projection_name              | trades_p
projection_column_name       | ask
column_position              | 2
sort_position                |
column_id                    | 45035996273718846
data_type                    | int
encoding_type                | AUTO
access_rank                  | 0
group_id                     | 45035996273718844
table_schema                 | public
table_id                     | 45035996273718836
table_name                   | trades
table_column_id              | 45035996273718836-3
table_column_name            | ask
statistics_type              | ROWCOUNT
statistics_updated_timestamp | 2012-12-08 13:51:20.016475-05
```

You can quickly query just the timestamp column to see when the columns were updated:

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

=> SELECT ANALYZE_STATISTICS('trades');
 ANALYZE_STATISTICS
--------------------
                  0
(1 row)

=> SELECT projection_column_name, statistics_type,
   statistics_updated_timestamp
   FROM PROJECTION_COLUMNS where table_name = 'trades';

 projection_column_name | statistics_type | statistics_updated_timestamp
------------------------+-----------------+------------------------------
```

```
 stock                    | FULL              | 2012-12-08 13:54:27.428622-05
 bid                      | FULL              | 2012-12-08 13:54:27.428632-05
 ask                      | FULL              | 2012-12-08 13:54:27.428639-05
(3 rows)
```

See Also

V_CATALOG.PROJECTIONS (page 961)

ANALYZE_STATISTICS (page 440)

CREATE PROJECTION (page 742)

Collecting Statistics in the Administrator's Guide

PROJECTION_DELETE_CONCERNS

Lists projections whose design may cause performance issues when deleting data. This table is generated by calling the ***EVALUATE_DELETE_PERFORMANCE*** (page 484) function. See Optimizing Deletes and Updates for Performance in the Administrator's Guide for more information.

Column Name	Data Type	Description
PROJECTION_ID	INTEGER	The ID number of the projection
PROJECTION_SCHEMA	VARCHAR	The schema containing the projection
PROJECTION_NAME	VARCHAR	The projection's name
CREATION_TIME	TIMESTAMPTZ	When the projection was created
LAST_MODIFIED_TIME	TIMESTAMPTZ	When the projection was last modified
COMMENT	VARCHAR	A comment describing the potential delete performance issue.

PROJECTIONS

Provides information about projections.

Column Name	Data Type	Description
PROJECTION_SCHEMA_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the specific schema that contains the projection.
PROJECTION_SCHEMA	VARCHAR	The name of the schema that contains the projection.
PROJECTION_ID	INTEGER	A unique numeric ID assigned by the HP Vertica

		catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed.
OWNER_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the projection owner.
OWNER_NAME	VARCHAR	The name of the projection's owner.
ANCHOR_TABLE_ID	INTEGER	The unique numeric identification (OID) of the anchor table for pre-join projections, or the OID of the table from which the projection was created if it is not a pre-join projection. **Note:** A projection has only one anchor (fact) table.
ANCHOR_TABLE_NAME	VARCHAR	The name of the anchor table for pre-join projections, or the name of the table from which the projection was created if it is not a pre-join projection.
NODE_ID	INTEGER	A unique numeric ID (OID) that identifies the node(s) that contain the projection.
NODE_NAME	VARCHAR	The name of the node(s) that contain the projection. **Note:** this column returns information for unsegmented projections only, not for segmented and pinned projections.
IS_PREJOIN	BOOLEAN	Indicates whether the projection is a pre-join projection, where *t* is true and *f* is false.
IS_SUPER_PROJECTION	BOOLEAN	Indicates *t* (true) if a projection is a super-projection or *f* (false) if it is not.
CREATED_EPOCH	INTEGER	The epoch in which the projection was created.
CREATE_TYPE	VARCHAR	The method in which the projection was created: ▪ CREATE PROJECTION—A custom projection created using a CREATE PROJECTION statement. ▪ CREATE TABLE—A superprojection that was automatically created when its associated table was created using a CREATE TABLE statement. ▪ CREATE TABLE WITH PROJ CLAUSE—A superprojection created using a CREATE TABLE statement. ▪ DELAYED_CREATION—A superprojection that was automatically created when data was loaded into its associated table. ▪ DESIGNER—A new projection created by the Database Designer.

		▪ SYSTEM TABLE—A projection that was automatically created for a system table. Rebalancing does not change the CREATE_TYPE value for a projection.
VERIFIED_FAULT_TOLERANCE	INTEGER	The projection K-safe value. This value can be greater than the database K-safety value (if more replications of a projection exist than are required to meet the database K-safety). This value cannot be less than the database K-safe setting.
IS_UP_TO_DATE	BOOLEAN	Indicates whether the projection is up to date, where *t* is true and *f* is false. Projections must be up to date to be used in queries.
HAS_STATISTICS	BOOLEAN	Indicates whether there are statistics for any column in the projection, where *t* is true and *f* is false. **Notes:** ▪ This column returns true only when all non-epoch columns for a table have full statistics. Otherwise the column returns false. See ***ANALYZE_STATISTICS()*** (page 440). ▪ Projections that have no data never have full statistics. Use the ***PROJECTION_STORAGE*** (page 1059) system table to see if your projection contains data.

Example

The following example queries the PROJECTION_COLUMNS table to see if the projections are up to date and are pre-join projections.

```
=> SELECT projection_name, anchor_table_name, is_prejoin, is_up_to_date
   FROM projections;

       projection_name        |   anchor_table_name   | is_prejoin | is_up_to_date
------------------------------+-----------------------+------------+---------------
 customer_dimension_site01    | customer_dimension    | f          | t
 customer_dimension_site02    | customer_dimension    | f          | t
 customer_dimension_site03    | customer_dimension    | f          | t
 customer_dimension_site04    | customer_dimension    | f          | t
 product_dimension_site01     | product_dimension     | f          | t
 product_dimension_site02     | product_dimension     | f          | t
 product_dimension_site03     | product_dimension     | f          | t
 product_dimension_site04     | product_dimension     | f          | t
 store_sales_fact_p1          | store_sales_fact      | t          | t
 store_sales_fact_p1_b1       | store_sales_fact      | t          | t
 store_orders_fact_p1         | store_orders_fact     | t          | t
 store_orders_fact_p1_b1      | store_orders_fact     | t          | t
 online_sales_fact_p1         | online_sales_fact     | t          | t
 online_sales_fact_p1_b1      | online_sales_fact     | t          | t
```

```
 promotion_dimension_site01   | promotion_dimension   | f          | t
 promotion_dimension_site02   | promotion_dimension   | f          | t
 promotion_dimension_site03   | promotion_dimension   | f          | t
 promotion_dimension_site04   | promotion_dimension   | f          | t
 date_dimension_site01        | date_dimension        | f          | t
 date_dimension_site02        | date_dimension        | f          | t
 date_dimension_site03        | date_dimension        | f          | t
 date_dimension_site04        | date_dimension        | f          | t
 vendor_dimension_site01      | vendor_dimension      | f          | t
 vendor_dimension_site02      | vendor_dimension      | f          | t
 vendor_dimension_site03      | vendor_dimension      | f          | t
 vendor_dimension_site04      | vendor_dimension      | f          | t
 employee_dimension_site01    | employee_dimension    | f          | t
 employee_dimension_site02    | employee_dimension    | f          | t
 employee_dimension_site03    | employee_dimension    | f          | t
 employee_dimension_site04    | employee_dimension    | f          | t
 shipping_dimension_site01    | shipping_dimension    | f          | t
 shipping_dimension_site02    | shipping_dimension    | f          | t
 shipping_dimension_site03    | shipping_dimension    | f          | t
 shipping_dimension_site04    | shipping_dimension    | f          | t
 warehouse_dimension_site01   | warehouse_dimension   | f          | t
 warehouse_dimension_site02   | warehouse_dimension   | f          | t
 warehouse_dimension_site03   | warehouse_dimension   | f          | t
 warehouse_dimension_site04   | warehouse_dimension   | f          | t
 inventory_fact_p1            | inventory_fact        | f          | t
 inventory_fact_p1_b1         | inventory_fact        | f          | t
 store_dimension_site01       | store_dimension       | f          | t
 store_dimension_site02       | store_dimension       | f          | t
 store_dimension_site03       | store_dimension       | f          | t
 store_dimension_site04       | store_dimension       | f          | t
 online_page_dimension_site01 | online_page_dimension | f          | t
 online_page_dimension_site02 | online_page_dimension | f          | t
 online_page_dimension_site03 | online_page_dimension | f          | t
 online_page_dimension_site04 | online_page_dimension | f          | t
 call_center_dimension_site01 | call_center_dimension | f          | t
 call_center_dimension_site02 | call_center_dimension | f          | t
 call_center_dimension_site03 | call_center_dimension | f          | t
 call_center_dimension_site04 | call_center_dimension | f          | t
(52 rows)
```

See Also

ANALYZE_STATISTICS() (page 440)

PROJECTION_COLUMNS (page 955)

PROJECTION_STORAGE (page 1059)

RESOURCE_POOL_DEFAULTS

Provides information about the default values for resource pools. Information is for both HP Vertica-internal and DBA-created pools.

For any resource pool, you can restore default values contained in this table by simply using the DEFAULT keyword in the `ALTER_RESOURCE_POOL` statement.

To see default values for resource pools:

```
VMart= > SELECT * FROM V_CATALOG.RESOURCE_POOL_DEFAULTS;
```

Permissions

No explicit permissions are required.

See Also

Built-in Pool Configuration (page 759)

RESOURCE_POOLS (page 965)

CREATE RESOURCE POOL (page 753)

ALTER RESOURCE POOL (page 663)

SET SESSION RESOURCE_POOL (page 916)

DROP RESOURCE POOL (page 819)

Managing Workloads and Guidelines for Setting Pool Parameters in the Administrator's Guide

RESOURCE_POOLS

Displays information about the parameters specified for the resource pool in the ***CREATE RESOURCE POOL*** (page 753) statement.

Column Name	Data Type	Description
NAME	VARCHAR	The name of the resource pool.
IS_INTERNAL	BOOLEAN	Denotes whether a pool is one of the ***built-in pools*** (page 757).
MEMORYSIZE	VARCHAR	Value of the amount of memory allocated to the resource pool.
MAXMEMORYSIZE	VARCHAR	Value assigned as the maximum size the resource pool could grow by borrowing memory from the GENERAL pool.
EXECUTIONPARALLELISM	INTEGER	[Default: AUTO] Limits the number of threads used to process any single query issued in this resource pool. When set to AUTO, HP Vertica sets this value based on the number of cores, available memory, and amount of data in the system. Unless data is limited, or the amount of data is very small, HP Vertica sets this value to the number of cores on the node. Reducing this value increases the throughput of short queries issued in the pool, especially if the queries are executed concurrently.

		If you choose to set this parameter manually, set it to a value between 1 and the number of cores.
PRIORITY	INTEGER	Value of PRIORITY parameter specified when defining the pool.
RUNTIMEPRIORITY	VARCHAR	Value that indicates the amount of run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to running queries in the resource pool. Valid values are: ▪ HIGH ▪ MEDIUM (Default) ▪ LOW These values are relative to each other. Queries with a HIGH run-time priority are given more CPU and I/O resources than those with a MEDIUM or LOW run-time priority.
RUNTIMEPRIORITYTHRESHOLD	INTEGER	Value that specifies the time limit (in seconds) by which a query must finish before the Resource Manager assigns to it the RUNTIMEPRIORITY of the resource pool. All queries begin running at a HIGH priority. When a query's duration exceeds this threshold, it is assigned the RUNTIMEPRIORITY of the resource pool. Default is 2.
QUEUETIMEOUT	INTEGER	Value in seconds of QUEUETIMEOUT parameter specified when defining the pool. Represents the maximum amount of time the request is allowed to wait for resources to become available before being rejected.
PLANNEDCONCURRENCY	INTEGER	Value of PLANNEDCONCURRENCY parameter specified when defining the pool, which represents the number of concurrent queries that are normally expected to be running against the resource pool.
MAXCONCURRENCY	INTEGER	Value of MAXCONCURRENCY parameter specified when defining the pool, which represents the maximum number of concurrent execution slots available to the resource pool.
RUNTIMECAP	INTERVAL	[Default: NONE] Sets the maximum amount of time any query on the pool can execute. Set RUNTIMECAP using interval, such as '1 minute' or '100 seconds' (see ***Interval Values*** (page 37) for details). This value cannot exceed one year. Setting this value

		to NONE specifies that there is no time limit on queries running on the pool. If the user or session also has a RUNTIMECAP, the shorter limit applies.
SINGLEINITIATOR	BOOLEAN	Value that indicates whether all requests using this pool are issued against the same initiator node or whether multiple initiator nodes can be used; for instance in a round-robin configuration.

Notes

Column names in the RESOURCE_POOL table mirror syntax in the ***CREATE RESOURCE POOL*** (page 753) statement; therefore, column names do not use underscores.

Example

To see values set for your resource pools:

```
SELECT * FROM V_CATALOG.RESOURCE_POOLS;
```

See also

CREATE RESOURCE POOL (page 753)

Managing Workloads and Monitoring Resource Pools and Resource Usage by Queries in the Administrator's Guide for usage and examples.

ROLES

Contains the names of all roles the user can access, along with any roles that have been assigned to those roles.

Column Name	Data Type	Description
ROLE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the role.
NAME	VARCHAR	The name of a role that the user can access.
ASSIGNED_ROLES	VARCHAR	The names of any roles that have been granted to this role. By enabling the role, the user also has access to the privileges of these additional roles. **Note:** If you see an asterisk in the ASSIGNED_ROLES column output, it means the user has roles WITH ADMIN OPTION.

Tip: You can also use the ***HAS_ROLE()*** (page 501) function to see if a role is available to a user.

Example

```
=> SELECT * FROM roles;

      role_id      |      name       | assigned_roles
-------------------+-----------------+----------------
 45035996273704964 | public          |
 45035996273704966 | dbduser         |
 45035996273704968 | dbadmin         | dbduser*
 45035996273704972 | pseudosuperuser | dbadmin*
 45035996273704974 | logreader       |
 45035996273704976 | logwriter       |
 45035996273704978 | logadmin        | logreader, logwriter
(7 rows)
```

See Also

GRANTS (page 944)

HAS_ROLE (page 501)

USERS (page 985)

Managing Users and Privileges and Viewing a user's role in the Administrator's Guide

SCHEMATA

Provides information about schemas in the database.

Column Name	Data Type	Description
SCHEMA_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the specific schema.
SCHEMA_NAME	VARCHAR	Schema name for which information is listed.
SCHEMA_OWNER_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the owner who created the schema.
SCHEMA_OWNER	VARCHAR	Name of the owner who created the schema.
SYSTEM_SCHEMA_CREATOR	VARCHAR	Creator information for system schema or NULL for non-system schema
CREATE_TIME	TIMESTAMPTZ	Time when the schema was created.
IS_SYSTEM_SCHEMA	BOOLEAN	Indicates whether the schema was created for system use, where *t* is true and *f* is false.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> SELECT * FROM schemata;

-[ RECORD 1 ]---------+------------------------------
schema_id             | 8300
schema_name           | v_internal
schema_owner_id       | 45035996273704962
schema_owner          | dbadmin
system_schema_creator | dbadmin
create_time           | 2012-03-22 11:19:46.311825-04
is_system_schema      | t
-[ RECORD 2 ]---------+------------------------------
schema_id             | 8301
schema_name           | v_catalog
schema_owner_id       | 45035996273704962
schema_owner          | dbadmin
system_schema_creator | dbadmin
create_time           | 2012-03-22 11:19:46.311905-04
is_system_schema      | t
-[ RECORD 3 ]---------+------------------------------
schema_id             | 8302
schema_name           | v_monitor
schema_owner_id       | 45035996273704962
schema_owner          | dbadmin
system_schema_creator | dbadmin
create_time           | 2012-03-22 11:19:46.31193-04
is_system_schema      | t
-[ RECORD 4 ]---------+------------------------------
schema_id             | 45035996273704968
schema_name           | public
schema_owner_id       | 45035996273704962
schema_owner          | dbadmin
system_schema_creator |
create_time           | 2012-03-22 11:19:40.75002-04
is_system_schema      | f
```

SEQUENCES

Displays information about the parameters specified for a sequence using the ***CREATE SEQUENCE*** (page 765) statement.

Column Name	Data Type	Description
SEQUENCE_SCHEMA	VARCHAR	Schema in which the sequence was created.

SEQUENCE_NAME	VARCHAR	Name of the sequence defined in the CREATE SEQUENCE statement.
OWNER_NAME	VARCHAR	Name of the owner; for example, dbadmin.
IDENTITY_TABLE_NAME	VARCHAR	If created by an identity column, the name of the table to which it belongs. See ***column constraints*** (page 783) in the ***CREATE TABLE*** (page 770) statement.
SESSION_CACHE_COUNT	INTEGER	Count of values cached in a session.
ALLOW_CYCLE	BOOLEAN	Values allowed to cycle when max/min is reached. See `CYCLE | NO CYCLE` parameter in ***CREATE SEQUENCE*** (page 765).
OUTPUT_ORDERED	BOOLEAN	Values guaranteed to be ordered (always false).
INCREMENT_BY	INTEGER	Sequence values are incremented by this number (negative for reverse sequences).
MINIMUM	INTEGER	Minimum value the sequence can generate.
MAXIMUM	INTEGER	Maximum value the sequence can generate.
CURRENT_VALUE	INTEGER	Current value of the sequence.
SEQUENCE_SCHEMA_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the schema.
SEQUENCE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the sequence.
OWNER_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the user who created the sequence.
IDENTITY_TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the table to which the column belongs (if created by an identity column).

Example

Create a simple sequence:

```
=> CREATE SEQUENCE my_seq MAXVALUE 5000 START 150;
CREATE SEQUENCE
```

Return information about the sequence you just created:

```
=> \x
Expanded display is on.
=> SELECT * FROM sequences;
-[ RECORD 1 ]-------+------------------
sequence_schema     | public
sequence_name       | my_seq
owner_name          | dbadmin
identity_table_name |
session_cache_count | 250000
```

```
allow_cycle         | f
output_ordered      | f
increment_by        | 1
minimum             | 1
maximum             | 5000
current_value       | 149
sequence_schema_id  | 45035996273704966
sequence_id         | 45035996273844996
owner_id            | 45035996273704962
identity_table_id   | 0
```

An identity column is a sequence available only for numeric column types. To identify what column in a table, if any, is an identity column, search the `COLUMNS` table to find the identity column in a table:

```
=> CREATE TABLE testid (c1 IDENTITY(1, 1, 1000), c2 INT)
=> \x
Expanded display is on.
=> SELECT * FROM COLUMNS WHERE is_identity='t' AND table_name='testid';
-[ RECORD 1 ]------------+------------------
table_id                 | 45035996274150730
table_schema             | public
table_name               | testid
is_system_table          | f
column_name              | c1
data_type                | int
data_type_id             | 6
data_type_length         | 8
character_maximum_length |
numeric_precision        |
numeric_scale            |
datetime_precision       |
interval_precision       |
ordinal_position         | 1
is_nullable              | f
column_default           |
is_identity              | t
```

Use the `SEQUENCES` table to get detailed information about the sequence in testid:

```
=> SELECT * FROM sequences WHERE identity_table_name='testid';
-[ RECORD 1 ]-------+--------------------
sequence_schema     | public
sequence_name       | testid_c1_seq
owner_name          | dbadmin
identity_table_name | testid
session_cache_count | 1000
allow_cycle         | f
output_ordered      | f
increment_by        | 1
minimum             | 1
maximum             | 9223372036854775807
current_value       | 0
sequence_schema_id  | 45035996273704976
sequence_id         | 45035996274150770
owner_id            | 45035996273704962
```

```
identity_table_id    | 45035996274150768
```

Use the vsql command `\ds` to return a list of sequences. The following results show the two sequences created in the preceding examples. If more sequences existed, the table would list them.

The CurrentValue of the new sequence is one less than the start number you specified in the `CREATE SEQUENCE` and `IDENTITY` commands, because you have not yet used ***NEXTVAL*** (page 351) to instantiate the sequences to assign their cache or supply their first start values.

```
=> \ds
 List of Sequences
-[ RECORD 1 ]+--------------------
Schema       | public
Sequence     | my_seq
CurrentValue | 149
IncrementBy  | 1
Minimum      | 1
Maximum      | 5000
AllowCycle   | f
Comment      |
-[ RECORD 2 ]+--------------------
Schema       | public
Sequence     | testid_c1_seq
CurrentValue | 0
IncrementBy  | 1
Minimum      | 1
Maximum      | 9223372036854775807
AllowCycle   | f
Comment      |
```

See Also

CREATE SEQUENCE (page 765)

The \d [PATTERN] meta-commands in the Programmer's Guide

Using Named Sequences in the Administrator's Guide

STORAGE_LOCATIONS

Provides information about storage locations, their IDs labels, and status.

Column Name	Data Type	Description
LOCATION_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the storage location.
NODE_NAME	VARCHAR	The node name on which the storage location exists.
LOCATION_PATH	VARCHAR	The path where the storage location is mounted.

LOCATION_USAGE	VARCHAR	The type of information stored in the location: ▪ DATA: Only data is stored in the location. ▪ TEMP: Only temporary files that are created during loads or queries are stored in the location. ▪ DATA,TEMP: Both types of files are stored in the location. ▪ USER: The storage location can be used by non-dbadmin users, who are granted access to the storage location ▪ CATALOG: The area is used for the HP Vertica catalog. This usage is set internally and cannot be removed or changed.
IS_RETIRED	BOOLEAN	Whether the storage location has been retired. This column has a value of `t (true)` if the location is retired, or `f (false)` if it is not.
LOCATION_LABEL	VARCHAR	The label associated with a specific storage location, added with the ***ALTER_LOCATION_LABEL*** (page 430) function.
RANK	INTEGER	The Access Rank value either assigned or supplied to the storage location, as described in Prioritizing Column Access Speed.
THROUGHPUT	INTEGER	The throughput performance of the storage location, measured in MB/sec. You can get location performance values using ***MEASURE_LOCATION_PERFORMANCE*** (page 511), and set them with the ***SET_LOCATION_PERFORMANCE*** (page 532) function.
LATENCY	INTEGER	The measured latency of the storage location as number of data seeks per second. You can get location performance values using ***MEASURE_LOCATION_PERFORMANCE*** (page 511), and set them with the ***SET_LOCATION_PERFORMANCE*** (page 532) function.

Permissions

Must be a superuser

Example

Query the STORAGE_LOCATIONS on a one-node cluster database:

```
onenode=> SELECT * FROM storage_locations;
```

```
-[ RECORD 1 ]--+-----------------------------------------------
location_id    | 45035996273704982
node_name      | v_onenode_node0001
location_path  | /home/dbadmin/onenode/v_onenode_node0001_data
location_usage | DATA,TEMP
is_retired     | f
location_label |
rank           | 0
throughput     | 0
latency        | 0
```

The next example uses the Vmart example database.

```
VMart=> select * from storage_locations;
    location_id    |    node_name     |              location_path               | location_usage
| is_retired | location_label | rank | throughput | latency
------------------+------------------+------------------------------------------+-------------
--+------------+---------------+------+------------+---------
 45035996273704982 | v_vmart_node0001 | /home/dbadmin/VMart/v_vmart_node0001_data | DATA,TEMP      |
f          |                |    1 |         24 |      38
 45035996273721840 | v_vmart_node0001 | home/dbadmin/SSD/schemas                  | DATA           |
f          |                |    0 |         46 |      42
 45035996273770760 | v_vmart_node0001 | /home/dbadmin/SSD/tables                  | DATA           |
f          | SSD            |    2 |          0 |       0
 45035996273770762 | v_vmart_node0001 | /home/dbadmin/SSD/schemas                 | DATA           |
f          | Schema         |    2 |          0 |       0
 45035996273789564 | v_vmart_node0002 | /home/dbadmin/VMart/v_vmart_node0002_data | DATA,TEMP      |
f          |                |    0 |          0 |       0
 45035996273828402 | v_vmart_node0002 | /home/dbadmin/SSD/tables                  | DATA           |
f          |                |    0 |          0 |       0
 45035996273828406 | v_vmart_node0002 | /home/dbadmin/SSD/schemas                 | DATA           |
f          |                |    0 |          0 |       0
 45035996273789596 | v_vmart_node0003 | /home/dbadmin/VMart/v_vmart_node0003_data | DATA,TEMP      |
f          |                |    0 |          0 |       0
 45035996273828404 | v_vmart_node0003 | /home/dbadmin/SSD/tables                  | DATA           |
f          |                |    0 |          0 |       0
 45035996273828408 | v_vmart_node0003 | /home/dbadmin/SSD/schemas                 | DATA           |
f          |                |    0 |          0 |       0
(10 rows)
```

See Also

DISK_STORAGE (page 1014)

MEASURE_LOCATION_PERFORMANCE (page 511)

SET_LOCATION_PERFORMANCE (page 532)

STORAGE_POLICIES (page 1101)

STORAGE_USAGE (page 1104)

Storage Management Functions (page 636)

Creating and Configuring Storage Locations in the Administrator's Guide

SYSTEM_COLUMNS

Provides table column information for ***SYSTEM_TABLES*** (page 976).

Column Name	Data Type	Description
TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the table.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
TABLE_NAME	VARCHAR	The table name for which information is listed.
IS_SYSTEM_TABLE	BOOLEAN	Indicates whether the table is a system table, where *t* is true and *f* is false.
COLUMN_ID	VARCHAR	A unique VARCHAR ID, assigned by the HP Vertica catalog, that identifies a column in a table.
COLUMN_NAME	VARCHAR	The column name for which information is listed in the database.
DATA_TYPE	VARCHAR	The data type assigned to the column; for example VARCHAR(16).
DATA_TYPE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the data type.
DATA_TYPE_LENGTH	INTEGER	The maximum allowable length of the data type.
CHARACTER_MAXIMUM_LENGTH	INTEGER	The maximum allowable length of the column.
NUMERIC_PRECISION	INTEGER	The number of significant decimal digits.
NUMERIC_SCALE	INTEGER	The number of fractional digits.
DATETIME_PRECISION	INTEGER	For TIMESTAMP data type, returns the declared precision; returns null if no precision was declared.
INTERVAL_PRECISION	INTEGER	The number of fractional digits retained in the seconds field.
ORDINAL_POSITION	INTEGER	The position of the column respective to other columns in the table.
IS_NULLABLE	BOOLEAN	Indicates whether the column can contain null values, where *t* is true and *f* is false.
COLUMN_DEFAULT	VARCHAR	The default value of a column, such as empty or expression.

Example

```
=> SELECT table_schema, table_name, column_name, column_id, data_type
```

```
    FROM system_columns
    WHERE table_name = 'projection_columns' ORDER BY 1,2;

 table_schema |     table_name     |          column_name          | column_id   |   data_type
--------------+--------------------+-------------------------------+-------------+---------------
 v_catalog    | projection_columns | statistics_updated_timestamp | 10038-17    | timestamptz
 v_catalog    | projection_columns | statistics_type               | 10038-16    | varchar(8192)
 v_catalog    | projection_columns | table_column_name             | 10038-15    | varchar(128)
 v_catalog    | projection_columns | table_column_id               | 10038-14    | varchar(41)
 v_catalog    | projection_columns | table_name                    | 10038-13    | varchar(128)
 v_catalog    | projection_columns | table_id                      | 10038-12    | int
 v_catalog    | projection_columns | table_schema                  | 10038-11    | varchar(128)
 v_catalog    | projection_columns | group_id                      | 10038-10    | int
 v_catalog    | projection_columns | access_rank                   | 10038-9     | int
 v_catalog    | projection_columns | encoding_type                 | 10038-8     | varchar(18)
 v_catalog    | projection_columns | data_type                     | 10038-7     | varchar(128)
 v_catalog    | projection_columns | column_id                     | 10038-6     | int
 v_catalog    | projection_columns | sort_position                 | 10038-5     | int
 v_catalog    | projection_columns | column_position               | 10038-4     | int
 v_catalog    | projection_columns | projection_column_name        | 10038-3     | varchar(128)
 v_catalog    | projection_columns | projection_name               | 10038-2     | varchar(128)
 v_catalog    | projection_columns | projection_id                 | 10038-1     | int
(17 rows)
```

SYSTEM_TABLES

Returns a list of all system table names.

Column Name	Data Type	Description
TABLE_SCHEMA_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the schema.
TABLE_SCHEMA	VARCHAR	The schema name in which the system table resides; for example, V_CATALOG (page 933) or V_MONITOR (page 989).
TABLE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the table.
TABLE_NAME	VARCHAR	The name of the system table.
TABLE_DESCRIPTION	VARCHAR	A description of the system table's purpose.

Example

Call all the system tables and order them by schema:

```
=> SELECT * FROM system_tables ORDER BY 1, 2;
```

Ask for tables related to column information:

```
=> SELECT * FROM system_tables WHERE table_name ILIKE '%col%';
```

```
 table_schema_id | table_schema | table_id |     table_name     |         table_description
-----------------+--------------+----------+--------------------+-----------------------------
----
            8301 | v_catalog    |    10038 | projection_columns | Projection columns ...
            8301 | v_catalog    |    10212 | system_columns     | System column ...
            8301 | v_catalog    |    10208 | odbc_columns       | An ODBC compliant ...
            8301 | v_catalog    |    10134 | constraint_columns | Table column constraint ...
            8301 | v_catalog    |    10000 | columns            | Table column information ...
            8301 | v_catalog    |    10024 | view_columns       | View column information ...
            8302 | v_monitor    |    10200 | column_storage     | Information on amount of disk ...
            8302 | v_monitor    |    10058 | data_collector     | Statistics on usage Data Col...
(8 rows)
```

TABLE_CONSTRAINTS

Provides information about table constraints.

Column Name	Data Type	Description
CONSTRAINT_ID	VARCHAR	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the constraint.
CONSTRAINT_NAME	VARCHAR	The name of the constraint, if specified as UNIQUE, FOREIGN KEY, NOT NULL, or PRIMARY KEY.
CONSTRAINT_SCHEMA_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the schema containing the constraint.
CONSTRAINT_KEY_COUNT	INTEGER	The number of constraint keys.
FOREIGN_KEY_COUNT	INTEGER	The number of foreign keys.
TABLE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the table.
FOREIGN_TABLE_ID	INTEGER	The unique object ID of the foreign table referenced in a foreign key constraint (zero if not a foreign key constraint).
CONSTRAINT_TYPE	INTEGER	Is one of 'c', 'f', 'p', 'U' or 'd,' which refer to 'check', 'foreign', 'primary', 'unique' and 'determines', respectively.

Example

The following command returns constraint column names and types against the VMart schema.

```
vmartdb=> SELECT constraint_name, constraint_type FROM table_constraints
          ORDER BY constraint_type;

      constraint_name      | constraint_type
---------------------------+-----------------
 fk_online_sales_promotion | f
```

```
 fk_online_sales_warehouse | f
 fk_online_sales_shipping  | f
 fk_online_sales_op        | f
 fk_online_sales_cc        | f
 fk_online_sales_customer  | f
 fk_online_sales_product   | f
 fk_online_sales_shipdate  | f
 fk_online_sales_saledate  | f
 fk_store_orders_employee  | f
 fk_store_orders_vendor    | f
 fk_store_orders_store     | f
 fk_store_orders_product   | f
 fk_store_sales_employee   | f
 fk_store_sales_customer   | f
 fk_store_sales_promotion  | f
 fk_store_sales_store      | f
 fk_store_sales_product    | f
 fk_store_sales_date       | f
 fk_inventory_warehouse    | f
 fk_inventory_product      | f
 fk_inventory_date         | f
 -                         | p
 -                         | p
 -                         | p
 -                         | p
 -                         | p
 -                         | p
 -                         | p
 -                         | p
 -                         | p
 -                         | p
 -                         | p
(33 rows)
```

See Also

ANALYZE_CONSTRAINTS (page 432)

Adding Constraints in the Administrator's Guide

TABLES

Provides information about all tables in the database.

Column Name	Data Type	Description
TABLE_SCHEMA_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the schema.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.

TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the table.
TABLE_NAME	VARCHAR	The table name for which information is listed.
OWNER_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the owner.
OWNER_NAME	VARCHAR	The name of the user who created the table.
IS_TEMP_TABLE	BOOLEAN	Indicates whether table is a system table, where *t* is true and *f* is false.
IS_SYSTEM_TABLE	BOOLEAN	Indicates whether table is a temporary table, where *t* is true and *f* is false.
SYSTEM_TABLE_CREATOR	VARCHAR	The name of the process that created the table, such as Designer.
PARTITION_EXPRESSION	VARCHAR	The partition expression for the table.
CREATE_TIME	TIMESTAMP	Returns the timestamp for when the table was created.
TABLE_DEFINITION	VARCHAR	The COPY statement table definition. This column is applicable only to external tables.

Notes

The TABLE_SCHEMA and TABLE_NAME columns are case sensitive when you run queries that contain the equality (=) predicate. Use the ILIKE predicate instead:

```
=> SELECT table_schema, table_name FROM v_catalog.tables
   WHERE table_schema ILIKE 'schema1';
```

Example

To return information on all tables in the Vmart schema:

```
vmartdb=> SELECT table_schema, table_name, owner_name, is_system_table
          FROM tables;
 table_schema |      table_name       | owner_name | is_system_table
--------------+-----------------------+------------+-----------------
 public       | customer_dimension    | release    | f
 public       | product_dimension     | release    | f
 public       | promotion_dimension   | release    | f
 public       | date_dimension        | release    | f
 public       | vendor_dimension      | release    | f
 public       | employee_dimension    | release    | f
 public       | shipping_dimension    | release    | f
 public       | warehouse_dimension   | release    | f
 public       | inventory_fact        | release    | f
 store        | store_dimension       | release    | f
 store        | store_sales_fact      | release    | f
 store        | store_orders_fact     | release    | f
 online_sales | online_page_dimension | release    | f
 online_sales | call_center_dimension | release    | f
 online_sales | online_sales_fact     | release    | f
```

```
(15 rows)
```

To return the timestamp for when the tables were created:

```
vmartdb=> SELECT table_schema, table_name, create_time FROM tables;
 table_schema |      table_name       |          create_time
--------------+-----------------------+-------------------------------
 public       | customer_dimension    | 2011-08-15 11:18:25.784203-04
 public       | product_dimension     | 2011-08-15 11:18:25.815653-04
 public       | promotion_dimension   | 2011-08-15 11:18:25.850592-04
 public       | date_dimension        | 2011-08-15 11:18:25.892347-04
 public       | vendor_dimension      | 2011-08-15 11:18:25.942805-04
 public       | employee_dimension    | 2011-08-15 11:18:25.966985-04
 public       | shipping_dimension    | 2011-08-15 11:18:25.999394-04
 public       | warehouse_dimension   | 2011-08-15 11:18:26.461297-04
 public       | inventory_fact        | 2011-08-15 11:18:26.513525-04
 store        | store_dimension       | 2011-08-15 11:18:26.657409-04
 store        | store_sales_fact      | 2011-08-15 11:18:26.737535-04
 store        | store_orders_fact     | 2011-08-15 11:18:26.825801-04
 online_sales | online_page_dimension | 2011-08-15 11:18:27.007329-04
 online_sales | call_center_dimension | 2011-08-15 11:18:27.476844-04
 online_sales | online_sales_fact     | 2011-08-15 11:18:27.49749-04
(15 rows)
```

TYPES

Provides information about supported data types.

Note: This table was updated with new columns in Release 5.1.

Column Name	Data Type	Description
TYPE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the specific data type.
ODBC_TYPE	INTEGER	The numerical ODBC type.
ODBC_SUBTYPE	INTEGER	The numerical ODBC subtype, used to differentiate types such as time and interval that have multiple subtypes.
MIN_SCALE	INTEGER	The minimum number of digits supported to the right of the decimal point for the data type.
MAX_SCALE	INTEGER	The maximum number of digits supported to the right of the decimal point for the data type. A value of 0 is used for types that do not use decimal points.
COLUMN_SIZE	INTEGER	The number of characters required to display the type. See: ***http://msdn.microsoft.com/en-us/library/windows/desktop/ms711786%28v=VS.85%29.aspx*** ***http://www.*** for the details on COLUMN_SIZE for each type.

INTERVAL_MASK	INTEGER	For data types that are intervals, the bitmask to determine the range of the interval from the HP Vertica TYPE_ID. Details are available in the HP Vertica SDK.
TYPE_NAME	VARCHAR	The data type name associated with a particular data type ID.
CREATION_PARAMETERS	VARCHAR	A list of keywords, separated by commas, corresponding to each parameter that the application may specify in parentheses when using the name that is returned in the TYPE_NAME field. The keywords in the list can be any of the following: length, precision, or scale. They appear in the order that the syntax requires them to be used.

Example

```
dbadmin=> \x
Expanded display is on.
dbadmin=> select * from types limit 3;
-[ RECORD 1 ]-------+-------------------------
type_id             | 5
odbc_type           | -7
odbc_subtype        | 0
min_scale           | 0
max_scale           | 0
column_size         | 1
interval_mask       | 0
type_name           | Boolean
creation_parameters |
-[ RECORD 2 ]-------+-------------------------
type_id             | 6
odbc_type           | -5
odbc_subtype        | 0
min_scale           | 0
max_scale           | 0
column_size         | 20
interval_mask       | 0
type_name           | Integer
creation_parameters |
-[ RECORD 3 ]-------+-------------------------
type_id             | 7
odbc_type           | 8
odbc_subtype        | 0
min_scale           | 0
max_scale           | 0
column_size         | 15
interval_mask       | 0
type_name           | Float
creation_parameters | precision
```

USER_AUDITS

Lists the results of database and object size audits generated by users calling the ***AUDIT*** (page 446) function. See Monitoring Database Size for License Compliance in the Administrator's Guide for more information.

Column Name	Data Type	Description
SIZE_BYTES	INTEGER	The estimated raw data size of the database
USER_ID	INTEGER	The ID of the user who generated the audit
USER_NAME	VARCHAR	The name of the user who generated the audit
OBJECT_ID	INTEGER	The ID of the object being audited
OBJECT_TYPE	VARCHAR	The type of object being audited (table, schema, etc.)
OBJECT_SCHEMA	VARCHAR	The schema containing the object being audited
OBJECT_NAME	VARCHAR	The name of the object being audited
AUDIT_START_TIMESTAMP	TIMESTAMPTZ	When the audit started
AUDIT_END_TIMESTAMP	TIMESTAMPTZ	When the audit finished
CONFIDENCE_LEVEL_PERCENT	FLOAT	The confidence level of the size estimate
ERROR_TOLERANCE_PERCENT	FLOAT	The error tolerance used for the size estimate
USED_SAMPLING	BOOLEAN	Whether data was randomly sampled (if false, all of the data was analyzed)
CONFIDENCE_INTERVAL_LOWER_BOUND_BYTES	INTEGER	The lower bound of the data size estimate within the confidence level
CONFIDENCE_INTERVAL_UPPER_BOUND_BYTES	INTEGER	The upper bound of the data size estimate within the confidence level
SAMPLE_COUNT	INTEGER	The number of data samples used to generate the estimate
CELL_COUNT	INTEGER	The number of cells in the database

USER_FUNCTIONS

Returns metadata about user-defined SQL functions (which store commonly used SQL expressions as a function in the HP Vertica catalog) and User Defined functions (UDx).

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	The name of the schema in which this function exists.
FUNCTION_NAME	VARCHAR	The name assigned by the user to the SQL function or User Defined Function.
PROCEDURE_TYPE	VARCHAR	The type of user defined function. For example, 'User Defined Function'.
FUNCTION_RETURN_TYPE	VARCHAR	The data type name that the SQL function returns.
FUNCTION_ARGUMENT_TYPE	VARCHAR	The number and data types of parameters for the function.
FUNCTION_DEFINITION	VARCHAR	The SQL expression that the user defined in the SQL function's function body.
VOLATILITY	VARCHAR	The SQL function's volatility (whether a function returns the same output given the same input). Can be immutable, volatile, or stable.
IS_STRICT	BOOLEAN	Indicates whether the SQL function is strict, where *t* is true and *f* is false.
IS_FENCED	BOOLEAN	Indicates whether the function runs in Fenced Mode or not.
COMMENT	VARCHAR	A comment about this function provided by the function creator.

Notes

- The volatility and strictness of a SQL function are automatically inferred from the function definition in order that HP Vertica perform constant folding optimization, when possible, and determine the correctness of usage, such as where an immutable function is expected but a volatile function is provided.
- The volatility and strictness of UDx is set by the superuser when creating the function. See ***CREATE FUNCTION (UDF)*** (page 725) and ***CREATE TRANSFORM FUNCTION*** (page 734) for details.

Example

Create a SQL function called `myzeroifnull` in the public schema:

```
=> CREATE FUNCTION myzeroifnull(x INT) RETURN INT
   AS BEGIN
     RETURN (CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END);
   END;
```

Now query the `USER_FUNCTIONS` table. The query returns just the `myzeroifnull` macro because it is the only one created in this schema:

```
=> SELECT * FROM user_functions;
-[ RECORD 1 ]----------+--------------------------------------------------
schema_name            | public
```

```
function_name          | myzeroifnull
procedure_type         | User Defined Function
function_return_type   | Integer
function_argument_type | x Integer
function_definition    | RETURN CASE WHEN (x IS NOT NULL) THEN x ELSE 0 END
volatility             | immutable
is_strict              | f
is_fenced              | f
comment                |
```

See Also

CREATE FUNCTION (page 722)

ALTER FUNCTION (page 656)

DROP FUNCTION (page 811)

GRANT (Function) (page 843)

REVOKE (Function) (page 864)

See also Using SQL Functions in the Programmer's Guide

USER_PROCEDURES

Provides information about external procedures that have been defined for HP Vertica. User see only the procedures they can execute.

Column Name	Data Type	Description
PROCEDURE_NAME	VARCHAR	The name given to the external procedure through the CREATE PROCEDURE statement.
PROCEDURE_ARGUMENTS	VARCHAR	Lists arguments for the external procedure.
SCHEMA_NAME	VARCHAR	Indicates the schema in which the external procedure is defined.

Example

```
=> SELECT * FROM user_procedures;

 procedure_name | procedure_arguments | schema_name
----------------+---------------------+-------------
 helloplanet    | arg1 Varchar        | public
(1 row)
```

USERS

Provides information about all users in the database.

Column Name	Data Type	Description
USER_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	The user name for which information is listed.
IS_SUPER_USER	BOOLEAN	Indicates whether the current user is superuser, where *t* is true and *f* is false.
PROFILE_NAME	VARCHAR	The name of the profile to which the user is assigned. The profile controls the user's password policy.
IS_LOCKED	BOOLEAN	Whether the user's account is locked. A locked user cannot log into the system.
LOCK_TIME	DATETIME	When the user's account was locked. Used to determine when to automatically unlock the account, if the user's profile has a PASSWORD_LOCK_TIME parameter set.
RESOURCE_POOL	VARCHAR	The resource pool to which the user is assigned.
MEMORY_CAP_KB	VARCHAR	The maximum amount of memory a query run by the user can consume, in kilobytes.
TEMP_SPACE_CAP_KB	VARCHAR	The maximum amount of temporary disk space a query run by the user can consume, in kilobytes.
RUN_TIME_CAP	VARCHAR	The maximum amount of time any of the user's queries is allowed to run.
ALL_ROLES	VARCHAR	Roles assigned to the user. An asterisk in ALL_ROLES output means role granted WITH ADMIN OPTION. See Database Roles in the Administrator's Guide.
DEFAULT_ROLES	VARCHAR	Default role(s) assigned to the user. An asterisk in DEFAULT_ROLES output means role granted WITH ADMIN OPTION. See Default roles for database users in the Administrator's Guide.
SEARCH_PATH	VARCHAR	Sets the default schema search path for the user. See Setting Schema Search Paths in the Administrator's Guide.

Notes

You can call the ***HAS_ROLE()*** (page 501) function to see if a role has been assigned to a user.

Example

```
=> \x
Expanded display is on.
=> SELECT * FROM users;

-[ RECORD 1 ]-----+--------------------------
user_id           | 45035996273704962
user_name         | dbadmin
is_super_user     | t
profile_name      | default
is_locked         | f
lock_time         |
resource_pool     | general
memory_cap_kb     | unlimited
temp_space_cap_kb | unlimited
run_time_cap      | unlimited
all_roles         | dbadmin*, pseudosuperuser*
default_roles     | dbadmin*, pseudosuperuser*
search_path       | "$user", public, v_catalog, v_monitor, v_internal
-[ RECORD 2 ]-----+--------------------------
user_id           | 45035996273713664
user_name         | Alice
is_super_user     | f
profile_name      | default
is_locked         | f
lock_time         |
resource_pool     | general
memory_cap_kb     | unlimited
temp_space_cap_kb | unlimited
run_time_cap      | unlimited
all_roles         | logadmin
default_roles     |
search_path       | "$user", public, v_catalog, v_monitor, v_internal
-[ RECORD 3 ]-----+--------------------------
user_id           | 45035996273714428
user_name         | Bob
is_super_user     | f
profile_name      | default
is_locked         | f
lock_time         |
resource_pool     | general
memory_cap_kb     | unlimited
temp_space_cap_kb | unlimited
run_time_cap      | unlimited
all_roles         | logadmin, commentor*
default_roles     |
search_path       | "$user", public, v_catalog, v_monitor, v_internal
```

Note: An asterisk in the output means role WITH ADMIN OPTION.

See Also

GRANTS (page 944)

HAS_ROLE (page 501)

ROLES (page 967)

Managing Users and Privileges in the Administrator's Guide

VIEW_COLUMNS

Provides view attribute information.

Column Name	Data Type	Description
TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the view of the table.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
TABLE_NAME	VARCHAR	The table name for which information is listed.
COLUMN_ID	VARCHAR	A unique VARCHAR ID, assigned by the HP Vertica catalog, that identifies a column in a table.
COLUMN_NAME	VARCHAR	The column name for which information is listed.
DATA_TYPE	VARCHAR	The data type of the column for which information is listed; for example, VARCHAR(128).
DATA_TYPE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the data type.
DATA_TYPE_LENGTH	INTEGER	The maximum allowable length for the data type.
CHARACTER_MAXIMUM_LENGTH	INTEGER	The maximum allowable length for the column, valid for character types.
NUMERIC_PRECISION	INTEGER	The number of significant decimal digits.
NUMERIC_SCALE	INTEGER	The number of fractional digits.
DATETIME_PRECISION	INTEGER	For TIMESTAMP data type, returns the declared precision; returns null if no precision was declared.
INTERVAL_PRECISION	INTEGER	The number of fractional digits retained in the seconds field.
ORDINAL_POSITION	INTEGER	The position of the column respective to other columns.

Notes

A warning like the following means only that view <t> had its associated table dropped. The view is not returned by the SELECT * FROM view_columns command, and the warning is returned merely to notify users about an orphaned view.

```
WARNING:  invalid view v: relation "public.t" does not exist
```

Example

NULL fields in the results indicate that those columns were not defined. For example, given the following table, the result for the datetime_precision column is NULL because no precision was declared:

```
=> CREATE TABLE c (c TIMESTAMP);
CREATE TABLE

=> SELECT table_name, column_id, column_name, datetime_precision
     FROM columns WHERE table_name = 'c';

 table_name |      column_id      | column_name | datetime_precision
------------+---------------------+-------------+--------------------
 c          | 45035996273720664-1 | c           |
(1 row)
```

In the next statement, the datetime_precision column returns 4 because the precision was declared as 4 in the CREATE TABLE statement:

```
=> DROP TABLE c;

=> CREATE TABLE c (c TIMESTAMP(4));
CREATE TABLE

=> SELECT table_name, column_id, column_name, datetime_precision
     FROM columns WHERE table_name = 'c';
 table_name |      column_id      | column_name | datetime_precision
------------+---------------------+-------------+--------------------
 c          | 45035996273720700-1 | c           |                  4
(1 row)
```

See Also

VIEWS (page 988)

VIEWS

Provides information about all views within the system. See Implementing Views for more information.

Column Name	Data Type	Description
TABLE_SCHEMA_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the schema of the table that the view references.
TABLE_SCHEMA	VARCHAR	The name of the schema that contains the view.
TABLE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the view.
TABLE_NAME	VARCHAR	The view name for which information is listed.
OWNER_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the view owner.
OWNER_NAME	VARCHAR	The name of the view owner.
VIEW_DEFINITION	VARCHAR	The query that defines the view.
IS_SYSTEM_VIEW	BOOLEAN	Indicates whether the table is a system view, where *t* is true and *f* is false.
SYSTEM_VIEW_CREATOR	VARCHAR	The user name who created the view.
CREATE_TIME	TIMESTAMP	The date/time the view was created.

Example

Query the VIEWS table:

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM VIEWS;
-[ RECORD 1 ]-------+---------------------------
table_schema_id     | 45035996273704976
table_schema        | public
table_id            | 45035996273951536
table_name          | testview
owner_id            | 45035996273704962
owner_name          | dbadmin
view_definition     | SELECT bar.x FROM public.bar
is_system_view      | f
system_view_creator |
create_time         | 2013-01-14 12:02:03.244809-05
```

See Also

Implementing Views

VIEW_COLUMNS (page 987)

V_MONITOR Schema

The system tables in this section reside in the `v_monitor` schema. These tables provide information about the health of the HP Vertica database.

ACTIVE_EVENTS

Returns all active events in the cluster. See Monitoring Events.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name where the event occurred.
EVENT_CODE	INTEGER	A numeric ID that indicates the type of event. See Event Types for a list of event type codes.
EVENT_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the specific event.
EVENT_SEVERITY	VARCHAR	The severity of the event from highest to lowest. These events are based on standard syslog severity types. ▪ 0—Emergency ▪ 1—Alert ▪ 2—Critical ▪ 3—Error ▪ 4—Warning ▪ 5—Notice ▪ 6—Informational ▪ 7—Debug
EVENT_POSTED_TIMESTAMP	TIMESTAMP	The year, month, day, and time the event was reported. The time is posted in military time.
EVENT_EXPIRATION	VARCHAR	The year, month, day, and time the event expire. The time is posted in military time. If the cause of the event is still active, the event is posted again.
EVENT_CODE_DESCRIPTION	VARCHAR	A brief description of the event and details pertinent to the specific situation.
EVENT_PROBLEM_DESCRIPTION	VARCHAR	A generic description of the event.
REPORTING_NODE	VARCHAR	The name of the node within the cluster that reported the event.
EVENT_SENT_TO_CHANNELS	VARCHAR	The event logging mechanisms that are configured for HP Vertica. These can include `vertica.log`, (configured by default) syslog, and SNMP.
EVENT_POSTED_COUNT	INTEGER	Tracks the number of times an event occurs. Rather than posting the same event multiple times, HP Vertica posts the event once and then counts the number of additional instances in which the event occurs.

Example

Query the ACTIVE_EVENTS table:

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM active_events;

-[ RECORD 1 ]-------------+----------------------------------------
node_name                 | site01
event_code                | 6
event_id                  | 6
event_severity            | Informational
is_event_posted           | 2009-08-11 09:38:39.008458
event_expiration          | 2077-08-29 11:52:46.008458
event_code_description    | Node State Change
event_problem_description | Changing node site01 startup state to UP
reporting_node            | site01
event_sent_to_channels    | Vertica Log
event_posted_count        | 1
-[ RECORD 2 ]-------------+----------------------------------------
node_name                 | site02
event_code                | 6
event_id                  | 6
event_severity            | Informational
is_event_posted           | 2009-08-11 09:38:39.018172
event_expiration          | 2077-08-29 11:52:46.018172
event_code_description    | Node State Change
event_problem_description | Changing node site02 startup state to UP
reporting_node            | site02
event_sent_to_channels    | Vertica Log
event_posted_count        | 1
-[ RECORD 3 ]-------------+----------------------------------------
current_timestamp         | 2009-08-11 14:38:48.859987
node_name                 | site03
event_code                | 6
event_id                  | 6
event_severity            | Informational
is_event_posted           | 2009-08-11 09:38:39.027258
event_expiration          | 2077-08-29 11:52:46.027258
event_code_description    | Node State Change
event_problem_description | Changing node site03 startup state to UP
reporting_node            | site03
event_sent_to_channels    | Vertica Log
event_posted_count        | 1
-[ RECORD 4 ]-------------+----------------------------------------
node_name                 | site04
event_code                | 6
event_id                  | 6
event_severity            | Informational
is_event_posted           | 2009-08-11 09:38:39.008288
event_expiration          | 2077-08-29 11:52:46.008288
event_code_description    | Node State Change
event_problem_description | Changing node site04 startup state to UP
```

```
reporting_node            | site04
event_sent_to_channels    | Vertica Log
event_posted_count        | 1
...
```

COLUMN_STORAGE

Returns the amount of disk storage used by each column of each projection on each node.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
COLUMN_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the column.
COLUMN_NAME	VARCHAR	The column name for which information is listed.
ROW_COUNT	INTEGER	The number of rows in the column.
USED_BYTES	INTEGER	The disk storage allocation of the column in bytes.
ENCODINGS	VARCHAR	The encoding type for the column.
COMPRESSION	VARCHAR	The compression type for the column.
WOS_ROW_COUNT	INTEGER	The number of WOS rows in the column.
ROS_ROW_COUNT	INTEGER	The number of ROS rows in the column.
ROS_USED_BYTES	INTEGER	The number of ROS bytes in the column.
ROS_COUNT	INTEGER	The number of ROS containers.
PROJECTION_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	The associated projection name for the column.
PROJECTION_SCHEMA	VARCHAR	The name of the schema associated with the projection.
ANCHOR_TABLE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the anchor table.
ANCHOR_TABLE_NAME	VARCHAR	The associated table name.
ANCHOR_TABLE_SCHEMA	VARCHAR	The associated table's schema name.
ANCHOR_TABLE_COLUMN_ID	VARCHAR	A unique VARCHAR ID, assigned by the HP Vertica catalog, that identifies a column in a table.
ANCHOR_TABLE_COLUMN_NAME	VARCHAR	The name of the anchor table.

Notes

- WOS data is stored by row, so per-column byte counts are not available.
- The ENCODINGS and COMPRESSION columns let you compare how different encoding types affect column storage, when optimizing for compression.

Example

Query the COLUMN_STORAGE table:

```
=> \pset expanded
Expanded display is on.

=> SELECT * FROM column_storage;

-[ RECORD 1 ]------------+--------------------
node_name                | v_onenode_node0001
column_id                | 45035996273718840
column_name              | stock
row_count                | 1
used_bytes               | 31
encodings                | String
compressions             | lzo
wos_row_count            | 0
ros_row_count            | 1
ros_used_bytes           | 31
ros_count                | 1
projection_id            | 45035996273718838
projection_name          | trades_p
projection_schema        | public
anchor_table_id          | 45035996273718836
anchor_table_name        | trades
anchor_table_schema      | public
anchor_table_column_id   | 45035996273718836-1
anchor_table_column_name | stock
-[ RECORD 2 ]------------+--------------------
node_name                | v_onenode_node0001
column_id                | 45035996273718842
column_name              | bid
row_count                | 1
used_bytes               | 68
encodings                | Int_Delta
compressions             | lzo
wos_row_count            | 0
ros_row_count            | 1
ros_used_bytes           | 68
ros_count                | 1
projection_id            | 45035996273718838
projection_name          | trades_p
projection_schema        | public
anchor_table_id          | 45035996273718836
anchor_table_name        | trades
anchor_table_schema      | public
anchor_table_column_id   | 45035996273718836-2
```

```
anchor_table_column_name | bid
-[ RECORD 3 ]------------+--------------------
node_name                | v_onenode_node0001
column_id                | 45035996273718846
column_name              | ask
row_count                | 1
used_bytes               | 0
encodings                | Uncompressed
compressions             | lzo
wos_row_count            | 0
ros_row_count            | 1
ros_used_bytes           | 0
ros_count                | 1
projection_id            | 45035996273718838
projection_name          | trades_p
projection_schema        | public
anchor_table_id          | 45035996273718836
anchor_table_name        | trades
anchor_table_schema      | public
anchor_table_column_id   | 45035996273718836-3
anchor_table_column_name | ask
-[ RECORD 4 ]------------+--------------------
node_name                | v_onenode_node0001
column_id                | 45035996273718848
column_name              | epoch
row_count                | 1
used_bytes               | 48
encodings                | Int_Delta
compressions             | none
wos_row_count            | 0
ros_row_count            | 1
ros_used_bytes           | 48
ros_count                | 1
projection_id            | 45035996273718838
projection_name          | trades_p
projection_schema        | public
anchor_table_id          | 45035996273718836
anchor_table_name        | trades
anchor_table_schema      | public
anchor_table_column_id   | 45035996273718836-4
anchor_table_column_name |
```

Call specific columns from the `COLUMN_STORAGE` table:

```
SELECT column_name, row_count, projection_name, anchor_table_name
FROM column_storage WHERE node_name = 'site02' AND row_count = 1000;
```

```
     column_name      | row_count |       projection_name        |   anchor_table_name
----------------------+-----------+------------------------------+----------------------
 end_date             |      1000 | online_page_dimension_site02 | online_page_dimension
 epoch                |      1000 | online_page_dimension_site02 | online_page_dimension
 online_page_key      |      1000 | online_page_dimension_site02 | online_page_dimension
 page_description     |      1000 | online_page_dimension_site02 | online_page_dimension
 page_number          |      1000 | online_page_dimension_site02 | online_page_dimension
 page_type            |      1000 | online_page_dimension_site02 | online_page_dimension
 start_date           |      1000 | online_page_dimension_site02 | online_page_dimension
 ad_media_name        |      1000 | promotion_dimension_site02   | promotion_dimension
 ad_type              |      1000 | promotion_dimension_site02   | promotion_dimension
```

```
 coupon_type          |      1000 | promotion_dimension_site02   | promotion_dimension
 display_provider     |      1000 | promotion_dimension_site02   | promotion_dimension
 display_type         |      1000 | promotion_dimension_site02   | promotion_dimension
 epoch                |      1000 | promotion_dimension_site02   | promotion_dimension
 price_reduction_type |      1000 | promotion_dimension_site02   | promotion_dimension
 promotion_begin_date |      1000 | promotion_dimension_site02   | promotion_dimension
 promotion_cost       |      1000 | promotion_dimension_site02   | promotion_dimension
 promotion_end_date   |      1000 | promotion_dimension_site02   | promotion_dimension
 promotion_key        |      1000 | promotion_dimension_site02   | promotion_dimension
 promotion_media_type |      1000 | promotion_dimension_site02   | promotion_dimension
 promotion_name       |      1000 | promotion_dimension_site02   | promotion_dimension
20 rows)
```

CONFIGURATION_CHANGES

Records the change history of system configuration parameters (***V_MONITOR.CONFIGURATION_PARAMETERS*** (page 996)). This information is useful for identifying:

- Who changed the configuration parameter value
- When the configuration parameter was changed
- Whether nonstandard settings were in effect in the past

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMPTZ	Time when the row was recorded.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_ID	INTEGER	Identifier of the user who changed configuration parameters.
USER_NAME	VARCHAR	Name of the user who changed configuration parameters at the time HP Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
PARAMETER	VARCHAR	Name of the changed parameter. See Configuration Parameters in the Administrator's Guide for a detailed list of supported parameters.
VALUE	VARCHAR	New value of the configuration parameter.

Permissions

Must be a superuser.

Example

```
=> SELECT * FROM configuration_changes;

        event_timestamp        | node_name |      user_id      | user_name |    session_id     |
parameter            | value
```

```
-----------------------------+----------+------------------+----------+------------------+-
------------------------------+-------
 2011-09-16 14:40:31.575335-04 | e1        | 45035996273704962 | smith     | smthl-9010:0xb2e  |
UseOnlyResilientRedistribution | 0
 2011-09-16 14:40:31.576015-04 | initiator | 45035996273704962 | smith     | smthl-9010:0xb2e  |
UseOnlyResilientRedistribution | 0
 2011-09-16 14:40:31.576106-04 | e0        | 45035996273704962 | smith     | smthl-9010:0xb2e  |
UseOnlyResilientRedistribution | 0
 2011-09-16 14:40:33.103278-04 | e0        | 45035996273704962 | smith     | smthl-9010:0xb2e  |
UseOnlyResilientRedistribution | 1
 2011-09-16 14:40:33.10332-04  | e1        | 45035996273704962 | smith     | smthl-9010:0xb2e  |
UseOnlyResilientRedistribution | 1
 2011-09-16 14:40:33.104521-04 | initiator | 45035996273704962 | smith     | smthl-9010:0xb2e  |
UseOnlyResilientRedistribution | 1
 2011-09-16 14:51:59.884112-04 | e0        | 45035996273704962 | smith     | smthl-9010:0x109d |
UseOnlyResilientRedistribution | 0
 2011-09-16 14:51:59.884519-04 | e1        | 45035996273704962 | smith     | smthl-9010:0x109d |
UseOnlyResilientRedistribution | 0
 2011-09-16 14:51:59.884695-04 | initiator | 45035996273704962 | smith     | smthl-9010:0x109d |
UseOnlyResilientRedistribution | 0
 2011-09-16 14:52:00.580423-04 | initiator | 45035996273704962 | smith     | smthl-9010:0x109d |
UseOnlyResilientRedistribution | 1
 2011-09-16 14:52:00.580673-04 | e0        | 45035996273704962 | smith     | smthl-9010:0x109d |
UseOnlyResilientRedistribution | 1
 2011-09-16 14:52:00.581464-04 | e1        | 45035996273704962 | smith     | smthl-9010:0x109d |
UseOnlyResilientRedistribution | 1
(12 rows)
```

See Also

CONFIGURATION_PARAMETERS (page 996)

Configuration Parameters in the Administrator's Guide

CONFIGURATION_PARAMETERS

Provides information about configuration parameters currently in use by the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node names on the cluster for which information is listed.
PARAMETER_NAME	VARCHAR	The name of the configurable parameter. See Configuration Parameters in the Administrator's Guide for a detailed list of supported parameters.
CURRENT_VALUE	INTEGER	The value of the current setting for the parameter.
DEFAULT_VALUE	INTEGER	The default value for the parameter.
CHANGE_UNDER_SUPPORT_GUIDANCE	BOOLEAN	A *t* (true) setting indicates parameters intended for Vertica's use only.
CHANGE_REQUIRES_RESTART	BOOLEAN	Indicates whether the configuration change requires a restart, where *t* is true and *f* is false.

DESCRIPTION	VARCHAR	A description of the parameter's purpose.

Notes

The CONFIGURATION_PARAMETERS table returns the following error in non-default locales:

```
ERROR:  ORDER BY is not supported with UNION/INTERSECT/EXCEPT in non-default
locales
HINT:  Please move the UNION to a FROM clause subquery.
```

See the ***SET LOCALE*** command for details.

Example

The following command returns all current configuration parameters in HP Vertica:

```
=> SELECT * FROM CONFIGURATION_PARAMETERS;
```

See also

Configuration Parameters in the Administrator's Guide.

CPU_USAGE

Records CPU usage history on the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
START_TIME	TIMESTAMP	Beginning of history interval.
END_TIME	TIMESTAMP	End of history interval.
AVERAGE_CPU_USAGE_PERCENT	FLOAT	Average CPU usage in percent of total CPU time (0-100) during history interval

Permissions

Must be a superuser

Example

```
=> SELECT * FROM cpu_usage;
```

```
 node_name |          start_time           |           end_time            | cpu_usage_percent
-----------+-------------------------------+-------------------------------+-------------------
 initiator | 2011-09-16 15:23:38.000703-04 | 2011-09-16 15:24:00.005645-04 |             34.49
 initiator | 2011-09-16 15:24:00.005645-04 | 2011-09-16 15:25:00.002346-04 |             12.37
 e0        | 2011-09-16 15:23:37.002957-04 | 2011-09-16 15:24:00.003022-04 |             35.35
 e0        | 2011-09-16 15:24:00.003022-04 | 2011-09-16 15:25:00.004471-04 |             12.38
 e1        | 2011-09-16 15:23:37.000871-04 | 2011-09-16 15:24:00.002474-04 |             35.37
 e1        | 2011-09-16 15:24:00.002474-04 | 2011-09-16 15:25:00.002049-04 |             12.38
(6 rows)
```

CRITICAL_HOSTS

Lists the critical hosts whose failure would cause the database to become unsafe and force a shutdown.

Column Name	Data Type	Description
HOST_NAME	VARCHAR	Name of a critical host.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> SELECT * FROM critical_hosts;
  host_name
------------
 Host1
 Host3
(2 rows)
```

CRITICAL_NODES

Lists the critical nodes whose failure would cause the database to become unsafe and force a shutdown.

Column Name	Date Type	Description
NODE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the node.
NODE_NAME	VARCHAR	The name of a critical node.

Example

```
=> SELECT * FROM v_monitor.critical_nodes;

      node_id      |     node_name
-------------------+--------------------
 45035996273704980 | v_onenode_node0001
(1 row)
```

CURRENT_SESSION

Returns information about the current active session. You can use this table to find out the current session's sessionID and get the duration of the previously-run query.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
USER_NAME	VARCHAR	The name used to log into the database or NULL if the session is internal.
CLIENT_HOSTNAME	VARCHAR	The host name and port of the TCP socket from which the client connection was made; NULL if the session is internal
CLIENT_PID	INTEGER	The process identifier of the client process that issued this connection. **Note:** Remember that the client process could be on a different machine than the server.
LOGIN_TIMESTAMP	TIMESTAMP	The date and time the user logged into the database or when the internal session was created. This column can be useful for identifying sessions that have been left open and could be idle.
SESSION_ID	VARCHAR	The identifier required to close or interrupt a session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
CLIENT_LABEL	VARCHAR	A user-specified label for the client connection that can be set when using ODBC. See `Label` in DSN Parameters in Programmer's Guide.
TRANSACTION_START	TIMESTAMP	The date/time the current transaction started or NULL if no transaction is running.
TRANSACTION_ID	VARCHAR	A string containing the hexadecimal representation of the transaction ID, if any; otherwise NULL.
TRANSACTION_DESCRIPTION	VARCHAR	A description of the current transaction.
STATEMENT_START	TIMESTAMP	The date/time the current statement started execution, or NULL if no statement is running.
STATEMENT_ID	VARCHAR	Unique numeric ID for the currently-running statement. `NULL` indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.

LAST_STATEMENT_DURATION_US	INTEGER	The duration of the last completed statement in microseconds.
CURRENT_STATEMENT	VARCHAR	The currently-running statement, if any. NULL indicates that no statement is currently being processed.
LAST_STATEMENT	VARCHAR	NULL if the user has just logged in; otherwise the currently running statement or the most recently completed statement.
EXECUTION_ENGINE_PROFILING_CONFIGURATION	VARCHAR	Returns a value that indicates whether profiling is turned on. Results are: ▪ Empty when no profiling ▪ 'Local' when profiling on for this session ▪ 'Global' when on by default for all sessions ▪ 'Local, Global' when on by default for all sessions and on for current session
QUERY_PROFILING_CONFIGURATION	VARCHAR	Returns a value that indicates whether profiling is turned on. Results are: ▪ Empty when no profiling ▪ 'Local' when profiling on for this session ▪ 'Global' when on by default for all sessions ▪ 'Local, Global' when on by default for all sessions and on for current session
SESSION_PROFILING_CONFIGURATION	VARCHAR	Returns a value that indicates whether profiling is turned on. Results are: ▪ Empty when no profiling ▪ 'Local' when profiling on for this session ▪ 'Global' when on by default for all sessions ▪ 'Local, Global' when on by default for all sessions and on for current session

Notes

- The default for profiling is ON ('1') for all sessions. Each session can turn profiling ON or OFF.
- Profiling parameters (such as `GlobalEEProfiling` in the examples below) are set in the HP Vertica configuration file (`vertica.conf`). To turn profiling off, set the parameter to '0'. To turn profiling on, set the parameter to '1'.

Examples

Query the CURRENT_SESSION table:

```
=> SELECT * FROM CURRENT_SESSION;
-[ RECORD 1 ]---------------------------+--------------------------------------
node_name                               | v_vmartdb_node01
user_name                               | release
client_hostname                         | xxx.x.x.x:xxxxx
client_pid                              | 18082
login_timestamp                         | 2010-10-07 10:10:03.114863-04
session_id                              | myhost-17956:0x1d
client_label                            |
transaction_start                       | 2010-10-07 11:52:32.43386
transaction_id                          | 45035996273727909
transaction_description                 | user release (select * from passwords;)
statement_start                         | 2010-10-07 12:30:42.444459
statement_id                            | 11
last_statement_duration_us              | 85241
current_statement                       | SELECT * FROM CURRENT_SESSION;
last_statement                          | SELECT * FROM CONFIGURATION_PARAMETERS;
execution_engine_profiling_configuration | Local
query_profiling_configuration           |
session_profiling_configuration         |
```

Request specific columns from the table:

```
=> SELECT node_name, session_id, execution_engine_profiling_configuration
   FROM CURRENT_SESSION;

 node_name |     session_id      | execution_engine_profiling_configuration
-----------+---------------------+-----------------------------------------
 site01    | myhost-17956:0x1d   | Global
(1 row)
```

The sequence of commands in this example shows the use of disabling and enabling profiling for local and global sessions.

This command disables EE profiling for query execution runs:

```
=> SELECT disable_profiling('EE');
disable_profiling
-----------------------
EE Profiling Disabled
(1 row)
```

The following command sets the `GlobalEEProfiling` configuration parameter to 0, which turns off profiling:

```
=> SELECT set_config_parameter('GlobalEEProfiling', '0');
set_config_parameter
----------------------------
Parameter set successfully
(1 row)
```

The following command tells you whether profiling is set to 'Local' or 'Global' or none:

```
=> SELECT execution_engine_profiling_configuration FROM CURRENT_SESSION;
ee_profiling_config
--------------------
```

```
(1 row)
```

Note: The result set is empty because profiling was turned off in the preceding example.

This command now enables EE profiling for query execution runs:

```
=> SELECT enable_profiling('EE');
enable_profiling
----------------------
EE Profiling Enabled
(1 row)
```

Now when you run a select on the CURRENT_SESSION table, you can see profiling is ON for the local session:

```
=> SELECT execution_engine_profiling_configuration FROM CURRENT_SESSION;
ee_profiling_config
--------------------
Local
(1 row)
```

Now turn profiling on for all sessions by setting the GlobalEEProfiling configuration parameter to 1:

```
=> SELECT set_config_parameter('GlobalEEProfiling', '1');
set_config_parameter
---------------------------
Parameter set successfully
(1 row)
```

Now when you run a select on the CURRENT_SESSION table, you can see profiling is ON for the local sessions, as well as for all sessions:

```
=> SELECT execution_engine_profiling_configuration FROM CURRENT_SESSION;
ee_profiling_config
--------------------
Local, Global
(1 row)
```

See Also

CLOSE_SESSION (page 458), ***CLOSE_ALL_SESSIONS*** (page 461), ***EXECUTION_ENGINE_PROFILES*** (page 1021), ***QUERY_PROFILES*** (page 1071), ***SESSION_PROFILES*** (page 1093), and ***SESSIONS*** (page 1095)

Managing Sessions and Configuration Parameters in the Administrator's Guide

DATA_COLLECTOR

Shows the Data Collector components, their current retention policies, and statistics about how much data is retained and how much has been discarded for various reasons. The DATA_COLLECTOR system table also calculates approximate collection rate, to aid in sizing calculations.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name on which information is retained.
COMPONENT	VARCHAR	The name of the component and its policy.
TABLE_NAME	VARCHAR	The data collector (dc) table name for which information is listed.
DESCRIPTION	VARCHAR	A short description about the component.
IN_DB_LOG	BOOLEAN	Denotes if monitoring information is retained in the `db.log` file.
IN_VERTICA_LOG	BOOLEAN	Denotes if monitoring information is retained in the `vertica.log` file.
MEMORY_BUFFER_SIZE_KB	INTEGER	The size of the memory buffer in kilobytes.
DISK_SIZE_KB	INTEGER	The on-disk size of the table in kilobytes.
RECORD_TOO_BIG_ERRORS	INTEGER	A number that increments by one each time an error is thrown because data did not fit in memory (based on the data collector retention policy).
LOST_BUFFERS	INTEGER	The number of buffers lost.
LOST_RECORDS	INTEGER	The number of records lost.
RETIRED_FILES	INTEGER	The number of retired files.
RETIRED_RECORDS	INTEGER	The number of retired records.
CURRENT_MEMORY_RECORDS	INTEGER	The current number of rows in memory.
CURRENT_DISK_RECORDS	INTEGER	The current number of rows stored on disk.
CURRENT_MEMORY_BYTES	INTEGER	Total current memory used in kilobytes.
CURRENT_DISK_BYTES	INTEGER	Total current disk space used in kilobytes.
FIRST_TIME	TIMESTAMP	Timestamp of the first record.
LAST_TIME	TIMESTAMP	Timestamp of the last record
KB_PER_DAY	INTEGER	Total kilobytes used per day.

Notes

- Data Collector is on by default, but you can turn it off if you need to. See Enabling and Disabling Data Collector in the Administrator's Guide.
- You can configure monitoring information retention policies. See ***Data Collector Functions*** (page 560) in this guide and Configuring Data Retention Policies in the Administrator's Guide.
- Query the DATA_COLLECTOR system table for a list of all current component names; for example:

```
=> SELECT DISTINCT component, description FROM data_collector ORDER BY
1 ASC;
```

Examples

The following command, which queries all columns in the DATA_COLLECTOR system table, is truncated for brevity:

```
=> \x
=> SELECT * FROM data_collector;

-[ RECORD 1 ]----------+----------------------------------------
node_name              | v_vmartdb_node0003
component              | AllocationPoolStatistics
table_name             | dc_allocation_pool_statistics
description            | Information about global memory pools, ...
in_db_log              | f
in_vertica_log         | f
memory_buffer_size_kb  | 64
disk_size_kb           | 256
record_too_big_errors  | 0
lost_buffers           | 0
lost_records           | 0
retired_files          | 150
retired_records        | 66318
current_memory_records | 0
current_disk_records   | 1582
current_memory_bytes   | 0
current_disk_bytes     | 234536
first_time             | 2011-05-26 13:19:01.006121-04
last_time              | 2011-05-26 13:25:36.004994-04
kb_per_day             | 50014.1761771333
-[ RECORD 2 ]----------+----------------------------------------
node_name              | v_vmartdb_node0003
component              | AllocationPoolStatisticsBySecond
table_name             | dc_allocation_pool_statistics_by_second
description            | Information about global memory pools, ...
in_db_log              | f
in_vertica_log         | f
memory_buffer_size_kb  | 64
disk_size_kb           | 256
record_too_big_errors  | 0
lost_buffers           | 0
lost_records           | 0
retired_files          | 648
retired_records        | 66180
current_memory_records | 0
current_disk_records   | 349
current_memory_bytes   | 0
current_disk_bytes     | 222742
first_time             | 2011-05-26 13:24:09.002271-04
last_time              | 2011-05-26 13:25:36.005041-04
kb_per_day             | 214367.571703045
-[ RECORD 3 ]----------+----------------------------------------
...
```

The following command returns all component names and their descriptions. This is a useful query if you want to change the retention policy for a particular component and don't remember its name:

```
=> SELECT DISTINCT component, description FROM data_collector ORDER BY 1 ASC;

            component             |                          description
----------------------------------+-------------------------------------------------------------
-----
 AllocationPoolStatistics         | Information about global memory pools ...
 AllocationPoolStatisticsByDay    | Information about global memory pools, ... (historical, by day)
 AllocationPoolStatisticsByHour   | Information about global memory pools, ... (historical, by hour)
 AllocationPoolStatisticsByMinute | Information about global memory pools, ... (historical, by minute)
```

```
 AllocationPoolStatisticsBySecond | Information about global memory pools, ... (historical, by second)
 AnalyzeStatistics                | History of statistics collection
 Backups                          | Monitoring successful backups
 CatalogInfo                      | Catalog statistics and history
 CatalogInfoByDay                 | Catalog statistics and history (historical, by day)
 CatalogInfoByHour                | Catalog statistics and history (historical, by hour)
 CatalogInfoByMinute              | Catalog statistics and history (historical, by minute)
 CatalogInfoBySecond              | Catalog statistics and history (historical, by second)
 ClientServerMessages             | Client-Server Messages (Front End to Back End Protocol) sent
 ConfigurationChanges             | Changes to configuration parameters (vertica.conf)
 CpuAggregate                     | Aggregate CPU information
 CpuAggregateByDay                | Aggregate CPU information (historical, by day)
 CpuAggregateByHour               | Aggregate CPU information (historical, by hour)
 CpuAggregateByMinute             | Aggregate CPU information (historical, by minute)
 CpuAggregateBySecond             | Aggregate CPU information (historical, by second)
 CpuInfo                          | CPU information
 CpuInfoByDay                     | CPU information (historical, by day)
 CpuInfoByHour                    | CPU information (historical, by hour)
 CpuInfoByMinute                  | CPU information (historical, by minute)
 CpuInfoBySecond                  | CPU information (historical, by second)
 DeploymentsCompleted             | History of designs deployed
 DesignsCompleted                 | History of designs executed
 DiskResourceRejections           | Disk Resource Rejection Records
 Errors                           | History of all errors+warnings encountered
 ExecutionEngineEvents            | History of important events during local planning and execution
 ExecutionEngineProfiles          | History of EE profiles
 IoInfo                           | Information about device IOs
 IoInfoByDay                      | Information about device IOs (historical, by day)
 IoInfoByHour                     | Information about device IOs (historical, by hour)
 IoInfoByMinute                   | Information about device IOs (historical, by minute)
 IoInfoBySecond                   | Information about device IOs (historical, by second)
 LockAttempts                     | History of lock attempts (resolved requests)
 LockReleases                     | History of lock releases
 LockRequests                     | History of lock requests
 LoginFailures                    | Failed login attempts
 MemoryInfo                       | Information about node memory allocation, at the OS level
 MemoryInfoByDay                  | Information about node memory allocation, ... (historical, by day)
 MemoryInfoByHour                 | Information about node memory allocation, ... (historical, by
hour)
 MemoryInfoByMinute               | Information about node memory allocation, ... (historical, by
minute)
 MemoryInfoBySecond               | Information about node memory allocation, ... (historical, by
second)
 MonitoringEventsCleared          | Monitoring events cleared
 MonitoringEventsPosted           | Monitoring events posted
 NetworkInfo                      | Network interface information and statistics
 NetworkInfoByDay                 | Network interface information and statistics (historical, by day)
 NetworkInfoByHour                | Network interface information and statistics (historical, by
hour)
 NetworkInfoByMinute              | Network interface information and statistics (historical, by
minute)
 NetworkInfoBySecond              | Network interface information and statistics (historical, by
second)
 NodeState                        | History of all node state changes
 OptimizerEvents                  | History of important events during optimizer planning
 OptimizerStats                   | History of optimizer runtime statistics
 ProcessInfo                      | Information about vertica process memory, handles and system
limits
 ProcessInfoByDay                 | Information about vertica process memory, ... (historical, by day)
 ProcessInfoByHour                | Information about vertica process memory, ... (historical, by
hour)
 ProcessInfoByMinute              | Information about vertica process memory, ... (historical, by
minute)
 ProcessInfoBySecond              | Information about vertica process memory, ... (historical, by
second)
 ProjectionRecoveries             | Monitoring completed projection recoveries
 ProjectionRefreshesCompleted     | History of refreshed projections
 ProjectionsUsed                  | Projections used in each SQL request issued
```

```
 RebalancedSegments              | History of all segments rebalanced (EC)
 RequestsCompleted               | History of all SQL requests completed
 RequestsIssued                  | History of all SQL requests issued
 RequestsRetried                 | History of all SQL requests issued that were retried
 ResourceAcquisitions            | History of all resource acquisitions
 ResourceRejections              | Resource Rejection Records
 ResourceReleases                | History of all resource acquisition releases
 RosesCreated                    | History of all ROS and DVROS created
 RosesDestroyed                  | History of all ROS destroyed
 SessionEnds                     | Sessions ended
 SessionStarts                   | Sessions started
 Signals                         | History of process signals received
 Startups                        | History of all node startup events
 StorageInfo                     | Storage information (Used and Free space)
 StorageInfoByDay                | Storage information (Used and Free space) (historical, by day)
 StorageInfoByHour               | Storage information (Used and Free space) (historical, by hour)
 StorageInfoByMinute             | Storage information (Used and Free space) (historical, by minute)
 StorageInfoBySecond             | Storage information (Used and Free space) (historical, by second)
 StorageLayerStatistics          | Statistics and history of storage and caching layer
 StorageLayerStatisticsByDay     | Statistics and history of storage and caching layer (historical,
by day)
 StorageLayerStatisticsByHour    | Statistics and history of storage and caching layer (historical,
by hour)
 StorageLayerStatisticsByMinute  | Statistics and history of storage and caching layer (historical,
by minute)
 StorageLayerStatisticsBySecond  | Statistics and history of storage and caching layer (historical,
by second)
 Test                            | For data collector infrastructure testing
 TransactionEnds                 | History of end transactions (commit or rollback)
 TransactionStarts               | History of begin transactions
 TuningAnalysis                  | Tuning analysis history in Workload Analyzer
 TuningRecommendations           | Tuning Recommendations in Workload Analyzer
 TupleMoverEvents                | History of Tuple Mover activities
 Upgrades                        | Monitoring catalog upgrades
 UserAudits                      | History of user audits
(93 rows)
```

Related Topics

Data Collector Functions (page 560)

Retaining Monitoring Information and How HP Vertica Calculates Database Size in the Administrator's Guide

DATABASE_BACKUPS

Lists historical information for each backup that successfully completed after running the `vbr.py` utility. This information is useful for determining whether to create a new backup before you advance the AHM. Because this system table displays historical information, its contents do not always reflect the current state of a backup repository. For example, if you delete a backup from a repository, the DATABASE_BACKUPS system table continues to display information about it.

To monitor snapshot information while `vbr.py` is running, query the ***DATABASE_SNAPSHOTS*** (page 1008) system table. To list existing backups, run `vbr.py` as described in Viewing Backups in the Administrator's Guide.

Column Name	Data Type	Description
BACKUP_TIMESTAMP	TIMESTAMP	The timestamp of the backup.
NODE_NAME	VARCHAR	The name local or remote backup host, as specified in the `backupHost` parameter of the `vbr.py` configuration file.
SNAPSHOT_NAME	VARCHAR	The name of the backup, as specified in the `snapshotName` parameter of the `vbr.py` configuration file.
BACKUP_EPOCH	INTEGER	The database epoch at which the backup was saved.
NODE_COUNT	INTEGER	The number of nodes backed up in the completed backup, and as listed in the `[Mapping`*n*`]` sections of the configuration file.
OBJECTS	VARCHAR	The name of the object(s) contained in an object-level backup. This column is empty if the record is for a full cluster backup.

Permissions

Must be a superuser.

Example

```
VMart=> select * from v_monitor.database_backups;
-[ RECORD 1 ]----+-----------------------------
backup_timestamp | 2013-05-10 14:41:12.673381-04
node_name        | v_vmart_node0003
snapshot_name    | schemabak
backup_epoch     | 174
node_count       | 3
objects          | public, store, online_sales
-[ RECORD 2 ]----+-----------------------------
backup_timestamp | 2013-05-13 11:17:30.913176-04
node_name        | v_vmart_node0003
snapshot_name    | kantibak
backup_epoch     | 175
node_count       | 3
objects          |
.
.
.
-[ RECORD 15 ]---+-----------------------------
backup_timestamp | 2013-05-16 07:20:18.585076-04
node_name        | v_vmart_node0003
snapshot_name    | table2bak
backup_epoch     | 180
node_count       | 3
objects          | test2
-[ RECORD 16 ]---+-----------------------------
```

```
backup_timestamp | 2013-05-28 14:06:03.027673-04
node_name        | v_vmart_node0003
snapshot_name    | kantibak
backup_epoch     | 182
node_count       | 3
objects          |
```

See Also

DATABASE_SNAPSHOTS (page 1008)

DATABASE_CONNECTIONS

Lists the connections that have been established to other databases for importing and exporting data. See Moving Data Between HP Vertica Databases in the Administrator's Guide.

Column Name	Data Type	Description
DATABASE	VARCHAR	The name of the connected database
USERNAME	VARCHAR	The username used to create the connection
HOST	VARCHAR	The host name used to create the connection
PORT	VARCHAR	The port number used to create the connection
ISVALID	BOOLEAN	Whether the connection is still open and usable or not

Example

```
=> CONNECT TO VERTICA vmart USER dbadmin PASSWORD '' ON '10.10.20.150',5433;
CONNECT
=> SELECT * FROM DATABASE_CONNECTIONS;
 database | username |     host      | port | isvalid
----------+----------+---------------+------+---------
 vmart    | dbadmin  | 10.10.20.150 | 5433 | t
(1 row)
```

DATABASE_SNAPSHOTS

Displays information about database snapshots. A snapshot is a special temporary image of the database, which `vbr.py` uses internally as part of creating a full or object-level backup. When `vbr.py` completes a backup, it deletes the associated snapshot and its entry in the `database_snapshots` system table.

To see historical data about successfully created backups, query the ***DATABASE_BACKUPS*** (page 1006) system table. To see existing backups, run `vbr.py` as described in Viewing and Deleting Backups in the Administrator's Guide.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node where the snapshot will be stored, as specified in the `backupHost` parameter of the `vbr.py` configuration file.
SNAPSHOT_NAME	VARCHAR	The name of the snapshot you specify in the `vbr.py` configuration file `snapshotName` parameter.
IS_DURABLE_SNAPSHOT	BOOLEAN	Indicates the snapshot durability. This value is always set to `t` (`true`).
TOTAL_SIZE_BYTES	INTEGER	The total size (in bytes) of the data being backed up. This value differs from the data storage size (`STORAGE_COST_BYTES`), described next.
STORAGE_COST_BYTES	INTEGER	The amount of disk space used for a snapshot, and which will be freed when `vbr.py` deletes it. This value can change over time. For example, the `storage_cost_bytes` increases when storage is discarded from the database.
ACQUISITION_TIMESTAMP	TIMESTAMP	The recorded time at which `vbr.py` will create the snapshot.

Example

To monitor snapshot information during vbr.py execution:

```
=> SELECT * FROM database_snapshots;
```

```
vmartdb=> SELECT * FROM database_snapshots;
     node_name      | snapshot_name | is_durable_snapshot | total_size_bytes | storage_cost_bytes |
acquisition_timestamp
--------------------+---------------+---------------------+------------------+--------------------
-+-----------------------
 v_vmartdb_node0001 | mysnapshot    | t                   |        116108615 |            5054638 |
2010-10-07 12:39:22-04
 v_vmartdb_node0002 | mysnapshot    | t                   |        116385001 |            5066175 |
2010-10-07 12:39:35-04
 v_vmartdb_node0003 | mysnapshot    | t                   |        116379703 |            5054692 |
2010-10-07 12:38:00-04
 v_vmartdb_node0004 | mysnapshot    | t                   |        116354638 |            5043155 |
2010-10-07 12:36:10-04
(4 rows)
```

See Also

DATABASE_BACKUPS (page 1006)

DELETE_VECTORS

Holds information on deleted rows to speed up the delete process.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node storing the deleted rows.
SCHEMA_NAME	VARCHAR	The name of the schema where the deleted rows are located.
PROJECTION_NAME	VARCHAR	The name of the projection where the deleted rows are located.
STORAGE_TYPE	VARCHAR	The type of storage containing the delete vector (WOS or ROS).
DV_OID	INTEGER	The unique numeric ID (OID) that identifies this delete vector.
STORAGE_OID	INTEGER	The unique numeric ID (OID) that identifies the storage container that holds the delete vector.
DELETED_ROW_COUNT	INTEGER	The number of rows deleted.
USED_BYTES	INTEGER	The number of bytes used to store the deletion.
START_EPOCH	INTEGER	The start epoch of the data in the delete vector.
END_EPOCH	INTEGER	The end epoch of the data in the delete vector.

DEPLOY_STATUS

Records the history of the Database Designer designs that have been deployed and their deployment steps.

Column Name	Data Type	Description
EVENT_TIME	TIMESTAMP	Time when the row recorded the event.
USER_NAME	VARCHAR	Name of the user who deployed a design at the time HP Vertica recorded the session.
DEPLOY_NAME	VARCHAR	Name the deployment, same as the user-specified design name.
DEPLOY_STEP	VARCHAR	Steps in the design deployment.
DEPLOY_STEP_STATUS	VARCHAR	Textual status description of the current step in the deploy process.

DEPLOY_STEP_COMPLETE_PERCENT	FLOAT	Progress of current step in percentage (0–100).
DEPLOY_COMPLETE_PERCENT	FLOAT	Progress of overall deployment in percentage (0–100).
ERROR_MESSAGE	VARCHAR	Error or warning message during deployment.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

The following example shows the content of the DEPLOY_STATUS for a Database Designer deployment executed by the DBADMIN user:

```
=> select * from v_monitor.deploy_status;
     event_time | user_name | deploy_name | deploy_step | deploy_step_status |
deploy_step_complete_percent | deploy_complete_percent | error_message
--------------------+-----------+-------------+-----------------------------------+-----------
--------+-------------------------+-----------------------+--------------
 2012-02-14 10:33:14 | dbadmin | design1 | create_deployment | success | | | N/A
 2012-02-14 10:33:14 | dbadmin | design1 | populate_deployment | started | | | N/A
 2012-02-14 10:33:21 | dbadmin | design1 | populate_deployment | success | | | N/A
 2012-02-14 10:33:21 | dbadmin | design1 | deployment_script | started | | | N/A
 2012-02-14 10:33:23 | dbadmin | design1 | run_deployment | success | | | N/A
 2012-02-14 10:33:23 | dbadmin | design1 | deployment_script | success | | | N/A
 2012-02-14 10:33:24 | dbadmin | design1 | execute_deployment | started | | | N/A
 2012-02-14 10:33:26 | dbadmin | design1 | add: Dim_DBD_3_rep_ctx_design1 | complete | 100 | 0 | N/A
 2012-02-14 10:33:27 | dbadmin | design1 | drop: Dim_b0 | complete | 100 | 0 | N/A
 2012-02-14 10:33:27 | dbadmin | design1 | drop: Dim_b1 | complete | 100 | 0 | N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_11_seg_ctx_design1 | complete | 100 | 11.11
| N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_12_seg_1_ctx_design1 | complete | 100 | 22.22
| N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_1_seg_ctx_design1 | complete | 100 | 33.33
| N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_4_rep_ctx_design1 | complete | 100 | 37.04
| N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_5_seg_ctx_design1 | complete | 100 | 38.89
| N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_6_seg_ctx_design1 | complete | 100 | 40.74
| N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_7_seg_ctx_design1 | complete | 100 | 51.85
| N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_8_seg_ctx_design1 | complete | 100 | 53.7
| N/A
 2012-02-14 10:33:43 | dbadmin | design1 | add: Fact1_DBD_9_seg_ctx_design1 | complete | 100 | 55.56
| N/A
 2012-02-14 10:33:44 | dbadmin | design1 | drop: Fact1_b0 | complete | 100 | 55.56 | N/A
 2012-02-14 10:33:44 | dbadmin | design1 | drop: Fact1_b1 | complete | 100 | 55.56 | N/A
 2012-02-14 10:33:56 | dbadmin | design1 | add: Fact2_DBD_10_seg_ctx_design1 | complete | 100 | 77.78
| N/A
 2012-02-14 10:33:56 | dbadmin | design1 | add: Fact2_DBD_2_seg_ctx_design1 | complete | 100 | 100
| N/A
 2012-02-14 10:33:56 | dbadmin | design1 | drop: Fact2_b0 | complete | 100 | 100 | N/A
 2012-02-14 10:33:56 | dbadmin | design1 | drop: Fact2_b1 | complete | 100 | 100 | N/A
 2012-02-14 10:33:57 | dbadmin | design1 | run_deployment | success | | | N/A
 2012-02-14 10:33:57 | dbadmin | design1 | execute_deployment | success | | | N/A
 2012-02-14 10:33:57 | dbadmin | design1 | deployment | completed | | | N/A
...
```

DESIGN_STATUS

Records the progress of a running Database Designer design or history of the last Database Designer design executed by the current user.

Column Name	Data Type	Description
EVENT_TIME	TIMESTAMP	Time when the row recorded the event.
USER_NAME	VARCHAR	Name of the user who ran a design at the time HP Vertica recorded the session.
DESIGN_NAME	VARCHAR	Name of the user-specified design.
DESIGN_PHASE	VARCHAR	Phase of the design.
PHASE_STEP	VARCHAR	Substep in each design phase
PHASE_STEP_COMPLETE_PERCENT	FLOAT	Progress of current substep in percentage (0–100).
PHASE_COMPLETE_PERCENT	FLOAT	Progress of current design phase in percentage (0–100).

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

The following example shows the content of the DESIGN_STATUS table of a complete Database Designer run:

```
=> select * from v_monitor.design_status;
     event_time | user_name | design_name | design_phase | phase_step | phase_step_complete_percent
| phase_complete_percent
--------------------+-----------+-------------+-----------------------------------------------+
-----------------------------------------+----------------------------+--------------------
---
 2012-02-14 10:31:20 | dbadmin | design1 | Design started | ================ | |
 2012-02-14 10:31:21 | dbadmin | design1 | Design in progress: Analyze statistics phase |
================ | |
 2012-02-14 10:31:21 | dbadmin | design1 | Analyzing data statistics | public.Fact1 | 100 | 33.33
 2012-02-14 10:31:22 | dbadmin | design1 | Analyzing data statistics | public.Fact2 | 100 | 66.67
 2012-02-14 10:31:24 | dbadmin | design1 | Analyzing data statistics | public.Dim | 100 | 100
 2012-02-14 10:31:25 | dbadmin | design1 | Design in progress: Query optimization phase |
================ | |
 2012-02-14 10:31:25 | dbadmin | design1 | Optimizing query performance | iteration 1: Complete | 100
| 37.5
 2012-02-14 10:31:31 | dbadmin | design1 | Optimizing query performance | iteration 2: Complete | 100
| 62.5
 2012-02-14 10:31:36 | dbadmin | design1 | Optimizing query performance | iteration 3: Complete | 100
| 75
 2012-02-14 10:31:39 | dbadmin | design1 | Optimizing query performance | iteration 4: Complete | 100
| 87.5
 2012-02-14 10:31:41 | dbadmin | design1 | Optimizing query performance | iteration 5: Complete | 100
| 87.5
 2012-02-14 10:31:42 | dbadmin | design1 | Design in progress: Storage optimization phase |
```

```
================ | |
 2012-02-14 10:31:44 | dbadmin | design1 | Optimizing storage footprint | Fact1_DBD_4_rep_ctx_design1
| 100 | 4.17
 2012-02-14 10:31:44 | dbadmin | design1 | Optimizing storage footprint | Fact1_DBD_1_seg_ctx_design1
| 100 | 16.67
 2012-02-14 10:32:04 | dbadmin | design1 | Optimizing storage footprint | Fact1_DBD_7_seg_ctx_design1
| 100 | 29.17
 2012-02-14 10:32:04 | dbadmin | design1 | Optimizing storage footprint | Fact1_DBD_5_seg_ctx_design1
| 100 | 31.25
 2012-02-14 10:32:05 | dbadmin | design1 | Optimizing storage footprint | Fact1_DBD_8_seg_ctx_design1
| 100 | 33.33
 2012-02-14 10:32:05 | dbadmin | design1 | Optimizing storage footprint | Fact1_DBD_6_seg_ctx_design1
| 100 | 35.42
 2012-02-14 10:32:05 | dbadmin | design1 | Optimizing storage footprint | Fact1_DBD_9_seg_ctx_design1
| 100 | 37.5
 2012-02-14 10:32:05 | dbadmin | design1 | Optimizing storage footprint | Fact2_DBD_2_seg_ctx_design1
| 100 | 62.5
 2012-02-14 10:32:39 | dbadmin | design1 | Optimizing storage footprint | Fact2_DBD_10_seg_ctx_design1
| 100 | 87.5
 2012-02-14 10:32:39 | dbadmin | design1 | Optimizing storage footprint | Dim_DBD_3_rep_ctx_design1
| 100 | 87.5
 2012-02-14 10:32:41 | dbadmin | design1 | Optimizing storage footprint |
Fact1_DBD_45035996273720238_seg_ctx_design1 | 100 | 100
 2012-02-14 10:33:12 | dbadmin | design1 | Design completed successfully | ================ | |
(24 rows)
```

DISK_RESOURCE_REJECTIONS

Returns requests for resources that are rejected due to disk space shortages.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
RESOURCE_TYPE	VARCHAR	The resource request requester (example: Temp files).
REJECTED_REASON	VARCHAR	One of 'Insufficient disk space' or 'Failed volume'.
REJECTED_COUNT	INTEGER	Number of times this REJECTED_REASON has been given for this RESOURCE_TYPE.
FIRST_REJECTED_TIMESTAMP	TIMESTAMP	The time of the first rejection for this REJECTED_REASON and RESOURCE_TYPE.
LAST_REJECTED_TIMESTAMP	TIMESTAMP	The time of the most recent rejection for this REJECTED_REASON and RESOURCE_TYPE.
LAST_REJECTED_VALUE	INTEGER	The value of the most recent rejection for this REJECTED_REASON and RESOURCE_TYPE.

Notes

Output is aggregated by both RESOURCE_TYPE and REJECTED_REASON to provide more comprehensive information.

Example

```
=>\pset expanded
```

```
Expanded display on.
=> SELECT * FROM disk_resource_rejections;
-[ RECORD 1 ]------------+--------------------------
node_name                | e0
resource_type            | Table Data
rejected_reason          | Insufficient disk space
rejected_count           | 2
first_rejected_timestamp | 2009-10-16 15:55:16.336246
last_rejected_timestamp  | 2009-10-16 15:55:16.336391
last_rejected_value      | 1048576
-[ RECORD 2 ]------------+--------------------------
node_name                | e1
resource_type            | Table Data
rejected_reason          | Insufficient disk space
rejected_count           | 2
first_rejected_timestamp | 2009-10-16 15:55:16.37908
last_rejected_timestamp  | 2009-10-16 15:55:16.379207
last_rejected_value      | 1048576
```

See Also

RESOURCE_REJECTIONS (page 1089)

CLEAR_RESOURCE_REJECTIONS (page 456)

Managing Workloads and Managing System Resource Usage in the Administrator's Guide

DISK_STORAGE

Returns the amount of disk storage used by the database on each node.

Column Name	Date Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
STORAGE_PATH	VARCHAR	The path where the storage location is mounted.
STORAGE_USAGE	VARCHAR	The type of information stored in the location: ▪ DATA: Only data is stored in the location. ▪ TEMP: Only temporary files that are created during loads or queries are stored in the location. ▪ DATA,TEMP: Both types of files are stored in the location. ▪ USER: The storage location can be used by non-dbadmin users, who are granted access to the storage location ▪ CATALOG: The area is used for the HP Vertica catalog. This usage is set internally and cannot be removed or

		changed.
RANK	INTEGER	The rank assigned to the storage location based on its performance. Ranks are used to create a storage locations on which projections, columns, and partitions are stored on different disks based on predicted or measured access patterns. See Creating and Configuring Storage Locations in the Administrator's Guide.
THROUGHPUT	INTEGER	The measure of a storage location's performance in MB/sec. 1/throughput is the time taken to read 1MB of data.
LATENCY	INTEGER	The measure of a storage location's performance in seeks/sec. 1/latency is the time taken to seek to the data.
STORAGE_STATUS	VARCHAR	The status of the storage location: active or retired.
DISK_BLOCK_SIZE_BYTES	INTEGER	The block size of the disk in bytes.
DISK_SPACE_USED_BLOCKS	INTEGER	The number of disk blocks in use.
DISK_SPACE_USED_MB	INTEGER	The number of megabytes of disk storage in use.
DISK_SPACE_FREE_BLOCKS	INTEGER	The number of free disk blocks available.
DISK_SPACE_FREE_MB	INTEGER	The number of megabytes of free storage available.
DISK_SPACE_FREE_PERCENT	INTEGER	The percentage of free disk space remaining.

Notes

- All values returned are in the context of the operating system's filesystems and are not specific to HP Vertica-specific space.
- The storage usage annotation called CATALOG indicates the location is used to store the catalog. However, CATALOG location can only be specified when creating a new database and no new locations can be added as CATALOG locations using ***ADD_LOCATION*** (page 426). Existing CATALOG annotations cannot be removed.
- A storage location's performance is measured in throughput in MB/sec and latency in seeks/sec. These two values are converted to single number(Speed) with the following formula:

  ReadTime (time to read 1MB) = 1/throughput + 1 / latency
  - 1/throughput is the time taken to read 1MB of data
  - 1/latency is the time taken to seek to the data.
  - ReadTime is the time taken to read 1MB of data.
- A disk is faster than another disk if its ReadTime is less.
- There can be multiple storage locations per node, and these locations can be on different disks with different free/used space, block size, etc. This information is useful in letting you know where the data files reside.

Example

Query the DISK_STORAGE table:

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM DISK_STORAGE;

-[ RECORD 1 ]-----------+----------------------------
current_timestamp       | 2009-08-11 14:48:35.932541
node_name               | site01
storage_path            | /mydb/node01_catalog/Catalog
storage_usage           | CATALOG
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
disk_space_used_blocks  | 34708721
disk_space_used_mb      | 135581
disk_space_free_blocks  | 178816678
disk_space_free_mb      | 698502
disk_space_free_percent | 83%
-[ RECORD 2 ]-----------+----------------------------
current_timestamp       | 2009-08-11 14:48:53.884255
node_name               | site01
storage_path            | /mydb/node01_data
storage_usage           | DATA,TEMP
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
disk_space_used_blocks  | 34708721
disk_space_used_mb      | 135581
disk_space_free_blocks  | 178816678
disk_space_free_mb      | 698502
disk_space_free_percent | 83%
-[ RECORD 3 ]-----------+----------------------------
current_timestamp       | 2009-08-11 14:49:08.299012
node_name               | site02
storage_path            | /mydb/node02_catalog/Catalog
storage_usage           | CATALOG
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
disk_space_used_blocks  | 19968349
disk_space_used_mb      | 78001
disk_space_free_blocks  | 193557050
disk_space_free_mb      | 756082
disk_space_free_percent | 90%
-[ RECORD 4 ]-----------+----------------------------
```

```
current_timestamp       | 2009-08-11 14:49:22.696772
node_name               | site02
storage_path            | /mydb/node02_data
storage_usage           | DATA,TEMP
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
disk_space_used_blocks  | 19968349
disk_space_used_mb      | 78001
disk_space_free_blocks  | 193557050
disk_space_free_mb      | 756082
disk_space_free_percent | 90%
-[ RECORD 5 ]-----------+----------------------------
current_timestamp       | 2009-08-11 14:50:03.960157
node_name               | site03
storage_path            | /mydb/node03_catalog/Catalog
storage_usage           | CATALOG
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
disk_space_used_blocks  | 19902595
disk_space_used_mb      | 77744
disk_space_free_blocks  | 193622804
disk_space_free_mb      | 756339
disk_space_free_percent | 90%
-[ RECORD 6 ]-----------+----------------------------
current_timestamp       | 2009-08-11 14:50:27.415735
node_name               | site03
storage_path            | /mydb/node03_data
storage_usage           | DATA,TEMP
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
disk_space_used_blocks  | 19902595
disk_space_used_mb      | 77744
disk_space_free_blocks  | 193622804
disk_space_free_mb      | 756339
disk_space_free_percent | 90%
-[ RECORD 7 ]-----------+----------------------------
current_timestamp       | 2009-08-11 14:50:39.398879
node_name               | site04
storage_path            | /mydb/node04_catalog/Catalog
storage_usage           | CATALOG
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
```

```
disk_space_used_blocks  | 19972309
disk_space_used_mb      | 78017
disk_space_free_blocks  | 193553090
disk_space_free_mb      | 756066
disk_space_free_percent | 90%
-[ RECORD 8 ]-----------+-----------------------------
current_timestamp       | 2009-08-11 14:50:57.879302
node_name               | site04
storage_path            | /mydb/node04_data
storage_usage           | DATA,TEMP
rank                    | 0
throughput              | 0
latency                 | 0
storage_status          | Active
disk_block_size_bytes   | 4096
disk_space_used_blocks  | 19972309
disk_space_used_mb      | 78017
disk_space_free_blocks  | 193553090
disk_space_free_mb      | 756066
disk_space_free_percent | 90%
```

Request only specific columns from the table:

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

 node_name |         storage_path         | storage_status | disk_space_free_percent
-----------+------------------------------+----------------+-------------------------
 site01    | /mydb/node01_catalog/Catalog | Active         | 83%
 site01    | /mydb/node01_data            | Active         | 83%
 site02    | /mydb/node02_catalog/Catalog | Active         | 90%
 site02    | /mydb/node02_data            | Active         | 90%
 site03    | /mydb/node03_catalog/Catalog | Active         | 90%
 site03    | /mydb/node03_data            | Active         | 90%
 site04    | /mydb/node04_catalog/Catalog | Active         | 90%
 site04    | /mydb/node04_data            | Active         | 90%
(8 rows)
```

ERROR_MESSAGES

Lists system error messages and warnings HP Vertica encounters while processing queries.

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMPTZ	Time when the row recorded the event.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_ID	INTEGER	Identifier of the user who received the error message.
USER_NAME	VARCHAR	Name of the user who received the error message at the time HP Vertica recorded the session.

SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.
ERROR_LEVEL	VARCHAR	Severity of the error, can be one of: ▪ LOG ▪ INFO ▪ NOTICE ▪ WARNING ▪ ERROR ▪ ROLLBACK ▪ INTERNAL ▪ FATAL ▪ PANIC
ERROR_CODE	INTEGER	Error code that HP Vertica reports.
MESSAGE	VARCHAR	Textual output of the error message.
DETAIL	VARCHAR	Additional information about the error message, in greater detail.
HINT	VARCHAR	Actionable hint about the error. For example: `HINT: Set the locale in this session to en_US@collation=binary using the command "\locale en_US@collation=binary"`

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Notes

Some errors occur when no transaction is in progress, so the transaction identifier or statement identifier columns could return NULL.

Example

```
=> SELECT event_timestamp, statement_id, error_level, message FROM
error_messages;
```

```
        event_timestamp        | statement_id | error_level |                    message
-------------------------------+--------------+-------------+-----------------------------------
------------
 2012-03-30 17:52:12.326927-04 |           -1 | FATAL       | Client canceled session
raster-s1-10295:0x225b
 2012-03-30 17:42:12.249989-04 |           20 | ERROR       | Syntax error at or near "name"
 2012-03-30 17:42:07.018451-04 |           -1 | FATAL       | Client canceled session
raster-s1-10295:0x225b
 2012-03-30 17:40:22.665446-04 |           17 | ERROR       | Execution canceled by operator
 2012-03-30 16:32:38.784384-04 |           11 | ERROR       | Relation "systm_tables" does not exist
 2012-03-30 16:26:51.072655-04 |           98 | ERROR       | Relation "rebalance_projections" does
not exist
 2012-03-30 14:31:09.953241-04 |           -1 | FATAL       | Client canceled session
raster-s1-10295:0x70
 2012-03-30 14:26:06.655026-04 |           42 | ERROR       | Syntax error at or near "constraint"
 2012-03-30 11:48:22.30045-04  |           31 | ERROR       | Column "UDX" does not exist
 2012-03-30 11:21:14.659128-04 |           20 | ERROR       | Invalid attnum 1 for rangetable entry
t
...
```

EVENT_CONFIGURATIONS

Monitors the configuration of events.

Column Name	Date Type	Description
EVENT_ID	VARCHAR	The name of the event.
EVENT_DELIVERY_CHANNELS	VARCHAR	The delivery channel on which the event occurred.

Example

```
=> SELECT * FROM event_configurations;

                 event_id                  | event_delivery_channels
-------------------------------------------+-------------------------
 Low Disk Space                            | Vertica Log, SNMP Trap
 Read Only File System                     | Vertica Log, SNMP Trap
 Loss Of K Safety                          | Vertica Log, SNMP Trap
 Current Fault Tolerance at Critical Level | Vertica Log, SNMP Trap
 Too Many ROS Containers                   | Vertica Log, SNMP Trap
 WOS Over Flow                             | Vertica Log, SNMP Trap
 Node State Change                         | Vertica Log, SNMP Trap
 Recovery Failure                          | Vertica Log, SNMP Trap
 Recovery Error                            | Vertica Log
 Recovery Lock Error                       | Vertica Log
 Recovery Projection Retrieval Error       | Vertica Log
 Refresh Error                             | Vertica Log
 Refresh Lock Error                        | Vertica Log
 Tuple Mover Error                         | Vertica Log
 Timer Service Task Error                  | Vertica Log
 Stale Checkpoint                          | Vertica Log, SNMP Trap
(16 rows)
```

EXECUTION_ENGINE_PROFILES

Provides profiling information about query execution runs. The hierarchy of IDs, from highest level to actual execution is:

- PATH_ID
- BASEPLAN_ID
- LOCALPLAN_ID
- OPERATOR_ID

Counters (output from the COUNTER_NAME column) are collected for each actual Execution Engine (EE) operator instance.

The following columns form a unique key for rows in the Data Collector table DC_EXECUTION_ENGINE_EVENTS:

- TRANSACTION_ID
- STATEMENT_ID
- NODE_NAME
- OPERATOR_ID
- COUNTER_NAME
- COUNTER_TAG

For additional details about profiling and debugging, see Profiling Database Performance in the Administrator's Guide.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node name for which information is listed.
USER_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	User name for which query profile information is listed.
SESSION_ID	VARCHAR	Identifier of the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session if any; otherwise NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed.
OPERATOR_NAME	VARCHAR	Name of the Execution Engine (EE) component; for example, NetworkSend.

OPERATOR_ID	INTEGER	Identifier assigned by the EE operator instance that performs the work. OPERATOR_ID is different from LOCALPLAN_ID because each logical operator, such as Scan, may be executed by multiple threads concurrently. Each thread operates on a different operator instance, which has its own ID.
BASEPLAN_ID	INTEGER	Assigned by the optimizer on the initiator to EE operators in the original base (EXPLAIN) plan. Each EE operator in the base plan gets a unique ID.
PATH_ID	INTEGER	Identifier that HP Vertica assigns to a query operation or *path*; for example to a logical grouping operation that might be performed by multiple execution engine operators. For each path, the same PATH ID is shared between the query plan (using EXPLAIN output) and in error messages that refer to joins.
LOCALPLAN_ID	INTEGER	Identifier assigned by each local executor while preparing for plan execution (local planning). Some operators in the base plan, such as the Root operator, which is connected to the client, do not run on all nodes. Similarly, certain operators, such as ExprEval, are added and removed during local planning due to implementation details.
ACTIVITY_ID	INTEGER	Identifier of the plan activity.
RESOURCE_ID	INTEGER	Identifier of the plan resource.
COUNTER_NAME	VARCHAR	Name of the counter. See the "COUNTER_NAME Values" section below this table.
COUNTER_TAG	VARCHAR	String that uniquely identifies the counter for operators that might need to distinguish between different instances. For example, COUNTER_TAG is used to identify to which of the node bytes are being sent to or received from for the NetworkSend operator.
COUNTER_VALUE	INTEGER	Value of the counter.
IS_EXECUTING	BOOLEAN	Indicates whether the profile is active or completed, where *t* is active and *f* is completed.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

COUNTER_NAME Values

The value of COUNTER_NAME can be any of the following:

COUNTER_NAME	Description
buffers spilled	[NetworkSend] Buffers spilled to disk by NetworkSend.
bytes received	[NetworkRecv] The number of bytes received over the network for query execution.
bytes read from cache	[DataSource] The number of bytes read from HP Vertica cache when an EE DataSource operator is reading from ROS containers.
bytes read from disk	[DataSource] The number of bytes read from disk when an EE DataSource operator is reading from ROS containers.
bytes sent	[NetworkSend] Size of data after encoding and compression sent over the network (actual network bytes).
bytes spilled	[NetworkSend] Bytes spilled to disk by NetworkSend.
bytes total	Only relevant to SendFiles operator (that is, recover-by-container plan) total number of bytes to send / receive.
clock time (us)	Real-time clock time spent processing the query, in microseconds.
completed merge phases	Number of merge phases already completed by an LSort or DataTarget operator. Compare to the total merge phases. Variants on this value include join inner completed merge phases.
cumulative size of raw temp data (bytes)	Total amount of temporary data the operator has written to files. Compare to cumulative size of temp files (bytes) to understand impact of encoding and compression in an externalizing operator. Variants on this value include join inner cumulative size of raw temp files (bytes).
cumulative size of temp files (bytes)	For externalizing operators only, the total number of encoded and compressed temp data the operator has written to files. A sort operator might go through multiple merge phases, where at each pass sorted chunks of data are merged into fewer chunks. This counter remembers the cumulative size of all temp files past and present. Variants on this value include join inner cumulative size of temp files (bytes).
current size of temp files (bytes)	For externalizing operators only, the current size of the encoded and compressed temp data that the operator has written to files. Variants on this value include join inner current size of temp files (bytes).
distinct value estimation time (µs)	[Analyze Statistics] Time spent estimating the number of distinct values from the sample after data has been read off disk and into the statistical sample.

encoded bytes received	[NetworkRecv] Size of received data after decompressed (but still encoded) received over the network.
encoded bytes sent	[NetworkSend] Size of data sent over the network after encoding.
end time	Time (timestamp) when HP Vertica stopped processing the operation
estimated rows produced	Number of rows that the optimizer estimated would be produced. See rows produced for the actual number of rows that are produced.
execution time (us)	CPU clock time spent processing the query, in microseconds.
Exceptions cumulative size of raw temp data (bytes)	Counters that store total or current size of exception data.
Exceptions rows cumulative size of temp files (bytes)	
Exceptions rows current size of temp files (bytes)	
files completed	Relevant only to SendFiles/RecvFiles operators (that is, recover-by-container plan) number of files sent / received.
file handles	Number of file handles used by the operator.
files total	Relevant only to SendFiles/RecvFiles operators (that is, recover-by-container plan) total number of files to send / receive.
histogram creation time (µs)	[Analyze Statistics] Time spent estimating the number of distinct values from the sample after data has been read off disk and into the statistical sample.
input queue wait (µs)	Time in microseconds that an operator spends waiting for upstream operators.
input rows	Actual number of rows that were read into the operator.
input size (bytes)	Total number of bytes of the Load operator's input source, where NULL is unknown (read from FIFO).
join inner completed merge phases	See the completed merge phases counter.
join inner cumulative size of raw temp data (bytes)	
join inner cumulative size of temp files (bytes)	
join inner current size of temp files (bytes)	
join inner total merge phases	
max sample size (rows)	[Analyze Statistics] Maximum number of rows that will be stored in the statistical sample.

memory allocated (bytes)	Actual memory in bytes that the operator allocated at run time.
memory reserved (bytes)	Memory reserved by the operator in the ResourceManager operator. **Note:** An allocation slightly more than the reservation (a few MB) is not a cause for concern and is built into ResourceManager calculations.
network wait (µs)	[NetworkSend, NetworkRecv] Time in microseconds spent waiting on the network.
output queue wait (µs)	Time in microseconds that an operator spends waiting for the output buffer to be consumed by a downstream operator.
producer stall (µs)	[NetworkSend] Time in microseconds spent by NetworkSend when stalled waiting for network buffers to clear.
producer wait (µs)	[NetworkSend] Time in microseconds spent by the input operator making rows to send.
read (bytes)	Number of bytes read from the input source by the Load operator.
receive time (µs)	Time in microseconds that a Recv operator spends reading data from its socket.
rejected data cumulative size of raw temp data (bytes)	Counters that store total or current size of rejected row numbers. Are variants of: ▪ cumulative size of raw temp data (bytes) ▪ cumulative size of temp files (bytes) ▪ current size of temp files (bytes)
rejected data cumulative size of temp files (bytes)	
rejected data current size of temp files (bytes)	
rejected rows cumulative size of raw temp data (bytes)	
rejected rows cumulative size of temp files (bytes)	
rejected rows current size of temp files (bytes)	
rle rows produced	Number of physical tuples produced by an operator. Complements the rows produced counter, which shows the number of logical rows produced by an operator. For example, if a value occurs 1000 rows consecutively and is RLE encoded, it counts as 1000 rows produced not only 1 rle rows produced.
ROS blocks bounded	[DataTarget] Number of ROS blocks created, due to boundary alignment with RLE prefix columns, when an EE DataTarget operator is writing to ROS containers.
ROS blocks encoded	[DataTarget] Number of ros blocks created when an EE DataTarget operator is writing to ROS containers.

ROS bytes written	[DataTarget] Number of bytes written to disk when an EE DataTarget operator is writing to ROS containers.
rows in sample	[Analyze Statistics] Actual number of rows that will be stored in the statistical sample.
rows output by sort	[DataTarget] Number of rows sorted when an EE DataTarget operator is writing to ROS containers.
rows processed	[DataSource] Number of rows processed when an EE DataSource operator is writing to ROS containers
rows produced	Number of logical rows produced by an operator. See also the rle rows produced counter.
rows pruned by valindex	[DataSource] Number of rows it skips direct scanning with help of valindex when an EE DataSource operator is writing to ROS containers. This counter's value is not greater than "rows processed" counter.
rows received	[NetworkRecv] Number of received sent over the network.
rows rejected	The number of rows rejected by the Load operator.
rows sent	[NetworkSend] Number of rows sent over the network.
send time (µs)	Time in microseconds that a Send operator spends writing data to its socket.
start time	Time (timestamp) when HP Vertica started to process the operation.
total merge phases	Number of merge phases an LSort or DataTarget operator must complete to finish sorting its data. NULL until the operator can compute this value (all data must first be ingested by the operator). Variants on this value include join inner total merge phases.
wait clock time (µs)	StorageUnion wait time in microseconds.
WOS bytes acquired	Number of bytes acquired from the WOS by a DataTarget operator. **Note:** This is usually more but can be less than WOS bytes written if an earlier statement in the transaction acquired some WOS memory.
WOS bytes written	Number of bytes written to the WOS by a DataTarget operator.
written rows	[DataTarget] Number of rows written when an EE DataTarget operator writes to ROS containers

Examples

The next two examples show the contents of the EXECUTION_ENGINE_PROFILES table:

```
=> SELECT  operator_name, operator_id, counter_name, counter_value
   FROM EXECUTION_ENGINE_PROFILES WHERE operator_name = 'Scan'
```

```
   ORDER BY counter_value DESC;
 operator_name | operator_id |       counter_name       |  counter_value
---------------+-------------+--------------------------+-----------------
 Scan          |          12 | end time                 | 397916465478595
 Scan          |           9 | end time                 | 397916465478510
 Scan          |          12 | start time               | 397916465462098
 Scan          |           9 | start time               | 397916465447998
 Scan          |          14 | bytes read from disk     |        28044535
 Scan          |          14 | bytes read from disk     |        28030212
 Scan          |          12 | rows processed           |         5000000
 Scan          |          12 | estimated rows produced  |         4999999
 Scan          |          18 | rows produced            |         1074828
 Scan          |          18 | rle rows produced        |         1074828
 Scan          |           3 | memory allocated (bytes) |         1074568
 Scan          |           7 | rows produced            |          799526
 Scan          |           7 | rle rows produced        |          799526
 Scan          |           7 | memory allocated (bytes) |          682592
 Scan          |          12 | clock time (us)          |          673806
 Scan          |           7 | execution time (us)      |          545717
 Scan          |           3 | memory allocated (bytes) |          537400
 Scan          |          12 | clock time (us)          |          505315
 Scan          |          14 | execution time (us)      |          495176
 Scan          |           3 | bytes read from disk     |          452403
 Scan          |          14 | execution time (us)      |          420189
 Scan          |          12 | execution time (us)      |          404184
 Scan          |          18 | clock time (us)          |          398751
 Scan          |          18 | execution time (us)      |          339321
(24 rows)


=> SELECT DISTINCT counter_name FROM execution_engine_profiles;
                    counter_name
-----------------------------------------------------
 end time
 clock time (us)
 rle rows produced
 bytes read from disk
 start time
 rows processed
 memory allocated (bytes)
 estimated rows produced
 rows produced
 execution time (us)
(10 rows)
```

The notable thing about the following query is the `path_id` column, which links the path that the query optimizer takes (via the EXPLAIN command's textual output) with join error messages.

```
=> SELECT operator_name, path_id, counter_name,
   counter_value FROM execution_engine_profiles;

 operator_name | path_id |       counter_name        | counter_value
---------------+---------+---------------------------+---------------
```

```
Join          |       1 | estimated rows produced   |         10000
Join          |       1 | file handles              |             0
Join          |       1 | memory allocated (bytes)  |       2405824
Join          |       1 | memory reserved (bytes)   |       1769472
Join          |       1 | rle rows produced         |             3
Join          |       1 | rows produced             |             3
Join          |       1 | clock time (us)           |         24105
Join          |       1 | execution time (us)       |           235
```

See Also

Profiling Database Performance in the Troubleshooting Guide, particularly Viewing Profiling Data

Linking EXPLAIN Plan Output to Error Messages in the Troubleshooting Guide

HOST_RESOURCES

Provides a snapshot of the node. This is useful for regularly polling the node with automated tools or scripts.

Column Name	Data Type	Description
HOST_NAME	VARCHAR	The host name for which information is listed.
OPEN_FILES_LIMIT	INTEGER	The maximum number of files that can be open at one time on the node.
THREADS_LIMIT	INTEGER	The maximum number of threads that can coexist on the node.
CORE_FILE_LIMIT_MAX_SIZE_BYTES	INTEGER	The maximum core file size allowed on the node.
PROCESSOR_COUNT	INTEGER	The number of system processors.
PROCESSOR_CORE_COUNT	INTEGER	The number of processor cores in the system.
PROCESSOR_DESCRIPTION	VARCHAR	A description of the processor. For example: Inter(R) Core(TM)2 Duo CPU T8100 @2.10GHz (1 row)
OPENED_FILE_COUNT	INTEGER	The total number of open files on the node.
OPENED_SOCKET_COUNT	INTEGER	The total number of open sockets on the node.
OPENED_NONFILE_NONSOCKET_COUNT	INTEGER	The total number of *other* file descriptions open in which 'other' could be a directory or FIFO. It is not an open file or socket.
TOTAL_MEMORY_BYTES	INTEGER	The total amount of physical RAM, in bytes, available on the system.
TOTAL_MEMORY_FREE_BYTES	INTEGER	The amount of physical RAM, in bytes, left unused by the system.

TOTAL_BUFFER_MEMORY_BYTES	INTEGER	The amount of physical RAM, in bytes, used for file buffers on the system
TOTAL_MEMORY_CACHE_BYTES	INTEGER	The amount of physical RAM, in bytes, used as cache memory on the system.
TOTAL_SWAP_MEMORY_BYTES	INTEGER	The total amount of swap memory available, in bytes, on the system.
TOTAL_SWAP_MEMORY_FREE_BYTES	INTEGER	The total amount of swap memory free, in bytes, on the system.
DISK_SPACE_FREE_MB	INTEGER	The free disk space available, in megabytes, for all storage location file systems (data directories).
DISK_SPACE_USED_MB	INTEGER	The disk space used, in megabytes, for all storage location file systems.
DISK_SPACE_TOTAL_MB	INTEGER	The total free disk space available, in megabytes, for all storage location file systems.

Examples

Query the HOST_RESOURCES table:

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM HOST_RESOURCES;

-[ RECORD 1 ]-----------------+-------------------------------------
host_name                     | myhost-s1
open_files_limit              | 65536
threads_limit                 | 15914
core_file_limit_max_size_bytes | 1649680384
processor_count               | 2
processor_core_count          | 8
processor_description         | Intel(R) Xeon(R) CPU E5504  @ 2.00GHz
opened_file_count             | 5
opened_socket_count           | 4
opened_nonfile_nonsocket_count | 3
total_memory_bytes            | 16687161344
total_memory_free_bytes       | 4492627968
total_buffer_memory_bytes     | 1613922304
total_memory_cache_bytes      | 9349111808
total_swap_memory_bytes       | 36502126592
total_swap_memory_free_bytes  | 36411580416
disk_space_free_mb            | 121972
disk_space_used_mb            | 329235
disk_space_total_mb           | 451207
-[ RECORD 2 ]-----------------+-------------------------------------
host_name                     | myhost-s2
open_files_limit              | 65536
threads_limit                 | 15914
core_file_limit_max_size_bytes | 3772891136
processor_count               | 2
processor_core_count          | 4
processor_description         | Intel(R) Xeon(R) CPU E5504  @ 2.00GHz
opened_file_count             | 5
opened_socket_count           | 3
opened_nonfile_nonsocket_count | 3
total_memory_bytes            | 16687161344
total_memory_free_bytes       | 9525706752
```

```
total_buffer_memory_bytes      | 2840420352
total_memory_cache_bytes       | 3060588544
total_swap_memory_bytes        | 34330370048
total_swap_memory_free_bytes   | 34184642560
disk_space_free_mb             | 822190
disk_space_used_mb             | 84255
disk_space_total_mb            | 906445
-[ RECORD 3 ]------------------+-------------------------------------
host_name                      | myhost-s3
open_files_limit               | 65536
threads_limit                  | 15914
core_file_limit_max_size_bytes | 3758211072
processor_count                | 2
processor_core_count           | 4
processor_description          | Intel(R) Xeon(R) CPU E5504  @ 2.00GHz
opened_file_count              | 5
opened_socket_count            | 3
opened_nonfile_nonsocket_count | 3
total_memory_bytes             | 16687161344
total_memory_free_bytes        | 9718706176
total_buffer_memory_bytes      | 2928369664
total_memory_cache_bytes       | 2757115904
total_swap_memory_bytes        | 34315689984
total_swap_memory_free_bytes   | 34205523968
disk_space_free_mb             | 820789
disk_space_used_mb             | 85640
disk_space_total_mb            | 906429
-[ RECORD 4 ]------------------+-------------------------------------
host_name                      | myhost-s4
open_files_limit               | 65536
threads_limit                  | 15914
core_file_limit_max_size_bytes | 3799433216
processor_count                | 2
processor_core_count           | 8
processor_description          | Intel(R) Xeon(R) CPU E5504  @ 2.00GHz
opened_file_count              | 5
opened_socket_count            | 3
opened_nonfile_nonsocket_count | 3
total_memory_bytes             | 16687161344
total_memory_free_bytes        | 8772620288
total_buffer_memory_bytes      | 3792273408
total_memory_cache_bytes       | 2831040512
total_swap_memory_bytes        | 34356912128
total_swap_memory_free_bytes   | 34282590208
disk_space_free_mb             | 818896
disk_space_used_mb             | 55291
disk_space_total_mb            | 874187
```

IO_USAGE

Provides disk I/O bandwidth usage history for the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
START_TIME	TIMESTAMP	Beginning of history interval.

END_TIME	TIMESTAMP	End of history interval.
READ_KBYTES_PER_SEC	FLOAT	Counter history of the number of bytes read measured in kilobytes per second.
WRITTEN_KBYTES_PER_SEC	FLOAT	Counter history of the number of bytes written measured in kilobytes per second.

Permissions

Must be a superuser

Example

```
=> SELECT * FROM io_usage;

 node_name |          start_time           |           end_time            | read_kbytes_per_sec |
written_kbytes_per_sec
-----------++-----------------------------+-------------------------------+---------------------
+----------------------
 initiator | 2011-09-16 15:23:38.003714-04 | 2011-09-16 15:24:00.015769-04 |               11.63 |
620.03
 e0        | 2011-09-16 15:23:37.005431-04 | 2011-09-16 15:24:00.011519-04 |               11.13 |
593.24
 initiator | 2011-09-16 15:24:00.015728-04 | 2011-09-16 15:25:00.015439-04 |               18.67 |
473.87
 e1        | 2011-09-16 15:24:00.014491-04 | 2011-09-16 15:25:00.010432-04 |               18.67 |
473.9
 e1        | 2011-09-16 15:23:37.006595-04 | 2011-09-16 15:24:00.014533-04 |               11.13 |
593.18
 e0        | 2011-09-16 15:24:00.011478-04 | 2011-09-16 15:25:00.017536-04 |               18.66 |
473.82
(6 rows)
```

LOAD_STREAMS

Monitors active and historical load metrics for load streams on each node. This is useful for obtaining statistics about how many records got loaded and rejected from the previous load. HP Vertica maintains system table metrics until they reach a designated size quota (in kilobytes). The quota is set through internal processes and cannot be set or viewed directly.

Column Name	Date Type	Description
SESSION_ID	VARCHAR	Identifier of the session for which HP Vertica captures load stream information. This identifier is unique within the cluster for the current session, but can be reused in a subsequent session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within a session. If a session is active but no transaction has begun, this is NULL.

STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
STREAM_NAME	VARCHAR	Load stream identifier. If the user does not supply a specific name, the STREAM_NAME default value is: *tablename-ID* where *tablename* is the table into which data is being loaded, and *ID* is an integer value, guaranteed to be unique with the current session on a node. The LOAD_STREAMS system table includes stream names for every COPY statement that takes more than 1-second to run. The 1-second duration includes the time to plan and execute the statement.
SCHEMA_NAME	VARCHAR	Schema name for which load stream information is listed. Lets you identify two streams that are targeted at tables with the same name in different schemas
TABLE_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog that identifies the table.
TABLE_NAME	VARCHAR	Name of the table being loaded.
LOAD_START	VARCHAR	Linux system time when the load started.
LOAD_DURATION_MS	NUMERIC(54, 0)	Duration of the load stream in milliseconds.
IS_EXECUTING	BOOLEAN	Indicates whether the load is executing, where *t* is true and *f* is false.
ACCEPTED_ROW_COUNT	INTEGER	Number of rows loaded.
REJECTED_ROW_COUNT	INTEGER	Number of rows rejected.
READ_BYTES	INTEGER	Number of bytes read from the input file.
INPUT_FILE_SIZE_BYTES	INTEGER	Size of the input file in bytes. **Note:** When using STDIN as input, the input file size is zero (0).
PARSE_COMPLETE_PERCENT	INTEGER	Percent of rows from the input file that have been loaded.
UNSORTED_ROW_COUNT	INTEGER	Cumulative number rows not sorted across all projections. **Note:** UNSORTED_ROW_COUNT could be greater than ACCEPTED_ROW_COUNT because data is copied and sorted for every projection in the target table.

SORTED_ROW_COUNT	INTEGER	Cumulative number of rows sorted across all projections.
SORT_COMPLETE_PERCENT	INTEGER	Percent of rows from the input file that have been sorted.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

See also

Checking Load Stream Metrics and Using System Tables In the Administrator's Guide

LOCK_USAGE

Provides aggregate information about lock requests, releases, and attempts, such as wait time/count and hold time/count. HP Vertica records:

- Lock attempts at the end of the locking process
- Lock releases after locks attempts are released

See also system table ***LOCKS*** (page 1037).

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information on which lock interaction occurs.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
OBJECT_NAME	VARCHAR	Name of object being locked; can be a table or an internal structure (projection, global catalog, or local catalog).

MODE	VARCHAR	Intended operations of the transaction: ▪ S — Share lock needed for select operations. ▪ I — Insert lock needed for insert operations. ▪ SI — Share+Insert lock needed for operations that read and query the table. Distinguished from X because SI mode disallows delete/update operations. SI is also the result of lock promotion (see Table 2). ▪ X — Exclusive lock is always needed for delete operations. X lock is also the result of lock promotion (see Table 2). ▪ T — Tuple Mover lock used by the Tuple Mover and also used for COPY into pre-join projections. ▪ U — Usage lock needed for moveout and mergeout operations in the first phase; they then upgrade their U lock to a T lock for the second phase. U locks conflicts with no other locks but O. ▪ O — Owner lock needed for DROP_PARTITION, TRUNCATE TABLE, and ADD COLUMN. O locks conflict with all locks. O locks never promote. ▪ NONE
AVG_HOLD_TIME	INTERVAL	Average time (measured in intervals) that HP Vertica holds a lock.
MAX_HOLD_TIME	INTERVAL	Maximum time (measured in intervals) that HP Vertica holds a lock.
HOLD_COUNT	INTEGER	Total number of times lock was taken in the given mode.
AVG_WAIT_TIME	INTERVAL	Average time (measured in intervals) that HP Vertica waits on the lock
MAX_WAIT_TIME	INTERVAL	Maximum time (measured in intervals) that HP Vertica waits on a lock
WAIT_COUNT	INTEGER	Total number of times lock was unavailable at the time it was first requested.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Notes

The following two tables are adapted from *Transaction Processing: Concepts and Techniques* *http://www.amazon.com/gp/product/1558601902/ref=s9sdps_c1_14_at1-rfc_p-frt_p-3237_g1_si1?pf_rd_m=ATVPDKIKX0DER&pf_rd_s=center-1&pf_rd_r=1QHH6V589JEV0DR3DQ1D&pf_rd_t=101&pf_rd_p=463383351&pf_rd_i=507846* by Jim Gray (Figure 7.11, p. 408 and Figure 8.6, p. 467).

Table 1: Lock compatibility matrix

This table is for compatibility with other users. The table is symmetric.

	Granted Mode						
Requested Mode	**S**	**I**	**SI**	**X**	**T**	**U**	**O**
S	Yes	No	No	No	Yes	Yes	No
I	**No**	YES	No	No	Yes	Yes	No
SI	No	No	No	No	Yes	Yes	No
X	No	No	No	No	No	Yes	No
T	Yes	Yes	Yes	No	Yes	Yes	No
U	Yes	Yes	Yes	Yes	Yes	Yes	No
O	No	No	No	No	No	No	No

The following two examples refer to Table 1 above:

- **Example 1:** If someone else has an S lock, you cannot get an I lock.
- EXAMPLE 2: If someone has an I lock, you can get an I lock.

Table 2: Lock conversion matrix

This table is used for upgrading locks you already have. For example, If you have an S lock and you want an I lock, you request an **X** lock. If you have an S lock and you want an S lock, no lock requests is required.

	Granted Mode						
Requested Mode	**S**	**I**	**SI**	**X**	**T**	**U**	**O**
S	S	X	SI	X	S	S	O
I	**X**	I	SI	X	I	I	O
SI	SI	SI	SI	X	SI	SI	O
X	X	X	X	X	X	X	O
T	S	I	SI	X	T	T	O
U	S	I	SI	X	T	U	O

O	O	O	O	O	O	O	O

Example

```
=> SELECT * FROM lock_usage;

-[ RECORD 1 ]-+---------------------------------
node_name     | v_myvdb_node0004
session_id    | raster-f1.verticaco-13199:0x100
object_name   | Global Catalog
mode          | S
avg_hold_time | 00:00:00.032718
max_hold_time | 00:00:00.251213
hold_count    | 34
avg_wait_time | 00:00:00.000048
max_wait_time | 00:00:00.000119
wait_count    | 0
-[ RECORD 2 ]-+---------------------------------
node_name     | v_myvdb_node0004
session_id    | raster-f1.verticaco-13199:0x102
object_name   | Global Catalog
mode          | S
avg_hold_time | 00:00:00.038148
max_hold_time | 00:00:00.185088
hold_count    | 34
avg_wait_time | 00:00:00.000049
max_wait_time | 00:00:00.000124
wait_count    | 0
-[ RECORD 3 ]-+---------------------------------
node_name     | v_myvdb_node0004
session_id    | raster-f1.verticaco-13199:0x119
object_name   | Table:public.dwdate
mode          | T
avg_hold_time | 00:00:04.269108
max_hold_time | 00:00:04.269108
hold_count    | 1
avg_wait_time | 00:00:00.000019
max_wait_time | 00:00:00.000019
wait_count    | 0
-[ RECORD 4 ]-+---------------------------------
node_name     | v_myvdb_node0004
session_id    | raster-f1.verticaco-13199:0x11d
object_name   | Global Catalog
mode          | X
avg_hold_time | 00:00:00.027261
max_hold_time | 00:00:00.027261
hold_count    | 1
avg_wait_time | 00:00:00.000038
max_wait_time | 00:00:00.000038
wait_count    | 0
...
```

This example shows an INSERT lock in use:

```
=>\pset expanded
```

```
Expanded display is on.
=> SELECT * FROM LOCKS;
-[ RECORD 1
]----------+-------------------------------------------------------------
node_names              | node01,node02,node03,node04
object_name             | Table:fact
object_id               | 45035996273772278
transaction_description | Txn: a000000000112b 'COPY fact FROM '/data_dg/fact.dat'
                          DELIMITER '|' NULL '\\N';'
lock_mode               | I
lock_scope              | TRANSACTION
request_timestamp       | 2011-04-17 14:01:07.662325-04
grant_timestamp         | 2011-04-17 14:01:07.662325-04
```

See Also

DUMP_LOCKTABLE (page 478)

LOCKS (page 1037)

PROJECTION_REFRESHES (page 1056)

SELECT (page 870) FOR UPDATE clause

SESSION_PROFILES (page 1093)

LOCKS

Monitors lock grants and requests for all nodes.

See also ***LOCK_USAGE*** (page 1033) for aggregate information about lock requests, releases, and attempts, such as wait time/count and hold time/count.

Column Name	Date Type	Description
NODE_NAMES	VARCHAR	Nodes on which lock interaction occurs. **Note on node rollup:** If a transaction has the same lock in the same mode in the same scope on multiple nodes, it gets one (1) line in the table. NODE_NAMES are separated by commas.
OBJECT_NAME	VARCHAR	Name of object being locked; can be a table or an internal structure (projection, global catalog, or local catalog).
OBJECT_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the object being locked.
TRANSACTION_ID	VARCHAR	Identifier for the transaction within the session, if any; otherwise NULL. Useful for creating joins to

		other system tables.
TRANSACTION_DESCRIPTION	VARCHAR	Identification of transaction and associated description, typically the query that caused the transaction's creation.
LOCK_MODE	VARCHAR	Intended operation of the transaction: ▪ S — Share lock needed for select operations. Select operations in READ COMMITTED transaction mode do not require share (S) table locks. See Transactions in the Concepts Guide. ▪ I — Insert lock needed for insert operations. ▪ SI — Share+Insert lock needed for operations that read and query the table. Distinguished from X because SI mode disallows delete/update operations. SI is also the result of lock promotion (see Table 2). ▪ X — Exclusive lock is always needed for delete operations. X lock is also the result of lock promotion (see Table 2). ▪ T — Tuple Mover lock used by the Tuple Mover and also used for COPY into pre-join projections ▪ U — Usage lock needed for moveout and mergeout operations in the first phase; they then upgrade their U lock to a T lock for the second phase. U locks conflicts with no other locks but O. ▪ O — Owner lock needed for DROP_PARTITION, TRUNCATE TABLE, and ADD COLUMN. O locks conflict with all locks. O locks never promote.
LOCK_SCOPE	VARCHAR	Expected duration of the lock once it is granted. Before the lock is granted, the scope is listed as REQUESTED. Once a lock has been granted, the following scopes are possible: ▪ STATEMENT_LOCALPLAN ▪ STATEMENT_COMPILE ▪ STATEMENT_EXECUTE ▪ TRANSACTION_POSTCOMMIT ▪ TRANSACTION All scopes, other than TRANSACTION, are transient and are used only as part of normal query processing.

REQUEST_TIMESTAMP	TIMESTAMP	Time when the transaction started waiting on the lock.
GRANT_TIMESTAMP	TIMESTAMP	Time the transaction acquired or upgraded the lock. **Notes:** ▪ Return values are NULL until the grant occurs. ▪ Values could be the same as REQUEST_TIMESTAMP if the grant occurs immediately.

Notes

- Lock acquisition and wait times are collected and exposed via the START_TIMESTAMP column.
- Locks acquired on tables that were subsequently dropped by another transaction can result in the message, `Unknown or deleted object`, appearing in the output's `OBJECT` column.
- If a `SELECT..FROM LOCKS` query times out after five minutes, it is possible the cluster has failed. Run the Diagnostics Utility.
- A table call with no results indicates that no locks are in use.

The following two tables are adapted from *Transaction Processing: Concepts and Techniques http://www.amazon.com/gp/product/1558601902/ref=s9sdps_c1_14_at1-rfc_p-frt_p-3237_g1_si1?pf_rd_m=ATVPDKIKX0DER&pf_rd_s=center-1&pf_rd_r=1QHH6V589JEV0DR3DQ1D&pf_rd_t=101&pf_rd_p=463383351&pf_rd_i=507846* by Jim Gray (Figure 7.11, p. 408 and Figure 8.6, p. 467).

Table 1: Lock compatibility matrix

This table is for compatibility with other users. The table is symmetric.

	Granted Mode						
Requested Mode	**S**	**I**	**SI**	**X**	**T**	**U**	**O**
S	Yes	No	No	No	Yes	Yes	No
I	**No**	YES	No	No	Yes	Yes	No
SI	No	No	No	No	Yes	Yes	No
X	No	No	No	No	No	Yes	No
T	Yes	Yes	Yes	No	Yes	Yes	No
U	Yes	Yes	Yes	Yes	Yes	Yes	No
O	No	No	No	No	No	No	No

The following two examples refer to Table 1 above:

- **Example 1:** If someone else has an S lock, you cannot get an I lock.
- EXAMPLE 2: If someone has an I lock, you can get an I lock.

Table 2: Lock conversion matrix

This table is used for upgrading locks you already have. For example, If you have an S lock and you want an I lock, you request an **X** lock. If you have an S lock and you want an S lock, no lock requests is required.

	Granted Mode						
Requested Mode	**S**	**I**	**SI**	**X**	**T**	**U**	**O**
S	S	X	SI	X	S	S	O
I	**X**	I	SI	X	I	I	O
SI	SI	SI	SI	X	SI	SI	O
X	X	X	X	X	X	X	O
T	S	I	SI	X	T	T	O
U	S	I	SI	X	T	U	O
O	O	O	O	O	O	O	O

Example

This example shows an INSERT lock in use:

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM LOCKS;
-[ RECORD 1
]-----------+-------------------------------------------------------------
node_names              | node01,node02,node03,node04
object_name             | Table:fact
object_id               | 45035996273772278
transaction_description | Txn: a000000000112b 'COPY fact FROM '/data_dg/fact.dat'
                          DELIMITER '|' NULL '\\N';'
lock_mode               | I
lock_scope              | TRANSACTION
request_timestamp       | 2011-04-17 14:01:07.662325-04
grant_timestamp         | 2011-04-17 14:01:07.662325-04
```

See Also

Transactions in the Concepts Guide

DUMP_LOCKTABLE (page 478)

LOCK_USAGE (page 1033)

PROJECTION_REFRESHES (page 1056)

SELECT (page 870) FOR UPDATE clause

SESSION_PROFILES (page 1093)

LOGIN_FAILURES

Lists failures for each user failed login attempt. This information is useful for determining if a user is having trouble getting into the database, as well as identifying a possible intrusion attempt.

Column Name	Data Type	Description
LOGIN_TIMESTAMP	TIMESTAMPTZ	Time when HP Vertica recorded the login.
DATABASE_NAME	VARCHAR	The name of the database for the login attempt.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user whose login failed at the time HP Vertica recorded the session.
CLIENT_HOSTNAME	VARCHAR	Host name and port of the TCP socket from which the client connection was made; NULL if the session is internal.
CLIENT_PID	INTEGER	Identifier of the client process that issued this connection. **Note:** The client process could be on a different machine from the server.
CLIENT_VERSION	VARCHAR	Unused
AUTHENTICATION_METHOD	VARCHAR	Determines whether the client application (or the user who is running the client application) is permitted to connect to the server using the database user name provided. HP Vertica supports the following client authentication methods: ▪ Trust ▪ Reject ▪ Kerberos 5 ▪ GSS ▪ LDAP ▪ Ident ▪ Password See Implementing Client Authentication in the Administrator's Guide for details.

REASON	VARCHAR	Description of login failure reason: ▪ "INVALID USER" ▪ "ACCOUNT LOCKED" ▪ "REJECT" ▪ "FAILED" ▪ "INVALID AUTH METHOD" ▪ "INVALID DATABASE"

Permissions

Must be a superuser.

Example

```
=> SELECT * FROM login_failures;
        login_timestamp        | database_name | node_name | user_name | client_hostname | client_pid
| client_version | authentication_method |    reason
-------------------------------+---------------+-----------+-----------+-----------------+-------
-----+----------------+-----------------------+-------------
 2012-03-25 13:38:56.54813-04  | smith         | node01    | u1        | 127.0.0.1       |
32061 |                | Password              | FAILED
 2012-03-25 13:38:56.543396-04 | smith         | node01    | bad_user  | 127.0.0.1       |
32059 |                | Password              | FAILED
 2012-03-25 13:38:56.543225-04 | bad_user      | node01    | bad_user  | 127.0.0.1       |
32059 |                | Reject                | INVALID USER
(3 rows)
```

MEMORY_USAGE

Records system resource history for memory usage. This is useful for comparing memory that HP Vertica uses versus memory in use by the entire system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
START_TIME	TIMESTAMP	Beginning of history interval.
END_TIME	TIMESTAMP	End of history interval.
AVERAGE_MEMORY_USAGE_PERCENT	FLOAT	Records the average memory usage in percent of total memory (0-100) during the history interval.

Permissions

Must be a superuser.

Example

```
=> SELECT * FROM memory_usage;
```

```
 node_name |          start_time           |           end_time            |
average_memory_usage_percent
-----------+-------------------------------+-------------------------------+-----------------
--------
 initiator | 2012-03-25 15:23:38.001952-04 | 2012-03-25 15:24:00.011432-04 |
50.39
 initiator | 2012-03-25 15:24:00.011432-04 | 2012-03-25 15:25:00.008539-04 |
51.08
 e0        | 2012-03-25 15:23:37.00339-04  | 2012-03-25 15:24:00.006837-04 |
50.3
 e0        | 2012-03-25 15:24:00.006837-04 | 2012-03-25 15:25:00.011629-04 |
51.08
 e1        | 2012-03-25 15:23:37.001344-04 | 2012-03-25 15:24:00.009634-04 |
50.3
 e1        | 2012-03-25 15:24:00.009634-04 | 2012-03-25 15:25:00.005176-04 |
51.08
(6 rows)
```

MONITORING_EVENTS

Reports significant events that can affect database performance and functionality if you do not address their root causes.

See Monitoring Events in the Administrator's Guide for details.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
EVENT_CODE	INTEGER	Numeric identifier that indicates the type of event. See Event Types in Monitoring Events in the Administrator's Guide for a list of event type codes.
EVENT_ID	INTEGER	Unique numeric ID that identifies the specific event.
EVENT_SEVERITY	VARCHAR	Severity of the event from highest to lowest. These events are based on standard syslog severity types: 0 – Emergency 1 – Alert 2 – Critical 3 – Error 4 – Warning 5 – Notice 6 – Info 7 – Debug
EVENT_POSTED_TIMESTAMP	TIMESTAMPTZ	When this event was posted.

EVENT_CLEARED_TIMESTAMP	TIMESTAMPTZ	When this event was cleared. **Note:** You can also query the ***ACTIVE_EVENTS*** (page 990) system table to see events that have not been cleared.
EVENT_EXPIRATION	TIMESTAMPTZ	Time at which this event expires. If the same event is posted again prior to its expiration time, this field gets updated to a new expiration time.
EVENT_CODE_DESCRIPTION	VARCHAR	Brief description of the event and details pertinent to the specific situation.
EVENT_PROBLEM_DESCRIPTION	VARCHAR	Generic description of the event.

Permissions

Must be a superuser.

Notes

For details about where HP Vertica posts events, see Monitoring Vertica in the Administrator's Guide

Example

```
-[ RECORD 1 ]-------------+-------------------------------------------------
----------------------------------
node_name                 | v_myvdb_node0001
event_code                | 0
event_id                  | 0
event_severity            | Warning
event_posted_timestamp    | 2012-03-22 11:05:32.002682-04
event_cleared_timestamp   |
event_expiration          | 2012-03-22 17:07:32.002681-04
event_code_description    | Low Disk Space
event_problem_description | Warning: Low disk space detected (80% in use)
-[ RECORD 2 ]-------------+-------------------------------------------------
----------------------------------
node_name                 | v_myvdb_node0001
event_code                | 6
event_id                  | 6
event_severity            | Informational
event_posted_timestamp    | 2012-03-22 11:03:49.055705-04
event_cleared_timestamp   |
event_expiration          | 2080-04-09 13:17:56.055704-05
event_code_description    | Node State Change
event_problem_description | Changing node v_myvdb_node0001 startup state to UP
-[ RECORD 3 ]-------------+-------------------------------------------------
----------------------------------
node_name                 | v_myvdb_node0001
event_code                | 6
event_id                  | 7
event_severity            | Informational
event_posted_timestamp    | 2012-03-22 11:03:49.03912-04
event_cleared_timestamp   | 2012-03-22 11:03:49.05626-04
event_expiration          | 2080-04-09 13:17:56.039117-05
event_code_description    | Node State Change
event_problem_description | Changing node v_myvdb_node0001 startup state to REA
DY
-[ RECORD 4 ]-------------+-------------------------------------------------
----------------------------------
node_name                 | v_myvdb_node0001
```

```
event_code                | 6
event_id                  | 8
event_severity            | Informational
event_posted_timestamp    | 2012-03-12 10:26:16.731482-04
event_cleared_timestamp   |
event_expiration          | 2080-03-30 12:40:23.731482-05
event_code_description    | Node State Change
event_problem_description | Changing node v_myvdb_node0001 startup state to SHU
TDOWN
-[ RECORD 5 ]-------------+-----------------------------------
```

See Also

ACTIVE_EVENTS (page 990)

Monitoring Vertica in the Administrator's Guide

NETWORK_INTERFACES

Provides information about network interfaces on all HP Vertica nodes.

Column Name	Data Type	Description
NODE_ID	INTEGER	Unique identifier for the node that recorded the row.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
INTERFACE	VARCHAR	Network interface name.
IP_ADDRESS	VARCHAR	IP address for this interface.
SUBNET	VARCHAR	IP subnet for this interface.
MASK	VARCHAR	IP network mask for this interface.
BROADCAST_ADDRESS	VARCHAR	IP broadcast address for this interface.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> SELECT * FROM network_interfaces;

      node_id      |    node_name     | interface |   ip_address   |    subnet    |     mask
| broadcast_address
-------------------+------------------+-----------+----------------+--------------+-------------
-+------------------
 45035996273704972 | v_gry22_node0001 | lo        | 127.0.0.1      | 127.0.0.0    | 255.0.0.X     |
127.0.0.1
 45035996273704972 | v_gry22_node0001 | eth0      | 10.20.XX.XXX   | 10.20.XX.X   | 255.255.255.X |
10.20.XX.XXX
 45035996273704972 | v_gry22_node0001 | eth1      | 192.168.XX.XXX | 192.168.XX.0 | 255.255.255.X |
192.168.XX.XXX
```

```
 45035996273713380 | v_gry22_node0002 | lo        | 127.0.0.1      | 127.0.0.0    | 255.0.0.0     |
127.0.0.1
 45035996273713380 | v_gry22_node0002 | eth1      | 10.20.XX.XXX   | 10.20.XX.X   | 255.255.255.X |
10.20.XX.XXX
 45035996273713380 | v_gry22_node0002 | eth2      | 192.168.XX.XXX | 192.168.XX.0 | 255.255.255.X |
192.168.XX.XXX
 45035996273713442 | v_gry22_node0003 | lo        | 127.0.0.1      | 127.0.0.0    | 255.0.0.X     |
127.0.0.1
 45035996273713442 | v_gry22_node0003 | eth1      | 10.20.XX.XXX   | 10.20.XX.X   | 255.255.255.X |
10.20.XX.XXX
 45035996273713442 | v_gry22_node0003 | eth2      | 192.168.XX.XXX | 192.168.XX.0 | 255.255.255.X |
192.168.XX.XXX
 45035996273713504 | v_gry22_node0004 | lo        | 127.0.0.1      | 127.0.0.0    | 255.0.0.X     |
127.0.0.1
 45035996273713504 | v_gry22_node0004 | eth1      | 10.20.XX.XXX   | 10.20.XX.X   | 255.255.255.X |
10.20.XX.XXX
 45035996273713504 | v_gry22_node0004 | eth2      | 192.168.XX.XXX | 192.168.XX.0 | 255.255.255.0 |
192.168.XX.XXX
(12 rows)
```

NETWORK_USAGE

Provides network bandwidth usage history on the system. This is useful for determining if HP Vertica is using a large percentage of its available network bandwidth.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
START_TIME	TIMESTAMP	Beginning of history interval.
END_TIME	TIMESTAMP	End of history interval.
TX_KBYTES_PER_SEC	FLOAT	Counter history of outgoing (transmitting) usage in kilobytes per second.
RX_KBYTES_PER_SEC	FLOAT	Counter history of incoming (receiving) usage in kilobytes per second.

Permissions

Must be a superuser.

Example

```
=> SELECT * FROM network_usage;
```

```
 node_name |          start_time           |           end_time            | tx_kbytes_per_sec |
rx_kbytes_per_sec
-----------+-+-----------------------------+-------------------------------+-------------------+-
-----------------
 initiator | 2012-03-25 15:23:38.000834-04 | 2012-03-25 15:24:00.006087-04 |              0.22 |
0.86
 e0        | 2012-03-25 15:23:37.003062-04 | 2012-03-25 15:24:00.003448-04 |              0.21 |
0.83
 e1        | 2012-03-25 15:24:00.002963-04 | 2012-03-25 15:25:00.002497-04 |              0.81 |
0.45
```

```
 e1        | 2012-03-25 15:23:37.000989-04 | 2012-03-25 15:24:00.002963-04 |              0.21 |
0.83
 e0        | 2012-03-25 15:24:00.003448-04 | 2012-03-25 15:25:00.004897-04 |              0.81 |
0.45
 initiator | 2012-03-25 15:24:00.006087-04 | 2012-03-25 15:25:00.007525-04 |              0.81 |
0.45
(6 rows)
```

NODE_RESOURCES

Provides a snapshot of the node. This is useful for regularly polling the node with automated tools or scripts.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
HOST_NAME	VARCHAR	The hostname associated with a particular node.
PROCESS_SIZE_BYTES	INTEGER	The total size of the program.
PROCESS_RESIDENT_SET_SIZE_BYTES	INTEGER	The total number of pages that the process has in memory.
PROCESS_SHARED_MEMORY_SIZE_BYTES	INTEGER	The amount of shared memory used.
PROCESS_TEXT_MEMORY_SIZE_BYTES	INTEGER	The total number of text pages that the process has in physical memory. This does not include any shared libraries.
PROCESS_DATA_MEMORY_SIZE_BYTES	INTEGER	The amount of physical memory, in pages, used for performing processes. This does not include the executable code.
PROCESS_LIBRARY_MEMORY_SIZE_BYTES	INTEGER	The total number of library pages that the process has in physical memory.
PROCESS_DIRTY_MEMORY_SIZE_BYTES	INTEGER	The number of pages that have been modified since they were last written to disk.

Example

Query the NODE_RESOURCES table:

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM NODE_RESOURCES;

-[ RECORD 1 ]----------------------+-------------------
node_name                          | v_vmartdb_node01
host_name                          | myhost-s1
process_size_bytes                 | 2001829888
process_resident_set_size_bytes    | 40964096
process_shared_memory_size_bytes   | 16543744
```

```
process_text_memory_size_bytes    | 46649344
process_data_memory_size_bytes    | 0
process_library_memory_size_bytes | 1885351936
process_dirty_memory_size_bytes   | 0
-[ RECORD 2 ]---------------------+------------------
node_name                         | v_vmartdb_node02
host_name                         | myhost-s2
process_size_bytes                | 399822848
process_resident_set_size_bytes   | 31453184
process_shared_memory_size_bytes  | 10862592
process_text_memory_size_bytes    | 46649344
process_data_memory_size_bytes    | 0
process_library_memory_size_bytes | 299356160
process_dirty_memory_size_bytes   | 0
-[ RECORD 3 ]---------------------+------------------
node_name                         | v_vmartdb_node03
host_name                         | myhost-s3
process_size_bytes                | 399822848
process_resident_set_size_bytes   | 31100928
process_shared_memory_size_bytes  | 10735616
process_text_memory_size_bytes    | 46649344
process_data_memory_size_bytes    | 0
process_library_memory_size_bytes | 299356160
process_dirty_memory_size_bytes   | 0
-[ RECORD 4 ]---------------------+------------------
node_name                         | v_vmartdb_node04
host_name                         | myhost-s4
process_size_bytes                | 466923520
process_resident_set_size_bytes   | 31309824
process_shared_memory_size_bytes  | 10735616
process_text_memory_size_bytes    | 46649344
process_data_memory_size_bytes    | 0
process_library_memory_size_bytes | 366456832
process_dirty_memory_size_bytes   | 0
```

NODE_STATES

Monitors node recovery state-change history on the system.

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMPTZ	Time when HP Vertica recorded the event.
NODE_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the node.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.

NODE_STATE	VARCHAR	Shows the node's state. Can be one of: ▪ UP ▪ READY ▪ UNSAFE ▪ SHUTDOWN ▪ RECOVERING

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

Query the NODE_STATES table:

```
=> SELECT * FROM node_states;

        event_timestamp        |      node_id      |     node_name      |  node_state
-------------------------------+-------------------+--------------------+--------------
 2012-12-09 06:36:22.10059-05  | 45035996273704980 | v_onenode_node0001 | UP
 2012-12-09 06:36:22.10044-05  | 45035996273704980 | v_onenode_node0001 | READY
 2012-12-09 06:36:22.081961-05 | 45035996273704980 | v_onenode_node0001 | INITIALIZING
 2012-12-09 05:29:13.637828-05 | 45035996273704980 | v_onenode_node0001 | DOWN
 2012-12-09 05:29:12.739187-05 | 45035996273704980 | v_onenode_node0001 | SHUTDOWN
 2012-12-08 13:50:21.074688-05 | 45035996273704980 | v_onenode_node0001 | UP
 2012-12-08 13:50:21.074544-05 | 45035996273704980 | v_onenode_node0001 | READY
 2012-12-08 13:50:21.073529-05 | 45035996273704980 | v_onenode_node0001 | INITIALIZING
(8 rows)
```

PARTITION_REORGANIZE_ERRORS

Monitors all background partitioning tasks, and if HP Vertica encounters an error, creates an entry in this table with the appropriate information. Does not log repartitioning tasks that complete successfully.

Column Name	Data Type	Description
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
USER_NAME	VARCHAR	Name of the user who received the error at the time HP Vertica recorded the session.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
TABLE_NAME	VARCHAR	Name of the partitioned table.

PROJECTION_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed.
MESSAGE	VARCHAR	Textual output of the error message.
HINT	VARCHAR	Actionable hint about the error.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

PARTITION_STATUS

Shows, for each projection of each partitioned table, the fraction of its data that is actually partitioned according to the current partition expression. When the partitioning of a table is altered, the value in partition_reorganize_percent for each of its projections drops to zero and goes back up to 100 when all the data is repartitioned.

Column Name	Data Type	Description
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
TABLE_SCHEMA	VARCHAR	Name of the schema that contains the partitioned table.
TABLE_NAME	VARCHAR	Table name that is partitioned.
TABLE_ID	INTEGER	Unique numeric ID assigned by the HP Vertica, which identifies the table.
PROJECTION_SCHEMA	VARCHAR	Schema containing the projection.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed.
PARTITION_REORGANIZE_PERCENT	INTEGER	For each projection, drops to zero and goes back up to 100 when all the data is repartitioned after the partitioning of a table has been altered. Ideally all rows will show 100 (%).

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> SELECT * FROM partition_status;
```

```
-[ RECORD 1 ]----------------+----------------------
projection_id                | 45035996281788238
table_schema                 | public
table_name                   | lineorder
table_id                     | 45035996281787664
projection_schema            | public
projection_name              | VLINEORDER
partition_reorganize_percent | 100
-[ RECORD 2 ]----------------+----------------------
projection_id                | 45035996281788312
table_schema                 | public
table_name                   | lineorder
table_id                     | 45035996281787664
projection_schema            | public
projection_name              | VLINEORDER1
partition_reorganize_percent | 100
-[ RECORD 3 ]----------------+----------------------
projection_id                | 45035996281788752
table_schema                 | public
table_name                   | lineorder
table_id                     | 45035996281787664
projection_schema            | public
projection_name              | VLINEORDER2
partition_reorganize_percent | 100
-[ RECORD 4 ]----------------+----------------------
projection_id                | 45035996281788822
table_schema                 | public
table_name                   | lineorder
table_id                     | 45035996281787664
projection_schema            | public
projection_name              | VLINEORDER1_BUD
partition_reorganize_percent | 100
-[ RECORD 5 ]----------------+----------------------
```

PARTITIONS

Displays partition metadata, one row per partition key, per ROS container.

Column Name	Data Type	Description
PARTITION_KEY	VARCHAR	The partition value(s).
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the projection.
TABLE_SCHEMA	VARCHAR	The schema name for which information is listed.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed.

ROS_ID	VARCHAR	A unique numeric ID assigned by the HP Vertica catalog, which identifies the ROS container.
ROS_SIZE_BYTES	INTEGER	The ROS container size in bytes.
ROS_ROW_COUNT	INTEGER	Number of rows in the ROS container.
NODE_NAME	VARCHAR	Node where the ROS container resides.
DELETED_ROW_COUNT	INTEGER	The number of rows in the partition.
LOCATION_LABEL	VARCHAR	The location label of the default storage location.

Notes

- A many-to-many relationship exists between partitions and ROS containers. PARTITIONS displays information in a denormalized fashion.
- To find the number of ROS containers having data of a specific partition, aggregate PARTITIONS over the `partition_key` column.
- To find the number of partitions stored in a ROS container, aggregate PARTITIONS over the `ros_id` column.

Example

Given a projection named p1, with three ROS containers, the PARTITIONS function returns three rows:

```
=> SELECT PARTITION_KEY, PROJECTION_NAME, ROS_ID, ROS_SIZE_BYTES, ROS_ROW_COUNT, NODE_NAME FROM
partitions;

 PARTITION_KEY | PROJECTION_NAME  |      ROS_ID       | ROS_SIZE_BYTES | ROS_ROW_COUNT |
NODE_NAME
---------------+------------------+-------------------+----------------+---------------+---------
-------------
 2008          | trade_p_node0001 | 45035996273740461 |             90 |             1 | node0001
 2007          | trade_p_node0001 | 45035996273740477 |             99 |             2 | node0001
 2006          | trade_p_node0001 | 45035996273740493 |             99 |             2 | node0001
(3 rows)
```

PROCESS_SIGNALS

Returns a history of signals that were received and handled by the HP Vertica process.

Column Name	Data Type	Description
SIGNAL_TIMESTAMP	TIMESTAMPTZ	Time when HP Vertica recorded the signal.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.

SIGNAL_NUMBER	INTEGER	Signal number.
SIGNAL_CODE	INTEGER	Signal code.
SIGNAL_PID	INTEGER	Linux process identifier of the signal.
SIGNAL_UID	INTEGER	Process ID of sending process.
SIGNAL_ADDRESS	INTEGER	Address at which fault occurred.

Permissions

Must be a superuser

Example

```
=> SELECT * FROM process_signals;
       signal_timestamp        | node_name | signal_number | signal_code | signal_pid | signal_uid
| signal_address
-------------------------------+-----------+---------------+-------------+------------+----------
--+----------------
 2012-02-14 13:15:23.608359-04 | initiator |            15 |           0 |      31721 |        500
|  2147483679721
(1 row)
```

PROJECTION_RECOVERIES

Retains history about projection recoveries. Since HP Vertica adds an entry per recovery plan, a projection/node pair could appear multiple times in the output.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is recovering or has recovered the corresponding projection.
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	Name of the projection that is being or has been recovered on the corresponding node.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any. TRANSACTION_ID initializes as NO_TRANSACTION with a value of 0. HP Vertica will ignore the recovery query and keep (0) if there's no action to take (no data in the table, etc). When no recovery transaction starts, `ignored` value appears in this table's STATUS column.

STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
METHOD	VARCHAR	Recovery method that HP Vertica chooses. Possible values are: ▪ incremental ▪ incremental-replay-delete ▪ split ▪ recovery-by-container
STATUS	VARCHAR	Current projection-recovery status on the corresponding node. STATUS can be "queued," which indicates a brief period between the time the query is prepared and when it runs. Possible values are: ▪ queued ▪ running ▪ finished ▪ ignored ▪ error-retry ▪ error-fatal
PROGRESS	INTEGER	An estimate (value in the range [0,100]) of percent complete for the recovery task described by this information. **Note:** The actual amount of time it takes to complete a recovery task depends on a number of factors, including concurrent workloads and characteristics of the data; therefore, accuracy of this estimate can vary. The PROGRESS column value is NULL after the task completes.

DETAIL	VARCHAR	More detailed information about PROGRESS. The values returned for this column depend on the type of recovery plan: ▪ General recovery plans – value displays the estimated progress, as a percent, of the three primary parts of the plan: `Scan`, `Sort`, and `Write`. ▪ Recovery-by-container plans – value begins with `CopyStorage:` and is followed by the number of bytes copied over the total number of bytes to copy. ▪ Replay delete plans – value begins with `Delete:` and is followed by the number of deletes replayed over an estimate of the total number of deletes to replay. The DETAIL column value becomes `NULL` after the recovery plan completes.
START_TIME	TIMESTAMPTZ	Time the recovery task described by this information started.
END_TIME	TIMESTAMPTZ	Time the recovery task described by this information ended.
RUNTIME_PRIORITY	VARCHAR	Determines the amount of runtime resources (CPU, I/O bandwidth) the Resource Manager should dedicate to running queries in the resource pool. Valid values are: ▪ HIGH ▪ MEDIUM ▪ LOW

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Notes

If you are interested in monitoring recovery progress when recovery seems to be taking a while, note that you cannot query system tables table during cluster recovery; the cluster must be UP to accept connections.

Example

```
=> SELECT * FROM recovery_status;

 node_name | recover_epoch | recovery_phase    | splits_completed | splits_total |
historical_completed | historical_total | current_completed | current_total | is_running
-----------+---------------+-------------------+------------------+--------------+---------------
-------+-----------------+-------------------+---------------+------------
 node01    |               |                   | 0                | 0            | 0
| 0                | 0                 | 0             | f
```

```
 node02    | 0            | historical pass 1 | 0                | 0            | 0
| 0               | 0                | 0              | t
 node03    | 1            | current           | 0                | 0            | 0
| 0               | 0                | 0              | f


=> SELECT * FROM projection_recoveries;

 node_name | projection_id     | projection_name | transaction_id    | statement_id | method
| status  | progress | detail                        | start_time                    | ...
-----------+-------------------+-----------------+-------------------+--------------+------------
--------------+---------+----------+-------------------------------+---------------------------
---+----------
 node02    | 45035996273736792 | public.t_p1     | 49539595901075489 | 1            | incremental
| running | 69       | Scan:100% Sort:100% Write:87% | 2011-10-04 14:41:51.354757-04 | ...
 node02    | 45035996273736768 | public.t_p0     | 49539595901075490 | 1            |
incremental-replay-delete | running | 71       | Delete:0/6563442              | 2011-10-04
14:41:51.353797-04 | ...
 node02    | 45035996273736852 | public.tt_p0    | 49539595901075598 | 1            |
recovery-by-container     | running | 76       | CopyStorage:27427070/35938922 | 2011-10-04
14:44:50.525465-04 | ...
```

See Also

RECOVERY_STATUS (page 1079)

PROJECTION_REFRESHES

Provides information about refresh operations for projections.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node where the refresh was initiated.
PROJECTION_SCHEMA	VARCHAR	Name of the schema associated with the projection.
PROJECTION_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	Name of the projection that is targeted for refresh.
ANCHOR_TABLE_NAME	VARCHAR	Name of the projection's associated anchor table.
REFRESH_STATUS	VARCHAR	Status of the projection: ▪ Queued — Indicates that a projection is queued for refresh. ▪ Refreshing — Indicates that a refresh for a projection is in process. ▪ Refreshed — Indicates that a refresh for a projection has successfully completed. ▪ Failed — Indicates that a refresh for a

		projection did not successfully complete.
REFRESH_PHASE	VARCHAR	Indicates how far the refresh has progressed: ▪ Historical – Indicates that the refresh has reached the first phase and is refreshing data from historical data. This refresh phase requires the most amount of time. ▪ Current – Indicates that the refresh has reached the final phase and is attempting to refresh data from the current epoch. To complete this phase, refresh must be able to obtain a lock on the table. If the table is locked by some other transaction, refresh is put on hold until that transaction completes. The ***LOCKS*** (page 1037) system table is useful for determining if a refresh has been blocked on a table lock. To determine if a refresh has been blocked, locate the term "refresh" in the transaction description. A refresh has been blocked when the scope for the refresh is REQUESTED and one or more other transactions have acquired a lock on the table. **Note:** The REFRESH_PHASE field is NULL until the projection starts to refresh and is NULL after the refresh completes.
REFRESH_METHOD	VARCHAR	Method used to refresh the projection: ▪ Buddy – Uses the contents of a buddy to refresh the projection. This method maintains historical data. This enables the projection to be used for historical queries. ▪ Scratch – Refreshes the projection without using a buddy. This method does not generate historical data. This means that the projection cannot participate in historical queries from any point before the projection was refreshed. ▪ Rebalance – If the projection is segmented it is refreshed from scratch; if unsegmented it is refreshed from buddy.
REFRESH_FAILURE_COUNT	INTEGER	Number of times a refresh failed for the projection. FAILURE_COUNT does not indicate whether the projection was eventually

		refreshed. See REFRESH_STATUS to determine how the refresh operation is progressing.
SESSION_ID	VARCHAR	Unique numeric ID assigned by the HP Vertica catalog, which identifies the refresh session.
REFRESH_START	TIMESTAMPTZ	Time the projection refresh started (provided as a timestamp).
REFRESH_DURATION_SEC	INTERVAL SECOND (0)	Length of time that the projection refresh ran in seconds.
IS_EXECUTING	BOOLEAN	Distinguishes between active (*t*) and completed (*f*) refresh operations.
RUNTIME_PRIORITY	VARCHAR	Determines the amount of run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to running queries in the resource pool. Valid values are: ▪ HIGH ▪ MEDIUM ▪ LOW

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Notes

- Information about a refresh operation—whether successful or unsuccessful—is maintained in the PROJECTION_REFRESHES system table until either the ***CLEAR_PROJECTION_REFRESHES()*** (page 455) function is executed or the storage quota for the table is exceeded.
- Tables and projections can be dropped while a query runs against them. The query continues to run, even after the drop occurs. Only when the query finishes does it notice the drop, which could cause a rollback. The same is true for refresh queries. PROJECTION_REFRESHES, therefore, could report that a projection failed to be refreshed before the refresh query completes. In this case, the REFRESH_DURATION_SEC column continues to increase until the refresh query completes.

Example

The following command purges projection refresh history from the PROJECTION_REFRESHES table:

```
=> SELECT clear_projection_refreshes();
 clear_projection_refreshes
----------------------------
 CLEAR
(1 row)
```

Only the rows where the `IS_EXECUTING` column equals false are cleared.

See Also

CLEAR_PROJECTION_REFRESHES (page 455)

Clearing PROJECTION_REFRESHES History in the Administrator's Guide

PROJECTION_STORAGE

Monitors the amount of disk storage used by each projection on each node.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
PROJECTION_ID	VARCHAR	A unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed.
PROJECTION_SCHEMA	VARCHAR	The name of the schema associated with the projection.
PROJECTION_COLUMN_COUNT	INTEGER	The number of columns in the projection.
ROW_COUNT	INTEGER	The number of rows in the table's projections, including any rows marked for deletion.
USED_BYTES	INTEGER	The number of bytes of disk storage used by the projection.
WOS_ROW_COUNT	INTEGER	The number of WOS rows in the projection.
WOS_USED_BYTES	INTEGER	The number of WOS bytes in the projection.
ROS_ROW_COUNT	INTEGER	The number of ROS rows in the projection.
ROS_USED_BYTES	INTEGER	The number of ROS bytes in the projection.
ROS_COUNT	INTEGER	The number of ROS containers in the projection.
ANCHOR_TABLE_NAME	VARCHAR	The associated table name for which information is listed.
ANCHOR_TABLE_SCHEMA	VARCHAR	The associated table schema for which information is listed.

Notes

Projections that have no data never have full statistics. Querying this system table lets you see if your projection contains data.

Example

Query the system table:

```
=> SELECT * FROM projection_storage;
```

```
-[ RECORD 1 ]-----------+--------------------
node_name               | v_onenode_node0001
projection_id           | 45035996273718838
projection_name         | trades_p
projection_schema       | public
projection_column_count | 4
row_count               | 1
used_bytes              | 147
wos_row_count           | 0
wos_used_bytes          | 0
ros_row_count           | 1
ros_used_bytes          | 147
ros_count               | 1
anchor_table_name       | trades
anchor_table_schema     | public
anchor_table_id         | 45035996273718836


=> SELECT projection_name, row_count, ros_used_bytes, used_bytes
   FROM PROJECTION_STORAGE WHERE projection_schema = 'store' ORDER BY used_bytes;
                    projection_name                      | row_count | ros_used_bytes |
used_bytes
---------------------------------------------------------+-----------+----------------+-------
-----
 store_dimension_DBD_4_seg_vmartdb_design_vmartdb_design    |        53 |           2791 |
2791
 store_dimension_DBD_29_seg_vmartdb_design_vmartdb_design   |        53 |           2791 |
2791
 store_dimension_DBD_29_seg_vmartdb_design_vmartdb_design   |        56 |           2936 |
2936
 store_dimension_DBD_4_seg_vmartdb_design_vmartdb_design    |        56 |           2936 |
2936
 store_dimension_DBD_4_seg_vmartdb_design_vmartdb_design    |        68 |           3360 |
3360
 store_dimension_DBD_29_seg_vmartdb_design_vmartdb_design   |        68 |           3360 |
3360
 store_dimension_DBD_29_seg_vmartdb_design_vmartdb_design   |        73 |           3579 |
3579
 store_dimension_DBD_4_seg_vmartdb_design_vmartdb_design    |        73 |           3579 |
3579
 store_orders_fact_DBD_31_seg_vmartdb_design_vmartdb_design |     53974 |        1047782 |
1047782
 store_orders_fact_DBD_6_seg_vmartdb_design_vmartdb_design  |     53974 |        1047782 |
1047782
 store_orders_fact_DBD_6_seg_vmartdb_design_vmartdb_design  |     66246 |        1285786 |
1285786
 store_orders_fact_DBD_31_seg_vmartdb_design_vmartdb_design |     66246 |        1285786 |
1285786
 store_orders_fact_DBD_31_seg_vmartdb_design_vmartdb_design |     71909 |        1395258 |
1395258
 store_orders_fact_DBD_6_seg_vmartdb_design_vmartdb_design  |     71909 |        1395258 |
1395258
 store_orders_fact_DBD_6_seg_vmartdb_design_vmartdb_design  |    107871 |        2090941 |
2090941
 store_orders_fact_DBD_31_seg_vmartdb_design_vmartdb_design |    107871 |        2090941 |
2090941
 store_sales_fact_DBD_5_seg_vmartdb_design_vmartdb_design   |   1235825 |       24285740 |
24285740
 store_sales_fact_DBD_30_seg_vmartdb_design_vmartdb_design  |   1235825 |       24285740 |
24285740
 store_sales_fact_DBD_30_seg_vmartdb_design_vmartdb_design  |   1245865 |       24480819 |
24480819
 store_sales_fact_DBD_5_seg_vmartdb_design_vmartdb_design   |   1245865 |       24480819 |
24480819
```

```
 store_sales_fact_DBD_5_seg_vmartdb_design_vmartdb_design   |   1249547 |       24551817 |
24551817
 store_sales_fact_DBD_30_seg_vmartdb_design_vmartdb_design  |   1249547 |       24551817 |
24551817
 store_sales_fact_DBD_30_seg_vmartdb_design_vmartdb_design  |   1268763 |       24930549 |
24930549
 store_sales_fact_DBD_5_seg_vmartdb_design_vmartdb_design   |   1268763 |       24930549 |
24930549
(24 rows)
```

The following command returns the per-node storage used by a segmented table:

```
SELECT anchor_table_name, node_name,
  SUM(ros_used_bytes)/1024/1024 AS MB,
  SUM(ros_row_count) AS Rows
FROM projection_storage GROUP BY 1,2 ORDER BY 1,2;
```

anchor_table_name	node_name	MB	Rows
call_center_dimension	v_vmartdb_node0001	0	98
call_center_dimension	v_vmartdb_node0002	0	101
call_center_dimension	v_vmartdb_node0003	0	102
call_center_dimension	v_vmartdb_node0004	0	99
customer_dimension	v_vmartdb_node0001	0	24030
customer_dimension	v_vmartdb_node0002	0	1648
customer_dimension	v_vmartdb_node0003	0	25970
customer_dimension	v_vmartdb_node0004	1	48352
date_dimension	v_vmartdb_node0001	0	910
date_dimension	v_vmartdb_node0002	0	913
date_dimension	v_vmartdb_node0003	0	916
date_dimension	v_vmartdb_node0004	0	913
employee_dimension	v_vmartdb_node0001	0	4998
employee_dimension	v_vmartdb_node0002	0	4999
employee_dimension	v_vmartdb_node0003	0	5002
employee_dimension	v_vmartdb_node0004	0	5001
inventory_fact	v_vmartdb_node0001	0	150414
inventory_fact	v_vmartdb_node0002	0	149736
inventory_fact	v_vmartdb_node0003	0	149586
inventory_fact	v_vmartdb_node0004	0	150264
online_page_dimension	v_vmartdb_node0001	0	499
online_page_dimension	v_vmartdb_node0002	0	500
online_page_dimension	v_vmartdb_node0003	0	501
online_page_dimension	v_vmartdb_node0004	0	500
online_sales_fact	v_vmartdb_node0001	59	4941898
online_sales_fact	v_vmartdb_node0002	59	5024898
online_sales_fact	v_vmartdb_node0003	59	5058102
online_sales_fact	v_vmartdb_node0004	59	4975102
product_dimension	v_vmartdb_node0001	1	103185
product_dimension	v_vmartdb_node0002	1	100428
product_dimension	v_vmartdb_node0003	1	76815
product_dimension	v_vmartdb_node0004	1	79572
promotion_dimension	v_vmartdb_node0001	0	499
promotion_dimension	v_vmartdb_node0002	0	500
promotion_dimension	v_vmartdb_node0003	0	501
promotion_dimension	v_vmartdb_node0004	0	500
rostab	v_vmartdb_node0001	0	0
rostab	v_vmartdb_node0002	0	0
rostab	v_vmartdb_node0003	0	0
rostab	v_vmartdb_node0004	0	0
shipping_dimension	v_vmartdb_node0001	0	49
shipping_dimension	v_vmartdb_node0002	0	50
shipping_dimension	v_vmartdb_node0003	0	51
shipping_dimension	v_vmartdb_node0004	0	50
store_dimension	v_vmartdb_node0001	0	123
store_dimension	v_vmartdb_node0002	0	125
store_dimension	v_vmartdb_node0003	0	127
store_dimension	v_vmartdb_node0004	0	125
store_orders_fact	v_vmartdb_node0001	2	147768

```
 store_orders_fact     | v_vmartdb_node0002 |  2 |  149759
 store_orders_fact     | v_vmartdb_node0003 |  2 |  152232
 store_orders_fact     | v_vmartdb_node0004 |  2 |  150241
 store_sales_fact      | v_vmartdb_node0001 | 46 | 2501318
 store_sales_fact      | v_vmartdb_node0002 | 47 | 2512691
 store_sales_fact      | v_vmartdb_node0003 | 46 | 2498682
 store_sales_fact      | v_vmartdb_node0004 | 46 | 2487309
 vendor_dimension      | v_vmartdb_node0001 |  0 |      48
 vendor_dimension      | v_vmartdb_node0002 |  0 |      48
 vendor_dimension      | v_vmartdb_node0003 |  0 |      52
 vendor_dimension      | v_vmartdb_node0004 |  0 |      52
 warehouse_dimension   | v_vmartdb_node0001 |  0 |      49
 warehouse_dimension   | v_vmartdb_node0002 |  0 |      50
 warehouse_dimension   | v_vmartdb_node0003 |  0 |      51
 warehouse_dimension   | v_vmartdb_node0004 |  0 |      50
(64 rows)
```

See Also

PROJECTIONS (page 961)

ANALYZE_STATISTICS (page 440)

PROJECTION_USAGE

Records information about projections HP Vertica uses in each processed query.

Column Name	Data Type	Description
QUERY_START_TIMESTAMP	TIMESTAMPTZ	Value of query at beginning of history interval.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user at the time HP Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.
IO_TYPE	VARCHAR	Input/output.

PROJECTION_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed.
ANCHOR_TABLE_ID	INTEGER	Unique numeric ID assigned by the HP Vertica, which identifies the anchor table.
ANCHOR_TABLE_SCHEMA	VARCHAR	Name of the schema that contains the anchor table.
ANCHOR_TABLE_NAME	VARCHAR	Name of the projection's associated anchor table.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Examples

```
select * from projection_usage;
     query_start_timestamp     | node_name | user_name |    session_id    | request_id |
transaction_id   | io_type |   projection_id   | projection_name |  anchor_table_id  |
anchor_table_name
-------------------------------+-----------+-----------+------------------+------------+---------
---------+---------+------------------+----------------+------------------+-----------------
-
 2011-09-16 13:38:47.00034-04  | node01    | kelly     | keprl-15875:0x76 |         16 |
45035996273705736 |       1 | 45035996273737044 | t_super         | 45035996273737042 | t
 2011-09-16 13:38:52.920688-04 | node01    | kelly     | keprl-15875:0x76 |         18 |
45035996273705739 |       2 | 45035996273737044 | t_super         | 45035996273737042 | t
 2011-09-16 13:38:52.987128-04 | node01    | kelly     | keprl-15875:0x76 |         20 |
45035996273705743 |       1 | 45035996273737044 | t_super         | 45035996273737042 | t
(3 rows)
```

QUERY_EVENTS

Returns information about query planning, optimization, and execution events.

Column Name	Data Type	Description
EVENT_TIMESTAMP	TIMESTAMPTZ	Time when HP Vertica recorded the event.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_ID	INTEGER	Identifier of the user for the query event.
USER_NAME	VARCHAR	Name of the user for which HP Vertica lists query information at the time it recorded the session.

SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.
EVENT_CATEGORY	VARCHAR	Category of event: OPTIMIZATION or EXECUTION.
EVENT_TYPE	VARCHAR	Type of event. Examples include but are not limited to: ▪ PREDICATE OUTSIDE HISTOGRAM ▪ NO HISTOGRAM ▪ MEMORY LIMIT HIT ▪ GROUP_BY_SPILLED ▪ JOIN_SPILLED ▪ PARTITIONS_ELIMINATED ▪ MERGE_CONVERTED_TO_UNION
EVENT_DESCRIPTION	VARCHAR	Generic description of the event.
OPERATOR_NAME	VARCHAR	Name of the Execution Engine component that generated the event, if applicable; for example, NetworkSend. Values from the OPERATOR_NAME and PATH_ID columns let you tie a query event back to a particular operator in the EXPLAIN plan. If the event did not come from a specific operator, the OPERATOR_NAME column is NULL.
PATH_ID	INTEGER	Unique identifier that HP Vertica assigns to a query operation or path in a query plan. If the event did not come from a specific operator, the PATH_ID column is NULL. See ***EXECUTION_ENGINE_PROFILES*** (page 1021) for more information.
OBJECT_ID	INTEGER	Object identifier (such as projection or table) to which the event refers.
EVENT_DETAILS	VARCHAR	Free-form text describing the specific event.
SUGGESTED_ACTION	VARCHAR	Suggested user action, if any is available.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Examples

```
=> \x
Expanded display is on.
=> SELECT * FROM query_events;

-[ RECORD 1 ]-----+-----------------------------------------------------------
event_timestamp   | 2012-12-09 23:59:00.174464-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-31427:0x82fb
request_id        | 1
transaction_id    | 45035996273711993
statement_id      | 1
event_category    | EXECUTION
event_type        | RLE_OVERRIDDEN
event_description | Compressed execution will not be used on some columns,
                    because the average run counts are not large enough.
operator_name     | Scan
path_id           | 2
object_id         | 45035996273718840
event_details     | Column public.trades_p.stock will not be processed using RLE
.
suggested_action  |
-[ RECORD 2 ]-----+-----------------------------------------------------------
event_timestamp   | 2012-12-09 06:37:01.518944-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-31427:0x19
request_id        | 2
transaction_id    | 45035996273708310
statement_id      | 2
event_category    | EXECUTION
event_type        | RLE_OVERRIDDEN
event_description | Compressed execution will not be used on some columns,
                    because the average run counts are not large enough.
operator_name     | Scan
path_id           | 1
object_id         | 45035996273718840
event_details     | Column public.trades_p.stock will not be processed using RLE
.
suggested_action  |
-[ RECORD 3 ]-----+-----------------------------------------------------------
event_timestamp   | 2012-12-08 23:59:00.08586-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0x4c7c
request_id        | 1
transaction_id    | 45035996273707118
statement_id      | 1
event_category    | EXECUTION
event_type        | RLE_OVERRIDDEN
event_description | Compressed execution will not be used on some columns,
                    because the average run counts are not large enough.
operator_name     | Scan
path_id           | 2
object_id         | 45035996273718840
event_details     | Column public.trades_p.stock will not be processed using RLE.
```

```
suggested_action  |
-[ RECORD 4 ]-----+-------------------------------------------------------------
event_timestamp   | 2012-12-08 13:55:20.047935-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0xb0
request_id        | 0
transaction_id    | 45035996273705015
statement_id      | 1
event_category    | EXECUTION
event_type        | SMALL_MERGE_REPLACED
event_description | Small StorageMerge replaced with StorageUnion for efficiency
operator_name     | StorageMerge
path_id           |
object_id         | 45035996273718838
event_details     | Projection: public.trades_p
suggested_action  |
-[ RECORD 5 ]-----+-------------------------------------------------------------
event_timestamp   | 2012-12-08 13:54:27.394169-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0x14
request_id        | 10
transaction_id    | 45035996273705005
statement_id      | 1
event_category    | EXECUTION
event_type        | RLE_OVERRIDDEN
event_description | Compressed execution will not be used on some columns,
                    because the average run counts are not large enough.
operator_name     | Scan
path_id           | 2
object_id         | 45035996273718840
event_details     | Column public.trades_p.stock will not be processed using RLE.
suggested_action  |
-[ RECORD 6 ]-----+-------------------------------------------------------------
event_timestamp   | 2012-12-08 13:54:27.39424-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0x14
request_id        | 10
transaction_id    | 45035996273705005
statement_id      | 1
event_category    | EXECUTION
event_type        | SMALL_MERGE_REPLACED
event_description | Small StorageMerge replaced with StorageUnion for efficiency
operator_name     | StorageMerge
path_id           | 2
object_id         | 45035996273718838
event_details     | Projection: public.trades_p
suggested_action  |
-[ RECORD 7 ]-----+-------------------------------------------------------------
event_timestamp   | 2012-12-08 13:54:27.403897-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0x14
request_id        | 10
transaction_id    | 45035996273705005
statement_id      | 2
event_category    | EXECUTION
event_type        | RLE_OVERRIDDEN
event_description | Compressed execution will not be used on some columns,
                    because the average run counts are not large enough.
operator_name     | Scan
path_id           | 2
```

```
object_id         | 45035996273718840
event_details     | Column public.trades_p.stock will not be processed using RLE.
suggested_action  |
-[ RECORD 8 ]-----+-------------------------------------------------------------
event_timestamp   | 2012-12-08 13:54:27.415003-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0x14
request_id        | 10
transaction_id    | 45035996273705005
statement_id      | 3
event_category    | EXECUTION
event_type        | RLE_OVERRIDDEN
event_description | Compressed execution will not be used on some columns,
                    because the average run counts are not large enough.
operator_name     | Scan
path_id           | 2
object_id         | 45035996273718840
event_details     | Column public.trades_p.stock will not be processed using RLE.
suggested_action  |
-[ RECORD 9 ]-----+-------------------------------------------------------------
event_timestamp   | 2012-12-08 13:53:23.425399-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0x14
request_id        | 8
transaction_id    | 45035996273704996
statement_id      | 1
event_category    | EXECUTION
event_type        | RLE_OVERRIDDEN
event_description | Compressed execution will not be used on some columns,
                    because the average run counts are not large enough.
operator_name     | Scan
path_id           | 2
object_id         | 45035996273718840
event_details     | Column public.trades_p.stock will not be processed using RLE.
suggested_action  |
-[ RECORD 10 ]----+-------------------------------------------------------------
event_timestamp   | 2012-12-08 13:52:04.122981-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0x14
request_id        | 6
transaction_id    | 45035996273704983
statement_id      | 1
event_category    | EXECUTION
event_type        | RLE_OVERRIDDEN
event_description | Compressed execution will not be used on some columns,
                    because the average run counts are not large enough.
operator_name     | Scan
path_id           | 2
object_id         | 45035996273718840
event_details     | Column public.trades_p.stock will not be processed using RLE.
suggested_action  |
-[ RECORD 11 ]----+-------------------------------------------------------------
event_timestamp   | 2012-12-08 13:52:04.123235-05
node_name         | v_onenode_node0001
user_id           | 45035996273704962
user_name         | dbadmin
session_id        | doca01.verticacorp.-19852:0x14
request_id        | 6
transaction_id    | 45035996273704983
statement_id      | 1
event_category    | EXECUTION
event_type        | SMALL_MERGE_REPLACED
```

```
event_description | Small StorageMerge replaced with StorageUnion for efficiency
operator_name     | StorageMerge
path_id           | 2
object_id         | 45035996273718838
event_details     | Projection: public.trades_p
suggested_action  |
```

See also

EXECUTION_ENGINE_PROFILES (page 1021)

QUERY_PLAN_PROFILES (page 1069)

QUERY_METRICS

Monitors the sessions and queries running on each node.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
ACTIVE_USER_SESSION_COUNT	INTEGER	The number of active user sessions (connections).
ACTIVE_SYSTEM_SESSION_COUNT	INTEGER	The number of active system sessions.
TOTAL_USER_SESSION_COUNT	INTEGER	The total number of user sessions.
TOTAL_SYSTEM_SESSION_COUNT	INTEGER	The total number of system sessions.
TOTAL_ACTIVE_SESSION_COUNT	INTEGER	The total number of active user and system sessions.
TOTAL_SESSION_COUNT	INTEGER	The total number of user and system sessions.
RUNNING_QUERY_COUNT	INTEGER	The number of queries currently running.
EXECUTED_QUERY_COUNT	INTEGER	The total number of queries that ran.

Notes

Totals get reset each time you restart the database.

Example

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM QUERY_METRICS;

-[ RECORD 1 ]---------------+-------------------
node_name                   | v_vmartdb_node01
active_user_session_count   | 1
active_system_session_count | 2
total_user_session_count    | 2
total_system_session_count  | 6248
total_active_session_count  | 3
```

```
total_session_count         | 6250
running_query_count         | 1
executed_query_count        | 42
-[ RECORD 2 ]---------------+-------------------
node_name                   | v_vmartdb_node02
active_user_session_count   | 1
active_system_session_count | 2
total_user_session_count    | 2
total_system_session_count  | 6487
total_active_session_count  | 3
total_session_count         | 6489
running_query_count         | 0
executed_query_count        | 0
-[ RECORD 3 ]---------------+-------------------
node_name                   | v_vmartdb_node03
active_user_session_count   | 1
active_system_session_count | 2
total_user_session_count    | 2
total_system_session_count  | 6489
total_active_session_count  | 3
total_session_count         | 6491
running_query_count         | 0
executed_query_count        | 0
-[ RECORD 4 ]---------------+-------------------
node_name                   | v_vmartdb_node04
active_user_session_count   | 1
active_system_session_count | 2
total_user_session_count    | 2
total_system_session_count  | 6489
total_active_session_count  | 3
total_session_count         | 6491
running_query_count         | 0
executed_query_count        | 0
```

QUERY_PLAN_PROFILES

Provides detailed execution status for queries that are currently running in the system. Output from the table shows the real-time flow of data and the time and resources consumed for each path in each query plan.

Column Name	Data Type	Description
TRANSACTION_ID	INTEGER	An identifier for the transaction within the session if any; otherwise NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID and STATEMENT_ID uniquely identifies a statement within a session;

		these columns are useful for creating joins with other system tables.
PATH_ID	INTEGER	Unique identifier that HP Vertica assigns to a query operation or path in a query plan. Textual representation for this path is output in the PATH_LINE column.
PATH_LINE_INDEX	INTEGER	Each plan path in QUERY_PLAN_PROFILES could be represented with multiple rows. PATH_LINE_INDEX returns the relative line order. You should include the PATH_LINE_INDEX column in the QUERY_PLAN_PROFILES ... ORDER BY clause so rows in the result set appear as they do in EXPLAIN plan output.
PATH_IS_EXECUTING	BOOLEAN	Status of a path in the query plan. True (*t*) if the path has started running, otherwise false.
PATH_IS_COMPLETE	BOOLEAN	Status of a path in the query plan. True (*t*) if the path has finished running, otherwise false.
IS_EXECUTING	BOOLEAN	Status of a running query. True if the query is currently active (*t*), otherwise false (*f*).
RUNNING_TIME	INTERVAL	The amount of elapsed time the query path took to execute.
MEMORY_ALLOCATED_BYTES	INTEGER	The amount of memory the path used, in bytes.
READ_FROM_DISK_BYTES	INTEGER	The number of bytes the path read from disk (or the disk cache).
RECEIVED_BYTES	INTEGER	The number of bytes received over the network.
SENT_BYTES	INTEGER	Size of data sent over the network by the path.
PATH_LINE	VARCHAR	The EXPLAIN plan text string for the path, associated with the PATH ID and PATH_LINE_INDEX columns.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Best practice

Table results can be very wide. For best results when you query the QUERY_PLAN_PROFILES table, sort on these columns:

- TRANSACTION_ID
- STATEMENT_ID
- PATH_ID
- PATH_LINE_INDEX

For example:

```
=> SELECT ... FROM query_plan_profiles
    WHERE ...
    ORDER BY transaction_id, statement_id, path_id, path_line_index;
```

Example

For examples and additional information, see Profiling query plan profiles in the Administrator's Guide

See also

EXECUTION_ENGINE_PROFILES (page 1021)

EXPLAIN (page 828)

PROFILE (page 852)

QUERY_EVENTS (page 1063)

Understanding query plans and Profiling query plan profiles in the Administrator's Guide

QUERY_PROFILES

Provides information about queries that have run.

Column Name	Data Type	Description
SESSION_ID	VARCHAR	The identification of the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	An identifier for the transaction within the session if any; otherwise NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
IDENTIFIER	VARCHAR	A string to identify the query in system tables. **Note:** You can query the IDENTIFIER column to quickly identify queries you have labeled for profiling and debugging. See How to label queries for profiling in the Administrator's Guide for details.
NODE_NAME	VARCHAR	The node name for which information is listed.
QUERY	VARCHAR	The query string used for the query.

QUERY_SEARCH_PATH	VARCHAR	A list of schemas in which to look for tables.
SCHEMA_NAME	VARCHAR	The schema name in which the query is being profiled.
TABLE_NAME	VARCHAR	The table name in the query being profiled.
PROJECTIONS_USED	VARCHAR	The projections used in the query.
QUERY_DURATION_US	NUMERIC(18,0)	The duration of the query in microseconds.
QUERY_START_EPOCH	INTEGER	The epoch number at the start of the given query.
QUERY_START	VARCHAR	The Linux system time of query execution in a format that can be used as a DATE/TIME expression.
QUERY_TYPE	VARCHAR	Is one of INSERT, SELECT, UPDATE, DELETE, UTILITY, or UNKNOWN.
ERROR_CODE	INTEGER	The return error code for the query.
USER_NAME	VARCHAR	The name of the user who ran the query.
PROCESSED_ROW_COUNT	INTEGER	The number of rows returned by the query.
RESERVED_EXTRA_MEMORY	INTEGER	The amount of extra memory, in bytes, reserved for the query. Extra memory is the amount of memory reserved for the plan but not assigned to a particular operator. This is the memory from which unbounded operators pull first. If operators acquire all of the extra memory, then the plan must go back to the Resource Manager for more memory. See **Notes** section below this table.
IS_EXECUTING	BOOLEAN	Displays information about actively running queries, regardless of whether profiling is enabled.

Notes

- The total memory reserved by the query is available in ***RESOURCE_ACQUISITIONS.MEMORY_INUSE_KB*** (page 1081). The difference between RESOURCE_ACQUISITIONS.MEMORY_INUSE_KB and QUERY_PROFILES.EXTRA_MEMORY is the "essential memory."
  - RESOURCE_ACQUISITIONS.MEMORY_INUSE_KB is the total memory acquired.
  - QUERY_PROFILES.RESERVED_EXTRA_MEMORY is the unused portion of the acquired memory.
  - The difference gives you the memory in use.
- If the query has finished executing, query the ***RESOURCE_ACQUISITIONS*** (page 1081) table.

Example

Query the QUERY_PROFILES table:

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM QUERY_PROFILES;

-[ RECORD 1 ]-------+-----------------------------------------------------
...
-[ RECORD 18 ]------+-----------------------------------------------------
node_name           | v_vmartdb_node0001
session_id          | raster-s1-17956:0x1d
transaction_id      | 45035996273728061
statement_id        | 6
identifier          |
query               | SELECT * FROM event_configurations;
query_search_path   | "$user", public, v_catalog, v_monitor, v_internal
schema_name         |
table_name          |
projections_used    | v_monitor.event_configurations_p
query_duration_us   | 9647
query_start_epoch   | 429
query_start         | 2010-10-07 12:46:24.370044-04
query_type          | SELECT
error_code          | 0
user_name           | release
processed_row_count | 16
reserved_extra_memory | 0
is_executing        | f
-[ RECORD ...]------+-----------------------------------------------------
...
```

See Also

QUERY_REQUESTS (page 1073)

RESOURCE_ACQUISITIONS (page 1081)

The following topics in the Administrator's Guide:

- Profiling Database Performance
- Collecting Query Information
- Managing Workloads
- How to label queries for profiling

QUERY_REQUESTS

Returns information about user-issued query requests.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.

USER_NAME	VARCHAR	Name of the user who issued the query at the time HP Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.
REQUEST_TYPE	VARCHAR	Type of the query request. Examples include, but are not limited to: ▪ QUERY ▪ DDL ▪ LOAD ▪ UTILITY ▪ TRANSACTION ▪ PREPARE ▪ EXECUTE ▪ SET ▪ SHOW
REQUEST	VARCHAR	Query statement.
REQUEST_LABEL	VARCHAR	Label of the query, if available/
SEARCH_PATH	VARCHAR	Contents of the search path.
MEMORY_ACQUIRED_MB	FLOAT	Memory acquired by this query request in megabytes.
SUCCESS	BOOLEAN	Value returned if the query successfully executed.
ERROR_COUNT	INTEGER	Number of errors encountered in this query request (logged in ***ERROR_MESSAGES*** (page 1018) table).
START_TIMESTAMP	TIMESTAMPTZ	Beginning of history interval.
END_TIMESTAMP	TIMESTAMPTZ	End of history interval.
REQUEST_DURATION_MS	INTEGER	Length of time the query ran in milliseconds.
IS_EXECUTING	BOOLEAN	Distinguishes between actively-running (t) and completed (f) queries.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Examples

```
select * from query_requests;
 node_name | user_name |    session_id    | request_id |  transaction_id   | statement_id |
request_type |
request
| request_label |                   search_path                    | memory_acquired_mb | success
| error_count |        start_timestamp        |         end_timestamp         | request_duration_ms
| is_executing
-----------+-----------+------------------+------------+-------------------+--------------+------
--------+-------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
-----------------------------------------------+--------------+--------------------------
----------------------+-------------------+---------+-------------+--------------------------
----+------------------------------+--------------------+-------------
 node01    | kelly     | keprl-15875:0x76 |         22 | 45035996273705743 |            3 | QUERY
| select * from query_requests;
|               | "$user", public, v_catalog, v_monitor, v_internal |                100 |         |
| 2011-09-16 13:38:53.918111-04 |                                   |                    | t
 node01    | kelly     | keprl-15875:0x76 |         21 | 45035996273705743 |            2 | QUERY
| select * from query_events order by node_name;
|               | "$user", public, v_catalog, v_monitor, v_internal |             103.71 | t       |
| 2011-09-16 13:38:53.052501-04 | 2011-09-16 13:38:53.505158-04 |                453 | f
 node01    | kelly     | keprl-15875:0x76 |         20 | 45035996273705743 |            1 | QUERY
| select * from t where a = 200;
|               | "$user", public, v_catalog, v_monitor, v_internal |                100 | t       |
| 2011-09-16 13:38:52.98436-04  | 2011-09-16 13:38:53.047512-04 |                 63 | f
 node01    | kelly     | keprl-15875:0x76 |         19 | 45035996273705742 |            1 | DDL
| grant select on t to u;
|               | "$user", public, v_catalog, v_monitor, v_internal |                  | f       |
1 | 2011-09-16 13:38:52.979343-04 | 2011-09-16 13:38:52.97951-04  |                  0 | f
 node01    | kelly     | keprl-15875:0x76 |         18 | 45035996273705739 |            2 | QUERY
| select * from t where a = 100;
|               | "$user", public, v_catalog, v_monitor, v_internal |                100 | t       |
| 2011-09-16 13:38:52.915772-04 | 2011-09-16 13:38:52.970562-04 |                 55 | f
 node01    | kelly     | keprl-15875:0x76 |         17 | 45035996273705739 |            1 | QUERY
| select stream_name, schema_name, table_name, is_executing, accepted_row_count, rejected_row_count,
read_bytes, input_file_size_bytes, parse_complete_percent, unsorted_row_count, sorted_row_count,
sort_complete_percent from load_streams; |               | "$user", public, v_catalog, v_monitor,
v_internal |            103.55 | t       |            | 2011-09-16 13:38:47.119222-04 | 2011-09-16
13:38:49.784388-04 |                    2665 | f
 node01    | kelly     | keprl-15875:0x76 |         16 | 45035996273705736 |            1 | LOAD
| profile copy t from stdin delimiter ',' direct stream name 'dcv_test';
|               | "$user", public, v_catalog, v_monitor, v_internal |                200 | t       |
| 2011-09-16 13:38:46.997683-04 | 2011-09-16 13:38:47.104502-04 |                107 | f
 node01    | kelly     | keprl-15875:0x76 |         15 | 45035996273705737 |            1 | DDL
| CREATE PROJECTION public.t AS  SELECT * FROM public.t  ORDER BY  a SEGMENTED BY hash( a) ALL NODES
KSAFE;
|               | "$user", public, v_catalog, v_monitor, v_internal |                200 | t       |
| 2011-09-16 13:38:46.96377-04  | 2011-09-16 13:38:46.994311-04 |                 31 | f
 node01    | kelly     | keprl-15875:0x76 |         14 | 45035996273705735 |            1 | DDL
| create table t (a int);
|               | "$user", public, v_catalog, v_monitor, v_internal |                  | t       |
| 2011-09-16 13:38:46.939075-04 | 2011-09-16 13:38:46.953036-04 |                 14 | f
 node01    | kelly     | keprl-15875:0x76 |         13 | 45035996273705734 |           11 | UTILITY
| select clear_data_collector('Errors');
|               | "$user", public, v_catalog, v_monitor, v_internal |                  | t       |
| 2011-09-16 13:38:46.927015-04 | 2011-09-16 13:38:46.933721-04 |                  6 | f
 node01    | kelly     | keprl-15875:0x76 |         12 | 45035996273705734 |           10 | UTILITY
| select clear_data_collector('OptimizerEvents');
```

```
|              | "$user", public, v_catalog, v_monitor, v_internal |                    | t       |
| 2011-09-16 13:38:46.923711-04 | 2011-09-16 13:38:46.925269-04 |                  2 | f
 node01    | kelly     | keprl-15875:0x76 |         11 | 45035996273705734 |           9 | UTILITY
| select clear_data_collector('ExecutionEngineEvents');
|              | "$user", public, v_catalog, v_monitor, v_internal |                    | t       |
| 2011-09-16 13:38:46.9202-04   | 2011-09-16 13:38:46.921827-04 |                  1 | f
 node01    | kelly     | keprl-15875:0x76 |         10 | 45035996273705734 |           8 | UTILITY
| select clear_data_collector('TransactionEnds');
|              | "$user", public, v_catalog, v_monitor, v_internal |                    | t       |
| 2011-09-16 13:38:46.916656-04 | 2011-09-16 13:38:46.918302-04 |                  2 | f
 node01    | kelly     | keprl-15875:0x76 |          9 | 45035996273705734 |           7 | UTILITY
| select clear_data_collector('TransactionStarts');
|              | "$user", public, v_catalog, v_monitor, v_internal |                    | t       |
| 2011-09-16 13:38:46.912786-04 | 2011-09-16 13:38:46.914793-04 |                  2 | f
 node01    | kelly     | keprl-15875:0x76 |          8 | 45035996273705734 |           6 | UTILITY
| select clear_data_collector('ExecutionEngineProfiles');
|              | "$user", public, v_catalog, v_monitor, v_internal |                    | t       |
| 2011-09-16 13:38:46.906784-04 | 2011-09-16 13:38:46.910309-04 |                  4 | f
 node01    | kelly     | keprl-15875:0x76 |          7 | 45035996273705734 |           5 | UTILITY
| select clear_data_collector('ProjectionsUsed');
|              | "$user", public, v_catalog, v_monitor, v_internal |                    | t       |
| 2011-09-16 13:38:46.902882-04 | 2011-09-16 13:38:46.904731-04 |                  2 | f
 node01    | kelly     | keprl-15875:0x76 |          6 | 45035996273705734 |           4 | UTILITY
| select clear_data_collector('RequestsCompleted');
|              | "$user", public, v_catalog, v_monitor, v_internal |                    | t       |
| 2011-09-16 13:38:46.899131-04 | 2011-09-16 13:38:46.900843-04 |                  1 | f
 node01    | u1        | keprl-15875:0x7a |          1 | 45035996273705741 |           1 | QUERY
| select * from load_streams
|              | "$user", public, v_catalog, v_monitor, v_internal |             103.54 | t       |
| 2011-09-16 13:38:49.820262-04 | 2011-09-16 13:38:52.904827-04 |               3084 | f
 node01    | u1        | keprl-15875:0x7d |          1 | 45035996273705745 |           1 | QUERY
| select node_name, event_category, event_type, event_description, event_details, suggested_action
from query_events order by node_name
|              | "$user", public, v_catalog, v_monitor, v_internal |             103.71 | t       |
| 2011-09-16 13:38:53.520167-04 | 2011-09-16 13:38:53.906266-04 |                386 | f
(19 rows)
```

See Also

QUERY_PROFILES (page 1071)

REBALANCE_PROJECTION_STATUS

Maintain history on rebalance progress for relevant projections.

Column Name	Data Type	Description
PROJECTION_ID	INTEGER	Identifier of the projection that will be, was, or is being rebalanced.
PROJECTION_SCHEMA	VARCHAR	Schema of the projection that will be, was, or is being rebalanced.
PROJECTION_NAME	VARCHAR	Name of the projection that will be, was, or is being rebalanced.
ANCHOR_TABLE_ID	INTEGER	Anchor table identifier of the projection that will be, was, or is being rebalanced.

ANCHOR_TABLE_NAME	VARCHAR	Anchor table name of the projection that will be, was, or is being rebalanced.
REBALANCE_METHOD	VARCHAR	Method that will be, is, or was used to rebalance the projection. Possible values are: ▪ REFRESH ▪ REPLICATE ▪ ELASTIC_CLUSTER
DURATION_SEC	INTERVAL SEC	Length of time (seconds) rebalance has been working on this projection, including time to separate storage, if that work is required.
SEPARATED_PERCENT	NUMERIC(5,2)	Percent of storage that has been separated for this projection.
TRANSFERRED_PERCENT	NUMERIC(5,2)	Percent of storage that has been transferred, for this projection.
SEPARATED_BYTES	INTEGER	Number of bytes, separated by the corresponding rebalance operation, for this projection.
TO_SEPARATE_BYTES	INTEGER	Number of bytes that remain to be separated by the corresponding rebalance operation for this projection.
TRANSFERRED_BYTES	INTEGER	Number of bytes transferred by the corresponding rebalance operation for this projection.
TO_TRANSFER_BYTES	INTEGER	Number of bytes that remain to be transferred by the corresponding rebalance operation for this projection.
IS_LATEST	BOOLEAN	True if this row pertains to the most recent rebalance activity, where `elastic_cluster_version = (SELECT version FROM v_catalog.elastic_cluster;)`
ELASTIC_CLUSTER_VERSION	INTEGER	The Elastic Cluster has a version, and each time the cluster topology changes, this version is incremented. This column reflects the version to which this row of information pertains. The `TO_*` fields (TO_SEPARATE_* and TO_TRANSFER_*) are only valid for the current version. To view only rows from the current, latest or upcoming rebalance operation, use: `WHERE elastic_cluster_version = (SELECT version FROM v_catalog.elastic_cluster;)`

Permissions

Must be a superuser.

See Also

ELASTIC_CLUSTER (page 940)

REBALANCE_TABLE_STATUS (page 1078)

REBALANCE_TABLE_STATUS

Maintain history on rebalance progress for relevant tables.

Column Name	Data Type	Description
TABLE_ID	INTEGER	Identifier of the table that will be, was, or is being rebalanced.
TABLE_SCHEMA	VARCHAR	Schema of the table that will be, was, or is being rebalanced.
TABLE_NAME	VARCHAR	Name of the table that will be, was, or is being rebalanced.
REBALANCE_METHOD	VARCHAR	Method that will be, is, or was used to rebalance the projections of this table. Possible values are: ▪ REFRESH ▪ REPLICATE ▪ ELASTIC_CLUSTER
DURATION_SEC	INTERVAL SEC	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in ***REBALANCE_PROJECTION_STATUS*** (page 1076).
SEPARATED_PERCENT	NUMERIC(5,2)	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
TRANSFERRED_PERCENT	NUMERIC(5,2)	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
SEPARATED_BYTES	INTEGER	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
TO_SEPARATE_BYTES	INTEGER	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.

TRANSFERRED_BYTES	INTEGER	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
TO_TRANSFER_BYTES	INTEGER	Aggregate, by table_id, rebalance_method, and elastic_cluster_version, of the same in REBALANCE_PROJECTION_STATUS.
IS_LATEST	BOOLEAN	True if this row pertains to the most recent rebalance activity, where `elastic_cluster_version = (SELECT version FROM v_catalog.elastic_cluster;)`
ELASTIC_CLUSTER_VERSION	INTEGER	The Elastic Cluster has a version, and each time the cluster topology changes, this version is incremented. This column reflects the version to which this row of information pertains. The `TO_*` fields (TO_SEPARATE_* and TO_TRANSFER_*) are only valid for the current version. To view only rows from the current, latest or upcoming rebalance operation, use: `WHERE elastic_cluster_version = (SELECT version FROM v_catalog.elastic_cluster;)`

Permissions

Must be superuser.

See Also

ELASTIC_CLUSTER (page 940)

REBALANCE_PROJECTION_STATUS (page 1076)

RECOVERY_STATUS

Provides the status of recovery operations, returning one row for each node.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
RECOVER_EPOCH	INTEGER	Epoch the recovery operation is trying to catch up to.

RECOVERY_PHASE	VARCHAR	Current stage in the recovery process. Can be one of the following: ▪ NULL ▪ current ▪ historical pass *X*, where *X* is the iteration count
SPLITS_COMPLETED	INTEGER	Number of independent recovery SPLITS queries that have run and need to run.
SPLITS_TOTAL	INTEGER	Total number of SPLITS queries that ran. Each query corresponds to one row in the ***PROJECTION_RECOVERIES*** (page 1053) table. If SPLITS_TOTAL = 2, then there should be 2 rows added to PROJECTION_RECOVERIES, showing query details.
HISTORICAL_COMPLETED	INTEGER	Number of independent recovery HISTORICAL queries that have run and need to run.
HISTORICAL_TOTAL	INTEGER	Total number of HISTORICAL queries that ran. Each query corresponds to one row in the PROJECTION_RECOVERIES table. If HISTORICAL_TOTAL = 2, then there should be 2 rows added to PROJECTION_RECOVERIES, showing query details.
CURRENT_COMPLETED	INTEGER	Number of independent recovery CURRENT queries that have run and need to run.
CURRENT_TOTAL	INTEGER	Total number of CURRENT queries that ran. Each query corresponds to one row in the PROJECTION_RECOVERIES table. If CURRENT_TOTAL = 2, then there should be 2 rows added to PROJECTION_RECOVERIES, showing query details.
IS_RUNNING	BOOLEAN	True (*t*) if the node is still running recovery; otherwise false (*f*).

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Note

If you are interested in monitoring recovery progress when recovery seems to be taking a while, note that you cannot query system tables table during cluster recovery; the cluster must be UP to accept connections.

Example

```
=> SELECT * FROM recovery_status;

 node_name | recover_epoch | recovery_phase    | splits_completed | splits_total |
historical_completed | historical_total | current_completed | current_total | is_running
-----------+---------------+-------------------+------------------+--------------+--------------
-------+------------------+------------------+--------------+------------
 node01    |               |                   | 0                | 0            | 0
| 0                | 0                | 0             | f
 node02    | 0             | historical pass 1 | 0                | 0            | 0
| 0                | 0                | 0             | t
 node03    | 1             | current           | 0                | 0            | 0
| 0                | 0                | 0             | f
```

See Also

PROJECTION_RECOVERIES (page 1053)

RESOURCE_ACQUISITIONS

Retains information about resources (memory, open file handles, threads) acquired by each running request for each resource pool in the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node name for which information is listed.
TRANSACTION_ID	INTEGER	Transaction identifier for this request.
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
REQUEST_TYPE	VARCHAR	Type of request issued to a resource pool. Request type can be one of: ▪ Reserve—related to queries ▪ Acquire—[Internal] related to the optimizer and other internal services, such as the Database Designer ▪ Acquire additional—[Internal] related to size adjustment of acquisitions obtained through the first two methods; unusual, outside the WOS
POOL_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the resource pool.
POOL_NAME	VARCHAR	Name of the resource pool.
THREAD_COUNT	INTEGER	Number of threads in use by this request.
OPEN_FILE_HANDLE_COUNT	INTEGER	Number of open file handles in use by this

		request.
MEMORY_INUSE_KB	INTEGER	Amount of memory in kilobytes acquired by this request. See Notes section below this table.
QUEUE_ENTRY_TIMESTAMP	TIMESTAMP TZ	Timestamp when the request was queued at the Resource Manager.
ACQUISITION_TIMESTAMP	TIMESTAMP TZ	Timestamp when the request was admitted to run. See the Notes section below for the difference between these two timestamps.
RELEASE_TIMESTAMP	TIMESTAMP TZ	Time when HP Vertica released this resource acquisition.
DURATION_MS	INTEGER	Duration of the resource request in milliseconds.
IS_EXECUTING	BOOLEAN	Denotes if the query holding the resource is still executing (t).

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Notes

- The total memory reserved by the query is available in RESOURCE_ACQUISITIONS.MEMORY_INUSE_KB. The difference between RESOURCE_ACQUISITIONS.MEMORY_INUSE_KB and QUERY_PROFILES.EXTRA_MEMORY (page 1071) is the "essential memory."
  - RESOURCE_ACQUISITIONS.MEMORY_INUSE_KB is the total memory acquired.
  - QUERY_PROFILES.EXTRA_MEMORY is the unused portion of the acquired memory.
  - The difference gives you the memory in use.
- When monitoring resource pools and resource usage by queries, the “queue wait” time is the difference between acquisition_timestamp and queue_entry_timestamp. For example, to determine how long a query waits in the queue before it is admitted to run, you can get the difference between the acquisition_timestamp and the queue_entry_timestamp using a query like the following:

```
 => SELECT pool_name, queue_entry_timestamp, acquisition_timestamp,
    (acquisition_timestamp-queue_entry_timestamp) AS 'queue wait'
    FROM V_MONITOR.RESOURCE_ACQUISITIONS WHERE node_name ILIKE
'%node0001';
```

- For the WOSDATA ***built-in pool*** (page 757), the queue_entry_timestamp column shows the time when data was first loaded into the WOS. The acquisition_timestamp reflects the last time the amount of data in the WOS changed. There is always a delay between when the data in WOS shrinks in size and when the resource system tables reflect the new size.

Example

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

```
=> SELECT * FROM resource_acquisitions;

-[ RECORD 1 ]----------+-----------------------------
node_name              | v_onenode_node0001
transaction_id         | 45035996273708628
statement_id           | 1
request_type           | Reserve
pool_id                | 45035996273718740
pool_name              | sysquery
thread_count           | 4
open_file_handle_count | 2
memory_inuse_kb        | 16384
queue_entry_timestamp  | 2012-12-09 07:56:41.297533-05
acquisition_timestamp  | 2012-12-09 07:56:41.297536-05
release_timestamp      | 2012-12-09 07:56:41.303201-05
duration_ms            | 6
is_executing           | f
-[ RECORD 2 ]----------+-----------------------------
..
```

See Also

QUERY_PROFILES (page 1071)

RESOURCE_POOL_STATUS (page 1083)

RESOURCE_POOLS (page 965)

RESOURCE_QUEUES (page 1086)

RESOURCE_REJECTIONS (page 1089)

Managing Workloads and the following scenarios in the Administrator's Guide:

- Setting a Hard Limit on Concurrency For An Application
- Monitoring Resource Pools and Resource Usage by Queries

RESOURCE_POOL_STATUS

Provides configuration settings of the various resource pools in the system, including internal pools.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node for which information is provided.
POOL_OID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog that identifies the pool.

POOL_NAME	VARCHAR	The name of the resource pool.
IS_INTERNAL	BOOLEAN	Denotes whether a pool is one of the ***built-in pools*** (page 757).
MEMORY_SIZE_KB	INTEGER	Value of MEMORYSIZE setting of the pool in kilobytes
MEMORY_SIZE_ACTUAL_KB	INTEGER	Current amount of memory in kilobytes allocated to the pool by the resource manager. Note that the actual size can be less than specified in the DDL, if the pool has been recently altered in a running system and the request to shuffle memory is pending. See ***ALTER RESOURCE POOL*** (page 663).
MEMORY_INUSE_KB	INTEGER	Amount of memory in kilobytes acquired by requests running against this pool.
GENERAL_MEMORY_BORROWED_KB	INTEGER	Amount of memory in kilobytes borrowed from the General pool by requests running against this pool. The sum of MEMORY_INUSE_KB and GENERAL_MEMORY_BORROWED_KB should be less than MAX_MEMORY_SIZE_KB (see below).
QUEUEING_THRESHOLD_KB	INTEGER	Calculated as MAX_MEMORY_SIZE_KB * 95%. When the amount of memory used by all requests against this queue exceed the QUEUEING_THRESHOLD_KB (but less than MAX_MEMORY_SIZE_KB), new requests against the pool will be queued until memory becomes available.
MAX_MEMORY_SIZE_KB	INTEGER	Value of MAXMEMORYSIZE size parameter specified when defining the pool. Provides an upper limit on the amount of memory that can be taken up by requests running against this pool. Once this threshold is reached, new requests against this pool are rejected until memory becomes available.
RUNNING_QUERY_COUNT	INTEGER	Number of queries actually running using this pool.
PLANNED_CONCURRENCY	INTEGER	Value of PLANNEDCONCURRENCY parameter specified when defining the pool.
MAX_CONCURRENCY	INTEGER	Value of MAXCONCURRENCY parameter specified when defining the pool.
IS_STANDALONE	BOOLEAN	If the pool is configured to have MEMORYSIZE equal to MAXMEMORYSIZE, it does not borrow any memory from the General pool and hence said to be standalone.
QUEUE_TIMEOUT_IN_SECONDS	INTEGER	Value of QUEUETIMEOUT parameter that was

		specified when defining the pool.
EXECUTION_PARALLELISM	INTEGER	[Default: AUTO] Limits the number of threads used to process any single query issued in this resource pool. When set to AUTO, HP Vertica sets this value based on the number of cores, available memory, and amount of data in the system. Unless data is limited, or the amount of data is very small, HP Vertica sets this value to the number of cores on the node. Reducing this value increases the throughput of short queries issued in the pool, especially if the queries are executed concurrently. If you choose to set this parameter manually, set it to a value between 1 and the number of cores.
PRIORITY	INTEGER	Value of PRIORITY parameter specified when defining the pool.
RUNTIME_PRIORITY	VARCHAR	Value of RUNTIME_PRIORITY specified when defining the pool.
RUNTIME_PRIORITY_THRESHOLD	INTEGER	Value of RUNTIME_PRIORITY_THRESHOLD specified when defining the pool.
SINGLE_INITIATOR	BOOL	Value of SINGLEINITIATOR parameter specified when defining the pool.
QUERY_BUDGET_KB	INTEGER	The current amount of memory that queries are tuned to use.

Example

The following command finds all the configuration settings of the various resource pools on node02:

```
=> SELECT * FROM RESOURCE_POOL_STATUS WHERE node_name ILIKE '%node0002' limit 2;

-[ RECORD 1 ]--------------+-----------------------
node_name                  | v_clickstream_node0002
pool_oid                   | 45035996273719428
pool_name                  | general
is_internal                | t
memory_size_kb             | 596928
memory_size_actual_kb      | 596928
memory_inuse_kb            | 0
general_memory_borrowed_kb | 0
queueing_threshold_kb      | 447696
max_memory_size_kb         | 596928
running_query_count        | 0
planned_concurrency        | 4
max_concurrency            |
```

```
is_standalone               | t
queue_timeout_in_seconds    | 300
priority                    | 0
single_initiator            | false
query_budget_kb             | 111924
-[ RECORD 2 ]---------------+-----------------------
node_name                   | v_clickstream_node0002
pool_oid                    | 45035996273719430
pool_name                   | sysquery
is_internal                 | t
memory_size_kb              | 65536
memory_size_actual_kb       | 65536
memory_inuse_kb             | 0
general_memory_borrowed_kb  | 0
queueing_threshold_kb       | 496848
max_memory_size_kb          | 662464
running_query_count         | 0
planned_concurrency         | 4
max_concurrency             |
is_standalone               | f
queue_timeout_in_seconds    | 300
priority                    | 110
single_initiator            | false
query_budget_kb             | 16384
```

See Also

RESOURCE_ACQUISITIONS (page 1081)

RESOURCE_POOLS (page 965)

RESOURCE_QUEUES (page 1086)

RESOURCE_REJECTIONS (page 1089)

Managing Workloads, Monitoring Resource Pools and Resource Usage by Queries, Scenario: Restricting Resource Usage of Ad-hoc Query Application in the Administrator's Guide

RESOURCE_QUEUES

Provides information about requests pending for various resource pools.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The name of the node for which information is listed.
TRANSACTION_ID	INTEGER	Transaction identifier for this request
STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is

		currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID uniquely identifies a statement within a session.
POOL_NAME	VARCHAR	The name of the resource pool
MEMORY_REQUESTED_KB	INTEGER	Amount of memory in kilobytes requested by this request
PRIORITY	INTEGER	Value of PRIORITY parameter specified when defining the pool.
POSITION_IN_QUEUE	INTEGER	Position of this request within the pool's queue
QUEUE_ENTRY_TIMESTAMP	TIMESTAMP	Timestamp when the request was queued

See Also

RESOURCE_ACQUISITIONS (page 1081)

RESOURCE_POOLS (page 965)

RESOURCE_REJECTIONS (page 1089)

Managing Workloads in the Administrator's Guide

RESOURCE_REJECTION_DETAILS

Records an entry for each resource request that HP Vertica denies. This is useful for determining if there are resource space issues, as well as which users/pools encounter problems.

Column Name	Data Type	Description
REJECTED_TIMESTAMP	TIMESTAMPTZ	Time when HP Vertica rejected the resource.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user at the time HP Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
REQUEST_ID	INTEGER	Unique identifier of the query request in the user session.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL.

STATEMENT_ID	INTEGER	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID, STATEMENT_ID, and REQUEST_ID uniquely identifies a statement within a session.
POOL_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the resource pool.
POOL_NAME	VARCHAR	Name of the resource pool
REASON	VARCHAR	Reason for rejecting this request; for example: ▪ Usage of single request exceeds high limit ▪ Timed out waiting for resource reservation ▪ Canceled waiting for resource reservation
RESOURCE_TYPE	VARCHAR	Memory, threads, file handles or execution slots. The following list shows the resources that are limited by the resource manager. A query might need some amount of each resource, and if the amount needed is not available, the query is queued and could eventually time out of the queue and be rejected. ▪ Number of running plans ▪ Number of running plans on initiator node (local) ▪ Number of requested threads ▪ Number of requested file handles ▪ Number of requested KB of memory ▪ Number of requested KB of address space **Note:** Execution slots are determined by MAXCONCURRENCY parameter.
REJECTED_VALUE	INTEGER	Amount of the specific resource requested by the last rejection

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

See Also

RESOURCE_REJECTIONS (page 1089)

RESOURCE_REJECTIONS

Monitors requests for resources that are rejected by the Resource Manager.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
POOL_ID	INTEGER	A unique numeric ID, assigned by the HP Vertica catalog, which identifies the resource pool.
POOL_NAME	VARCHAR	The name of the resource pool.
REASON	VARCHAR	The reason for rejecting this request; for example: ▪ Usage of single request exceeds high limit ▪ Timed out waiting for resource reservation ▪ Canceled waiting for resource reservation
RESOURCE_TYPE	VARCHAR	Memory, threads, file handles or execution slots. The following list shows the resources that are limited by the resource manager. A query might need some amount of each resource, and if the amount needed is not available, the query is queued and could eventually time out of the queue and be rejected. ▪ Number of running plans ▪ Number of running plans on initiator node (local) ▪ Number of requested threads ▪ Number of requested file handles ▪ Number of requested KB of memory ▪ Number of requested KB of address space **Note:** Execution slots are determined by MAXCONCURRENCY parameter.
REJECTION_COUNT	INTEGER	Number of requests rejected due to specified reason and RESOURCE_TYPE.
FIRST_REJECTED_TIMESTAMP	TIMESTAMPTZ	The time of the first rejection for this pool
LAST_REJECTED_TIMESTAMP	TIMESTAMPTZ	The time of the last rejection for this pool

LAST_REJECTED_VALUE	INTEGER	The amount of the specific resource requested by the last rejection

Notes

Information is valid only as long as the node is up and the counters reset to 0 upon node restart.

Example

```
=> SELECT node_name, pool_name, reason, resource_type, rejection_count AS count,
   last_rejected_timestamp AS time, last_rejected_value AS value
   FROM resource_rejections;

 node_name |  pool_name  |               reason                |   resource_type   | count |
time                |       value
-----------+-------------+-------------------------------------+-------------------+-------
---------------------------+------------------
 initiator | alsohassome | Request exceeded high limit         | Memory(KB)        |  1    |
2011-09-20 17:06:35.549686-04 |            102400
 initiator | ceo         | Timedout waiting for resource request | Memory(KB)        |  1    |
2011-09-20 17:06:35.233777-04 |            102400
 initiator | empty       | Request exceeded high limit         | Queries           |  1    |
2011-09-20 17:06:40.788562-04 |                 1
 initiator | general     | Request exceeded high limit         | Address space(KB) |  2    |
2011-09-20 17:06:40.233878-04 | 45035996273704970
 initiator | general     | Request exceeded high limit         | Memory(KB)        | 24    |
2011-09-20 17:06:49.6701-04   |           8395584
 initiator | sa          | Request exceeded high limit         | Memory(KB)        |  3    |
2011-09-20 17:06:37.070787-04 |            102400
 initiator | sa          | Timedout waiting for resource request | Memory(KB)        |  1    |
2011-09-20 17:06:37.12109-04  |                10
 initiator | small       | Request exceeded high limit         | Memory(KB)        | 26    |
2011-09-20 17:06:32.410326-04 |            102400
 initiator | small       | Timedout waiting for resource request | Memory(KB)        |  2    |
2011-09-20 17:06:33.3467-04   |            122880
 initiator | sysdata     | Request exceeded high limit         | Memory(KB)        |  5    |
2011-09-20 17:06:41.751985-04 |            102400
(10 rows)
```

The following command returns the type of resources currently running on the node:

```
=> SELECT resource_type FROM resource_rejections;
 resource_type
---------------
 UPDATE_QUERY
 UPDATE_QUERY
 UPDATE_QUERY
(3 rows)
```

See Also

CLEAR_RESOURCE_REJECTIONS (page 456)

DISK_RESOURCE_REJECTIONS (page 1013)

Managing Workloads and Managing System Resource Usage in the Administrator's Guide

RESOURCE_USAGE

Monitors system resource management on each node.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
REQUEST_COUNT	INTEGER	The cumulative number of requests for threads, file handles, and memory (in kilobytes).
LOCAL_REQUEST_COUNT	INTEGER	The cumulative number of local requests.
REQUEST_QUEUE_DEPTH	INTEGER	The current request queue depth.
ACTIVE_THREAD_COUNT	INTEGER	The current number of active threads.
OPEN_FILE_HANDLE_COUNT	INTEGER	The current number of open file handles.
MEMORY_REQUESTED_KB	INTEGER	The memory requested in kilobytes.
ADDRESS_SPACE_REQUESTED_KB	INTEGER	The address space requested in kilobytes.
WOS_USED_BYTES	INTEGER	The size of the WOS in bytes.
WOS_ROW_COUNT	INTEGER	The number of rows in the WOS.
ROS_USED_BYTES	INTEGER	The size of the ROS in bytes.
ROS_ROW_COUNT	INTEGER	The number of rows in the ROS.
TOTAL_USED_BYTES	INTEGER	The total size of storage (WOS + ROS) in bytes.
TOTAL_ROW_COUNT	INTEGER	The total number of rows in storage (WOS + ROS).
RESOURCE_REQUEST_REJECT_ COUNT	INTEGER	The number of rejected plan requests.
RESOURCE_REQUEST_TIMEOUT_ COUNT	INTEGER	The number of resource request timeouts.
RESOURCE_REQUEST_CANCEL_ COUNT	INTEGER	The number of resource request cancelations.
DISK_SPACE_REQUEST_REJECT_ COUNT	INTEGER	The number of rejected disk write requests.
FAILED_VOLUME_REJECT_COUNT	INTEGER	The number of rejections due to a failed volume.
TOKENS_USED	INTEGER	For internal use only.
TOKENS_AVAILABLE	INTEGER	For internal use only.

Example

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM RESOURCE_USAGE;

-[ RECORD 1 ]-------------------+---------------------------
node_name                      | node01
request_count                  | 1
local_request_count            | 1
request_queue_depth            | 0
active_thread_count            | 4
open_file_handle_count         | 2
memory_requested_kb            | 4352
address_space_requested_kb     | 106752
wos_used_bytes                 | 0
wos_row_count                  | 0
ros_used_bytes                 | 10390319
ros_row_count                  | 324699
total_used_bytes               | 10390319
total_row_count                | 324699
resource_request_reject_count  | 0
resource_request_timeout_count | 0
resource_request_cancel_count  | 0
disk_space_request_reject_count | 0
failed_volume_reject_count     | 0
tokens_used                    | 1
tokens_available               | 7999999
-[ RECORD 2 ]-------------------+---------------------------
node_name                      | node02
request_count                  | 0
local_request_count            | 0
request_queue_depth            | 0
active_thread_count            | 0
open_file_handle_count         | 0
memory_requested_kb            | 0
address_space_requested_kb     | 0
wos_used_bytes                 | 0
wos_row_count                  | 0
ros_used_bytes                 | 10359489
ros_row_count                  | 324182
total_used_bytes               | 10359489
total_row_count                | 324182
resource_request_reject_count  | 0
resource_request_timeout_count | 0
resource_request_cancel_count  | 0
disk_space_request_reject_count | 0
failed_volume_reject_count     | 0
tokens_used                    | 0
tokens_available               | 8000000
-[ RECORD 3 ]-------------------+---------------------------
node_name                      | node03
request_count                  | 0
local_request_count            | 0
```

```
request_queue_depth             | 0
active_thread_count             | 0
open_file_handle_count          | 0
memory_requested_kb             | 0
address_space_requested_kb      | 0
wos_used_bytes                  | 0
wos_row_count                   | 0
ros_used_bytes                  | 10355231
ros_row_count                   | 324353
total_used_bytes                | 10355231
total_row_count                 | 324353
resource_request_reject_count   | 0
resource_request_timeout_count  | 0
resource_request_cancel_count   | 0
disk_space_request_reject_count | 0
failed_volume_reject_count      | 0
tokens_used                     | 0
tokens_available                | 8000000
-[ RECORD 4 ]-------------------+--------------------------
node_name                       | node04
request_count                   | 0
local_request_count             | 0
request_queue_depth             | 0
active_thread_count             | 0
open_file_handle_count          | 0
memory_requested_kb             | 0
address_space_requested_kb      | 0
wos_used_bytes                  | 0
wos_row_count                   | 0
ros_used_bytes                  | 10385744
ros_row_count                   | 324870
total_used_bytes                | 10385744
total_row_count                 | 324870
resource_request_reject_count   | 0
resource_request_timeout_count  | 0
resource_request_cancel_count   | 0
disk_space_request_reject_count | 0
failed_volume_reject_count      | 0
tokens_used                     | 0
tokens_available                | 8000000
```

SESSION_PROFILES

Provides basic session parameters and lock time out data. To obtain information about sessions, see Profiling Database Performance.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
USER_NAME	VARCHAR	The name used to log in to the database or

		NULL if the session is internal.
CLIENT_HOSTNAME	VARCHAR	The host name and port of the TCP socket from which the client connection was made; NULL if the session is internal.
LOGIN_TIMESTAMP	TIMESTAMP	The date and time the user logged into the database or when the internal session was created. This field is useful for identifying sessions that have been left open for a period of time and could be idle.
LOGOUT_TIMESTAMP	TIMESTAMP	The date and time the user logged out of the database or when the internal session was closed.
SESSION_ID	VARCHAR	A unique numeric ID assigned by the HP Vertica catalog, which identifies the session for which profiling information is captured. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
EXECUTED_STATEMENT_SUCCESS_COUNT	INTEGER	The number of successfully run statements.
EXECUTED_STATEMENT_FAILURE_COUNT	INTEGER	The number of unsuccessfully run statements.
LOCK_GRANT_COUNT	INTEGER	The number of locks granted during the session.
DEADLOCK_COUNT	INTEGER	The number of deadlocks encountered during the session.
LOCK_TIMEOUT_COUNT	INTEGER	The number of times a lock timed out during the session.
LOCK_CANCELLATION_COUNT	INTEGER	The number of times a lock was canceled during the session.
LOCK_REJECTION_COUNT	INTEGER	The number of times a lock was rejected during a session.
LOCK_ERROR_COUNT	INTEGER	The number of lock errors encountered during the session.

Example

Query the SESSION_PROFILES table:

```
=>\pset expanded
Expanded display on.
=> SELECT * FROM SESSION_PROFILES;
-[ RECORD 1 ]-------------------+--------------------------------
node_name                      | node04
user_name                      | dbadmin
client_hostname                | 192.168.1.1:46816
login_timestamp                | 2009-09-28 11:40:34.01518
```

```
logout_timestamp                 | 2009-09-28 11:41:01.811484
session_id                       | myhost.verticacorp-20790:0x32f
executed_statement_success_count | 51
executed_statement_failure_count | 1
lock_grant_count                 | 579
deadlock_count                   | 0
lock_timeout_count               | 0
lock_cancellation_count          | 0
lock_rejection_count             | 0
lock_error_count                 | 0
```

See Also

LOCKS (page 1037)

SESSIONS

Monitors external sessions. You can use this table to:

- Identify users who are running long queries
- Identify users who are holding locks due to an idle but uncommitted transaction
- Disconnect users in order to shut down the database
- Determine the details behind the type of database security (Secure Socket Layer (SSL) or client authentication) used for a particular session.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
USER_NAME	VARCHAR	The name used to log into the database or NULL if the session is internal.
CLIENT_HOSTNAME	VARCHAR	The host name and port of the TCP socket from which the client connection was made; NULL if the session is internal.
CLIENT_PID	INTEGER	The process identifier of the client process that issued this connection. Remember that the client process could be on a different machine than the server.
LOGIN_TIMESTAMP	TIMESTAMP	The date and time the user logged into the database or when the internal session was created. This can be useful for identifying sessions that have been left open for a period of time and could be idle.
SESSION_ID	VARCHAR	The identifier required to close or interrupt a session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.

CLIENT_LABEL	VARCHAR	A user-specified label for the client connection that can be set when using ODBC. See Label in DSN Parameters in Programmer's Guide. An MC output value means there are is a client connection to an MC-managed database for that USER_NAME.
TRANSACTION_START	DATE	The date/time the current transaction started or NULL if no transaction is running.
TRANSACTION_ID	INTEGER	A string containing the hexadecimal representation of the transaction ID, if any; otherwise NULL.
TRANSACTION_DESCRIPTION	VARCHAR	A description of the current transaction.
STATEMENT_START	TIMESTAMP	The timestamp the current statement started execution, or NULL if no statement is running.
STATEMENT_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the currently-executing statement. **Note:** NULL indicates that no statement is currently being processed.
LAST_STATEMENT_DURATION_US	INTEGER	The duration of the last completed statement in microseconds.
RUNTIME_PRIORITY	VARCHAR	Determines the amount of run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to queries already running in the resource pool. Valid values are: ▪ HIGH ▪ MEDIUM ▪ LOW Queries with a HIGH run-time priority are given more CPU and I/O resources than those with a MEDIUM or LOW run-time priority.
CURRENT_STATEMENT	VARCHAR	The currently executing statement, if any. NULL indicates that no statement is currently being processed.
LAST_STATEMENT	VARCHAR	NULL if the user has just logged in; otherwise the currently running statement or the most recently completed statement.
SSL_STATE	VARCHAR	Indicates if HP Vertica used Secure Socket Layer (SSL) for a particular session. Possible values are: ▪ None – HP Vertica did not use SSL. ▪ Server – Sever authentication was used, so the client could authenticate the server. ▪ Mutual – Both the server and the client

		authenticated one another through mutual authentication. See Implementing Security and Implementing SSL.
AUTHENTICATION_METHOD	VARCHAR	The type of client authentication used for a particular session, if known. Possible values are: ▪ Unknown ▪ Trust ▪ Reject ▪ Kerberos ▪ Password ▪ MD5 ▪ LDAP ▪ Kerberos-GSS See Implementing Security and Implementing Client Authentication.

Notes

- A superuser has unrestricted access to all session information, but users can only view information about their own, current sessions.
- During session initialization and termination, you might see sessions running only on nodes other than the node on which you ran the virtual table query. This is a temporary situation that corrects itself as soon as session initialization and termination completes.

Example

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM SESSIONS;

-[ RECORD 1 ]--------------+---------------------------------------
node_name                  | v_mcdb_node0001
user_name                  | dbadmin
client_hostname            | 00.00.000.00:xxxxx
client_pid                 | 12345
login_timestamp            | 2013-05-09 07:18:11.161721-04
session_id                 | myhost.verticacorp.-7695:0x4f337
client_label               | MC
transaction_start          | 2013-05-09 07:18:16.051058-04
transaction_id             | 49539595901174387
transaction_description    | user dbadmin (select * from sessions;)
statement_start            | 2013-05-09 07:18:19.54812-04
statement_id               | 2
```

```
last_statement_duration_us | 16508
runtime_priority           |
current_statement          | select * from sessions;
last_statement             | select * from sessions;
ssl_state                  | None
authentication_method      | Password
```

See Also

CLOSE_SESSION (page 458) and ***CLOSE_ALL_SESSIONS*** (page 461)

Managing Sessions and Configuration Parameters in the Administrator's Guide

STORAGE_CONTAINERS

Monitors information about WOS and ROS storage containers in the database.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Node name for which information is listed.
SCHEMA_NAME	VARCHAR	Schema name for which information is listed.
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	Projection name for which information is listed on that node.
STORAGE_TYPE	VARCHAR	Type of storage container: ROS or WOS.
STORAGE_OID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the storage.
TOTAL_ROW_COUNT	VARCHAR	Total rows in the storage container listed for that projection.
DELETED_ROW_COUNT	INTEGER	Total rows in the storage container deleted for that projection.
USED_BYTES	INTEGER	Total bytes in the storage container listed for that projection.
START_EPOCH	INTEGER	Number of the start epoch in the storage container for which information is listed.
END_EPOCH	INTEGER	Number of the end epoch in the storage container for which information is listed.
GROUPING	VARCHAR	The group by which columns are stored: ▪ ALL – All columns are grouped ▪ PROJECTION – Columns grouped according to projection definition ▪ NONE – No columns grouped, despite

		grouping in the projection definition ▪ OTHER – Some grouping but neither all nor according to projection (e.g., results from add column)
SEGMENT_LOWER_BOUND	INTEGER	Lower bound of the segment range spanned by the storage container or NULL if the corresponding projection is not elastic.
SEGMENT_UPPER_BOUND	INTEGER	Upper bound of the segment range spanned by the storage container or NULL if the corresponding projection is not elastic.
IS_SORTED	BOOLEAN	Whether the storage container's data is sorted (WOS containers only).
LOCATION_LABEL	VARCHAR (128)	The location label (if any) for the storage container is stored.
DELETE_VECTOR_COUNT	INTEGER	The number of delete vectors in the storage container.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

The following command returns all the nodes on which a segmented projection has data on the TickStore database:

```
TickStore=> SELECT node_name, projection_name, total_row_count
FROM storage_containers ORDER BY projection_name;
```

```
     node_name   |       projection_name    | total_row_count
-----------------+--------------------------+-----------------
 v_tick_node0001 | Quotes_Fact_tmp_node0001 |             512
 v_tick_node0001 | Quotes_Fact_tmp_node0001 |          480176
 v_tick_node0002 | Quotes_Fact_tmp_node0002 |             512
 v_tick_node0002 | Quotes_Fact_tmp_node0002 |          480176
 v_tick_node0003 | Quotes_Fact_tmp_node0003 |          480176
 v_tick_node0003 | Quotes_Fact_tmp_node0003 |             512
 v_tick_node0004 | Quotes_Fact_tmp_node0004 |          480176
 v_tick_node0004 | Quotes_Fact_tmp_node0004 |             512
 v_tick_node0001 | Trades_Fact_tmp_node0001 |             512
 v_tick_node0001 | Trades_Fact_tmp_node0001 |          500334
 v_tick_node0002 | Trades_Fact_tmp_node0002 |          500334
 v_tick_node0002 | Trades_Fact_tmp_node0002 |             512
 v_tick_node0003 | Trades_Fact_tmp_node0003 |          500334
 v_tick_node0003 | Trades_Fact_tmp_node0003 |             512
 v_tick_node0004 | Trades_Fact_tmp_node0004 |          500334
 v_tick_node0004 | Trades_Fact_tmp_node0004 |             512
(16 rows)
```

The following command returns information on inventory_fact projections on all nodes on the Vmart schema:

```
=> SELECT * FROM storage_containers WHERE projection_name LIKE
'inventory_fact_p%';
-[ RECORD 1 ]-----+-------------------------
```

```
node_name          | node01
schema_name        | public
projection_name    | inventory_fact_p_node0001
storage_type       | WOS
storage_oid        | 45035996273720173
total_row_count    | 3000
deleted_row_count  | 100
used_bytes         | 196608
start_epoch        | 1
end_epoch          | 2
grouping           | ALL
-[ RECORD 2 ]------+--------------------------
node_name          | node01
schema_name        | public
projection_name    | inventory_fact_p_node0001
storage_type       | ROS
storage_oid        | 45035996273722211
total_row_count    | 500
deleted_row_count  | 25
used_bytes         | 5838
start_epoch        | 1
end_epoch          | 1
grouping           | ALL
-[ RECORD 3 ]------+--------------------------
node_name          | node01
schema_name        | public
projection_name    | inventory_fact_p_node0001
storage_type       | ROS
storage_oid        | 45035996273722283
total_row_count    | 500
deleted_row_count  | 25
used_bytes         | 5794
start_epoch        | 1
end_epoch          | 1
grouping           | ALL
-[ RECORD 4 ]------+--------------------------
node_name          | node01
schema_name        | public
projection_name    | inventory_fact_p_node0001
storage_type       | ROS
storage_oid        | 45035996273723379
total_row_count    | 500
deleted_row_count  | 25
used_bytes         | 5838
start_epoch        | 1
end_epoch          | 1
grouping           | ALL
-[ RECORD 5 ]------+--------------------------
node_name          | node01
schema_name        | public
projection_name    | inventory_fact_p_node0001
storage_type       | ROS
storage_oid        | 45035996273723451
total_row_count    | 500
```

```
deleted_row_count | 25
used_bytes        | 5794
start_epoch       | 1
end_epoch         | 1
grouping          | ALL
-[ RECORD 6 ]-----+-------------------------
node_name         | node01
schema_name       | public
projection_name   | inventory_fact_p_node0001
storage_type      | ROS
storage_oid       | 45035996273724547
total_row_count   | 500
deleted_row_count | 0
used_bytes        | 5838
start_epoch       | 2
end_epoch         | 2
grouping          | ALL
-[ RECORD 7 ]-----+-------------------------
node_name         | node01
schema_name       | public
projection_name   | inventory_fact_p_node0001
storage_type      | ROS
storage_oid       | 45035996273724619
total_row_count   | 500
deleted_row_count | 0
used_bytes        | 5794
start_epoch       | 2
end_epoch         | 2
grouping          | ALL
-[ RECORD 8 ]-----+-------------------------
...
```

See Also

Column Store Architecture with FlexStore in the Concepts Guide

STORAGE_POLICIES

Monitors the current storage policies in effect for one or more database objects.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	Schema name for which information is listed.
OBJECT_NAME	VARCHAR	The name of the database object associated through the storage policy.
POLICY_DETAILS	VARCHAR	The object type of the storage policy.
LOCATION_LABEL	VARCHAR (128)	The label for this storage location.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

The following query returns the current storage polices:

```
VMART=> select * from v_monitor.storage_policies;
 schema_name | object_name | policy_details | location_label
-------------+-------------+----------------+----------------
 public      | states      | Table          | LEVEL3
 public      | newstates   | Table          | LEVEL3
(2 rows)
```

See Also

PARTITIONS (page 1051)

STORAGE_CONTAINERS (page 1098)

STORAGE_USAGE (page 1104)

Creating Storage Policies and Clearing a Storage Policy in the Administrator's Guide

STORAGE_TIERS

Provides information about all storage locations with the same label across all cluster nodes. This table lists data totals for all same-name labeled locations.

The system table shows what labeled locations exist on the cluster, as well as other cluster-wide data about the locations.

Column Name	Data Type	Description
LOCATION_LABEL	VARCHAR	The label associated with a specific storage location. The `storage_tiers` system table includes data totals for unlabeled locations, which are considered labeled with empty strings ('').
NODE_COUNT	INTEGER	The total number of nodes that include a storage location named `location_label`.

LOCATION_COUNT	INTEGER	The total number of storage locations named location_label. This value can differ from node_count if you create labeled locations with the same name at different paths on different nodes. For example: **node01:** Create one labeled location, FAST **node02**: Create two labeled locations, FAST, at different directory paths In this case, node_count value = 2, while location_count value = 3.
ROS_CONTAINER_COUNT	INTEGER	The total number of ROS containers stored across all cluster nodes for location_label.
TOTAL_OCCUPIED_SIZE	INTEGER	The total number of bytes that all ROS containers for location_label occupy across all cluster nodes.

Permissions

Must be a superuser

Example

```
VMart=> select * from v_monitor.storage_tiers;
 location_label | node_count | location_count | ros_container_count |
total_occupied_size
----------------+------------+----------------+---------------------+---------
------------
                |          1 |              2 |                  17 |
297039391
 SSD            |          1 |              1 |                   9 |
1506
 Schema         |          1 |              1 |                   0 |
0
(3 rows)
```

See Also

DISK_STORAGE (page 1014)

STORAGE_POLICIES (page 1101)

STORAGE_USAGE (page 1104)

Storage Management Functions (page 636)

Creating and Configuring Storage Locations in the Administrator's Guide

STORAGE_USAGE

Provides information about file system storage usage. This is useful for determining disk space usage trends.

Column Name	Data Type	Description
POLL_TIMESTAMP	TIMESTAMPTZ	Time when HP Vertica recorded the row.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
PATH	VARCHAR	Path where the storage location is mounted.
DEVICE	VARCHAR	Device on which the storage location is mounted.
FILESYSTEM	VARCHAR	Filesystem on which the storage location is mounted.
USED_BYTES	INTEGER	Counter history of number of used bytes.
FREE_BYTES	INTEGER	Counter history of number of free bytes.
USAGE_PERCENT	FLOAT	Percent of storage in use.

Permissions

Must be a superuser

Example

```
=> SELECT * FROM storage_usage;

        poll_timestamp         | node_name | path  |  device   | filesystem |  used_bytes  |
free_bytes  | usage_percent
-------------------------------+-----------+-------+-----------+------------+--------------+-----
--------+--------------
 2011-09-16 15:25:57.017042-04 | e0        | /dev  | udev      | devtmpfs   |       319488 |
2017337344 |          0.02
 2011-09-16 15:25:57.017044-04 | e0        | /var  | /dev/sda3 | ext4       |   2377822208 |
2906607616 |            45
 2011-09-16 15:25:57.016812-04 | e1        |       |           | vertica    | 118600830976 |
25081253888 |         82.54
 2011-09-16 15:25:57.016795-04 | e1        | /     | /dev/sda5 | ext4       | 118600830976 |
25081253888 |         82.54
 2011-09-16 15:25:57.016806-04 | e1        | /dev  | udev      | devtmpfs   |       319488 |
2017337344 |          0.02
 2011-09-16 15:25:57.011443-04 | initiator |       |           | vertica    | 118600830976 |
25081253888 |         82.54
 2011-09-16 15:25:57.017037-04 | e0        | /     | /dev/sda5 | ext4       | 118600839168 |
25081245696 |         82.54
 2011-09-16 15:25:57.01704-04  | e0        | /boot | /dev/sda1 | ext4       |     64685056 |
36844544 |         63.71
 2011-09-16 15:25:57.011425-04 | initiator | /     | /dev/sda5 | ext4       | 118600830976 |
25081253888 |         82.54
 2011-09-16 15:25:57.01144-04  | initiator | /var  | /dev/sda3 | ext4       |   2377822208 |
2906607616 |            45
 2011-09-16 15:25:57.017045-04 | e0        |       |           | vertica    | 118600830976 |
```

```
25081253888 |         82.54
 2011-09-16 15:25:57.016801-04 | e1        | /boot | /dev/sda1 | ext4       |     64685056 |
36844544 |         63.71
 2011-09-16 15:25:57.016809-04 | e1        | /var  | /dev/sda3 | ext4       |   2377822208 |
2906607616 |            45
 2011-09-16 15:25:57.011432-04 | initiator | /boot | /dev/sda1 | ext4       |     64685056 |
36844544 |         63.71
 2011-09-16 15:25:57.011436-04 | initiator | /dev  | udev      | devtmpfs   |       319488 |
2017337344 |          0.02
(15 rows)
```

See Also

DISK_STORAGE (page 1014)

STORAGE_CONTAINERS (page 1098)

STORAGE_POLICIES (page 1101)

STORAGE_TIERS (page 1102)

Storage Management Functions (page 636)

Creating and Configuring Storage Locations in the Administrator's Guide

STRATA

Contains internal details of how the Tuple Mover combines ROS containers in each projection, broken down by stratum and classifies the ROS containers by size and partition. The related ***STRATA_STRUCTURES*** (page 1108) table provides a summary of the strata values.

The STRATA table contains detailed information on

For a brief overview of how the Tuple Mover combines ROS containers, see Tuple Mover in the Administrator's Guide.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed
SCHEMA_NAME	VARCHAR	The schema name for which information is listed
PROJECTION_ID	INTEGER	A unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
PROJECTION_NAME	VARCHAR	The projection name for which information is listed on that node
PARTITION_KEY	VARCHAR	The data partition for which information is listed
STRATA_COUNT	INTEGER	The total number of strata for this projection partition

MERGING_STRATA_COUNT	INTEGER	The number of strata the Tuple Mover can merge out.
STRATUM_CAPACITY	INTEGER	The maximum number of ROS containers for the stratum before they must be merged.
STRATUM_HEIGHT	FLOAT	The size ratio between the smallest and largest ROS container in this stratum
STRATUM_NO	INTEGER	The stratum number. Strata are numbered starting at 0, for the stratum containing the smallest ROS containers
STRATUM_LOWER_SIZE	VARCHAR	The smallest ROS container size allowed in this stratum
STRATUM_UPPER_SIZE	VARCHAR	The largest ROS container size allowed in this stratum
ROS_CONTAINER_COUNT	INTEGER	The current number of ROS containers in the projection partition

Example

```
onenode=> SELECT * FROM strata;

-[ RECORD 1 ]--------+-------------------
node_name            | v_onenode_node0001
schema_name          | public
projection_id        | 45035996273718838
projection_name      | trades_p
partition_key        |
strata_count         | 11
merging_strata_count | 4
stratum_capacity     | 32
stratum_height       | 28.8
stratum_no           | 0
stratum_lower_size   | 0B
stratum_upper_size   | 4MB
ROS_container_count  | 1


vmartdb=> \pset expanded
Expanded display is on.
vmartdb=> SELECT node_name, schema_name, projection_name, strata_count,
            stratum_capacity, stratum_height, stratum_no, stratum_lower_size,
            stratum_upper_size, ros_container_count
            FROM strata WHERE node_name ILIKE 'node01' AND stratum_upper_size <
'15MB';

-[ RECORD 1
]-------+------------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | online_sales
projection_name     |
call_center_dimension_DBD_32_seg_vmart_design_vmart_design
strata_count        | 5
```

```
stratum_capacity    | 19
stratum_height      | 8.97589786696783
stratum_no          | 0
stratum_lower_size  | 0B
stratum_upper_size  | 13MB
ROS_container_count | 1
-[ RECORD 2
]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | online_sales
projection_name     |
call_center_dimension_DBD_8_seg_vmart_design_vmart_design
strata_count        | 5
stratum_capacity    | 19
stratum_height      | 8.97589786696783
stratum_no          | 0
stratum_lower_size  | 0B
stratum_upper_size  | 13MB
ROS_container_count | 1
-[ RECORD 3
]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | online_sales
projection_name     | online_sales_fact_DBD_33_seg_vmart_design_vmart_design
strata_count        | 5
stratum_capacity    | 13
stratum_height      | 8.16338338718601
stratum_no          | 1
stratum_lower_size  | 19MB
stratum_upper_size  | 155.104MB
ROS_container_count | 1
-[ RECORD 4
]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | online_sales
projection_name     | online_sales_fact_DBD_9_seg_vmart_design_vmart_design
strata_count        | 5
stratum_capacity    | 13
stratum_height      | 8.16338338718601
stratum_no          | 1
stratum_lower_size  | 19MB
stratum_upper_size  | 155.104MB
ROS_container_count | 1
-[ RECORD 5
]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | public
projection_name     | promotion_dimension_DBD_16_seg_vmart_design_vmart_design
strata_count        | 5
stratum_capacity    | 19
stratum_height      | 8.97589786696783
stratum_no          | 0
stratum_lower_size  | 0B
stratum_upper_size  | 13MB
```

```
ROS_container_count | 1
-[ RECORD 6
]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | public
projection_name     | promotion_dimension_DBD_17_seg_vmart_design_vmart_design
strata_count        | 5
stratum_capacity    | 19
stratum_height      | 8.97589786696783
stratum_no          | 0
stratum_lower_size  | 0B
stratum_upper_size  | 13MB
ROS_container_count | 1
-[ RECORD 7
]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | store
projection_name     | store_sales_fact_DBD_29_seg_vmart_design_vmart_design
strata_count        | 5
stratum_capacity    | 16
stratum_height      | 8.52187248329035
stratum_no          | 1
stratum_lower_size  | 16MB
stratum_upper_size  | 136.35MB
ROS_container_count | 1
-[ RECORD 8
]-------+-----------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | store
projection_name     | store_sales_fact_DBD_5_seg_vmart_design_vmart_design
strata_count        | 5
stratum_capacity    | 16
stratum_height      | 8.52187248329035
stratum_no          | 1
stratum_lower_size  | 16MB
stratum_upper_size  | 136.35MB
ROS_container_count | 1
```

STRATA_STRUCTURES

This table provides an overview of Tuple Mover internal details. It summarizes how the ROS containers are classified by size. A more detailed view can be found in the ***STRATA*** (page 1105) virtual table.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed
SCHEMA_NAME	VARCHAR	The schema name for which information is listed
PROJECTION_NAME	VARCHAR	The projection name for which information is listed on that node
PROJECTION_ID	INTEGER	A unique numeric ID assigned by the HP Vertica

		catalog, which identifies the projection.
PARTITION_KEY	VARCHAR	The data partition for which the information is listed
STRATA_COUNT	INTEGER	The total number of strata for this projection partition
MERGING_STRATA_COUNT	INTEGER	In certain hardware configurations, a high strata could contain more ROS containers than the Tuple Mover can merge out; output from this column denotes the number of strata the Tuple Mover can merge out.
STRATUM_CAPACITY	INTEGER	The maximum number of ROS containers that the strata can contained before it must merge them
STRATUM_HEIGHT	FLOAT	The size ratio between the smallest and largest ROS container in a stratum.
ACTIVE_STRATA_COUNT	INTEGER	The total number of strata that have ROS containers in them

Example

```
onenode=> SELECT * FROM strata_structures;

-[ RECORD 1 ]--------+-------------------
node_name            | v_onenode_node0001
schema_name          | public
projection_name      | trades_p
projection_id        | 45035996273718838
partition_key        |
strata_count         | 11
merging_strata_count | 4
stratum_capacity     | 32
stratum_height       | 28.8
active_strata_count  | 1


vmartdb=> \pset expanded
Expanded display is on.
vmartdb=> SELECT node_name, schema_name, projection_name, strata_count,
            stratum_capacity, stratum_height, stratum_no, stratum_lower_size,
            stratum_upper_size, ros_container_count
            WHERE stratum_capacity > 60;

-[ RECORD 1 ]-------+------------------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | public
projection_name     | shipping_dimension_DBD_22_seg_vmart_design_vmart_design
partition_key       |
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
```

```
-[ RECORD 2 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node01
schema_name         | public
projection_name     | shipping_dimension_DBD_23_seg_vmart_design_vmart_design
partition_key       |
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 3 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node02
schema_name         | public
projection_name     | shipping_dimension_DBD_22_seg_vmart_design_vmart_design
partition_key       |
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 4 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node02
schema_name         | public
projection_name     | shipping_dimension_DBD_23_seg_vmart_design_vmart_design
partition_key       |
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 5 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node03
schema_name         | public
projection_name     | shipping_dimension_DBD_22_seg_vmart_design_vmart_design
partition_key       |
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 6 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node03
schema_name         | public
projection_name     | shipping_dimension_DBD_23_seg_vmart_design_vmart_design
partition_key       |
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
-[ RECORD 7 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node04
schema_name         | public
projection_name     | shipping_dimension_DBD_22_seg_vmart_design_vmart_design
partition_key       |
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
```

```
-[ RECORD 8 ]-------+-----------------------------------------------------
node_name           | v_vmartdb_node04
schema_name         | public
projection_name     | shipping_dimension_DBD_23_seg_vmart_design_vmart_design
partition_key       |
strata_count        | 4
stratum_capacity    | 62
stratum_height      | 25.6511590887058
active_strata_count | 1
```

See Also

STRATA (page 1105)

SYSTEM

Monitors the overall state of the database.

Column Name	Data Type	Description
CURRENT_EPOCH	INTEGER	The current epoch number.
AHM_EPOCH	INTEGER	The AHM epoch number.
LAST_GOOD_EPOCH	INTEGER	The smallest (min) of all the checkpoint epochs on the cluster.
REFRESH_EPOCH	INTEGER	The oldest of the refresh epochs of all the nodes in the cluster
DESIGNED_FAULT_TOLERANCE	INTEGER	The designed or intended K-safety level.
NODE_COUNT	INTEGER	The number of nodes in the cluster.
NODE_DOWN_COUNT	INTEGER	The number of nodes in the cluster that are currently down.
CURRENT_FAULT_TOLERANCE	INTEGER	The number of node failures the cluster can tolerate before it shuts down automatically.
CATALOG_REVISION_NUMBER	INTEGER	The catalog version number.
WOS_USED_BYTES	INTEGER	The WOS size in bytes (cluster-wide).
WOS_ROW_COUNT	INTEGER	The number of rows in WOS (cluster-wide).
ROS_USED_BYTES	INTEGER	The ROS size in bytes (cluster-wide).
ROS_ROW_COUNT	INTEGER	The number of rows in ROS (cluster-wide).
TOTAL_USED_BYTES	INTEGER	The total storage in bytes (WOS + ROS) (cluster-wide).
TOTAL_ROW_COUNT	INTEGER	The total number of rows (WOS + ROS) (cluster-wide).

Example

Query the SYSTEM table:

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM SYSTEM;
-[ RECORD 1 ]------------+----------
current_epoch            | 429
ahm_epoch                | 428
last_good_epoch          | 428
refresh_epoch            | -1
designed_fault_tolerance | 1
node_count               | 4
node_down_count          | 0
current_fault_tolerance  | 1
catalog_revision_number  | 1590
wos_used_bytes           | 0
wos_row_count            | 0
ros_used_bytes           | 443131537
ros_row_count            | 21809072
total_used_bytes         | 443131537
total_row_count          | 21809072
```

If there are no projections in the system, `LAST_GOOD_EPOCH` returns the following:

```
=> SELECT get_last_good_epoch();
ERROR:  Last good epoch not set
```

And if there are projections in the system:

```
=> SELECT get_last_good_epoch();
 get_last_good_epoch
---------------------
                 428
(1 row)
```

SYSTEM_RESOURCE_USAGE

Provides history about system resources, such as memory, CPU, network, disk, I/O.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
END_TIME	TIMESTAMP	End time of the history interval.
AVERAGE_MEMORY_USAGE_PERCENT	FLOAT	Average memory usage in percent of total memory (0-100) during the history interval.

AVERAGE_CPU_USAGE_PERCENT	FLOAT	Average CPU usage in percent of total CPU time (0-100) during the history interval.
NET_RX_KBYTES_PER_SECOND	FLOAT	Average number of kilobytes received from network (incoming) per second during the history interval.
NET_TX_KBYTES_PER_SECOND	FLOAT	Average number of kilobytes transmitting to network (outgoing) per second during the history interval.
IO_READ_KBYTES_PER_SECOND	FLOAT	Disk I/O average number of kilobytes read from disk per second during the history interval.
IO_WRITTEN_KBYTES_PER_SECOND	FLOAT	Average number of kilobytes written to disk per second during the history interval.

Permissions

Must be a superuser.

Example

```
=> SELECT * FROM system_resource_usage WHERE node_name = 'v_myvdb_node04';
-[ RECORD 1 ]----------------+-------------------
node_name                   | v_myvdb_node04
end_time                    | 2012-03-30 17:43:00
average_memory_usage_percent | 3.6
average_cpu_usage_percent    | 9.1
net_rx_kbytes_per_second     | 12.75
net_tx_kbytes_per_second     | 5.77
io_read_kbytes_per_second    | 0
io_written_kbytes_per_second | 643.92
-[ RECORD 2 ]----------------+-------------------
node_name                   | v_myvdb_node04
end_time                    | 2012-03-30 17:38:00
average_memory_usage_percent | 3.59
average_cpu_usage_percent    | 10.4
net_rx_kbytes_per_second     | 5.78
net_tx_kbytes_per_second     | 0.88
io_read_kbytes_per_second    | 0
io_written_kbytes_per_second | 650.28
-[ RECORD 3 ]----------------+-------------------
node_name                   | v_myvdb_node04
end_time                    | 2012-03-30 17:37:00
average_memory_usage_percent | 3.59
average_cpu_usage_percent    | 8.47
net_rx_kbytes_per_second     | 5.41
net_tx_kbytes_per_second     | 0.77
io_read_kbytes_per_second    | 0
io_written_kbytes_per_second | 621.39
-[ RECORD 4 ]----------------+-------------------
node_name                   | v_myvdb_node04
end_time                    | 2012-03-30 17:31:00
average_memory_usage_percent | 3.59
```

```
average_cpu_usage_percent    | 12.35
net_rx_kbytes_per_second     | 5.71
net_tx_kbytes_per_second     | 0.87
io_read_kbytes_per_second    | 0
io_written_kbytes_per_second | 647.06
-[ RECORD 5 ]----------------+--------------------
node_name                    | v_myvdb_node04
end_time                     | 2012-03-30 17:29:00
average_memory_usage_percent | 3.59
average_cpu_usage_percent    | 8.52
net_rx_kbytes_per_second     | 5.56
net_tx_kbytes_per_second     | 0.81
io_read_kbytes_per_second    | 0
io_written_kbytes_per_second | 631.21
...
```

SYSTEM_SERVICES

Provides information about background system services that the Workload Analyzer monitors.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
SERVICE_TYPE	VARCHAR	Type of service; can be one of: ▪ SYSTEM ▪ TUPLE MOVER
SERVICE_GROUP	VARCHAR	Group name, if there are multiple services of the same type.
SERVICE_NAME	VARCHAR	Name of the service.
SERVICE_INTERVAL_SEC	INTEGER	How often the service is executed (in seconds) during the history interval.
IS_ENABLED	BOOLEAN	Denotes if the service is enabled.
LAST_RUN_START	TIMESTAMPTZ	Denotes when the service was started last time.
LAST_RUN_END	TIMESTAMPTZ	Denotes when the service was completed last time.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> SELECT * FROM system_services;

-[ RECORD 1 ]--------+-----------------------------
```

```
node_name             | v_myvdb_node0004
service_type          | System
service_group         |
service_name          | Ageout Session Profiling Data
service_interval_sec | 86400
is_enabled            | t
last_run_start        | 2012-04-03 11:36:54.001782-04
last_run_end          | 2012-04-03 11:36:54.001793-04
-[ RECORD 2 ]---------+------------------------------
node_name             | v_myvdb_node0004
service_type          | System
service_group         |
service_name          | AgeOutEvents
service_interval_sec | 3
is_enabled            | t
last_run_start        | 2012-04-03 14:41:24.001538-04
last_run_end          | 2012-04-03 14:41:24.001544-04
-[ RECORD 3 ]---------+------------------------------
node_name             | v_myvdb_node0004
service_type          | System
service_group         |
service_name          | CatalogCheckpointer
service_interval_sec | 86400
is_enabled            | t
last_run_start        | 2012-04-03 11:36:54.001788-04
last_run_end          | 2012-04-03 11:36:54.002721-04
-[ RECORD 4 ]---------+------------------------------
node_name             | v_myvdb_node0004
service_type          | System
service_group         |
service_name          | Cluster Inviter
service_interval_sec | 2
is_enabled            | t
last_run_start        | 2012-04-03 14:41:25.002031-04
last_run_end          | 2012-04-03 14:41:25.002671-04
-[ RECORD 5 ]---------+------------------------------
...


=> SELECT service_type, last_run_start, last_run_end FROM system_services;

 service_type |        last_run_start         |         last_run_end
--------------+-------------------------------+-------------------------------
 System       |                               |
 System       | 2012-03-01 08:10:05.010077-05 | 2012-03-01 08:10:05.010081-05
 System       |                               |
 System       | 2012-03-01 08:10:06.003775-05 | 2012-03-01 08:10:06.00499-05
 System       | 2012-03-01 07:27:05.004958-05 | 2012-03-01 07:27:05.005376-05
 System       | 2012-03-01 06:32:45.001812-05 | 2012-03-01 06:32:45.002249-05
 System       |                               |
 System       | 2012-03-01 08:10:05.006397-05 | 2012-03-01 08:10:05.006399-05
 System       | 2012-03-01 06:29:05.000905-05 | 2012-03-01 06:29:05.001517-05
 System       | 2012-03-01 08:10:05.006213-05 | 2012-03-01 08:10:05.006215-05
 System       | 2012-03-01 08:10:05.006379-05 | 2012-03-01 08:10:05.007055-05
 System       | 2012-03-01 08:10:05.009981-05 | 2012-03-01 08:10:05.009983-05
 System       | 2012-03-01 08:10:05.00988-05  | 2012-03-01 08:10:05.009882-05
 Tuple Mover  | 2012-03-01 08:10:05.006673-05 | 2012-03-01 08:10:05.006675-05
 Tuple Mover  | 2012-03-01 08:07:05.006837-05 | 2012-03-01 08:07:05.009541-05
```

```
 Tuple Mover  | 2012-03-01 08:09:05.001376-05 | 2012-03-01 08:09:05.001378-05
 Tuple Mover  | 2012-03-01 07:17:05.000908-05 | 2012-03-01 07:17:05.015156-05
 Tuple Mover  | 2012-03-01 08:07:05.00679-05  | 2012-03-01 08:07:05.007486-05
 Tuple Mover  | 2012-03-01 08:07:05.006673-05 | 2012-03-01 08:07:05.010128-05
 Tuple Mover  | 2012-03-01 08:07:05.002946-05 | 2012-03-01 08:07:05.010192-05
 Tuple Mover  | 2012-03-01 08:07:05.002946-05 | 2012-03-01 08:07:05.010192-05
 Tuple Mover  | 2012-03-01 08:07:05.002962-05 | 2012-03-01 08:07:05.007198-05
```

SYSTEM_SESSIONS

Provides information about system internal session history by system task.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user at the time HP Vertica recorded the session.
SESSION_ID	INTEGER	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any. If a session is active but no transaction has begun, TRANSACTION_ID returns NULL.
STATEMENT_ID	VARCHAR	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID and STATEMENT_ID uniquely identifies a statement within a session.
SESSION_TYPE	VARCHAR	Session type. Can be one of: ▪ LICENSE_AUDIT ▪ STARTUP ▪ SHUTDOWN ▪ VSPREAD
RUNTIME_PRIORITY	VARCHAR	Determines the amount of run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to queries already running in the resource pool. Valid values are: ▪ HIGH ▪ MEDIUM ▪ LOW Queries with a HIGH run-time priority are given more CPU and I/O resources than those with a MEDIUM or LOW run-time priority.
DESCRIPTION	VARCHAR	Transaction description in this session.

SESSION_START_TIMESTAMP	TIMESTAMPTZ	Value of session at beginning of history interval.
SESSION_END_TIMESTAMP	TIMESTAMPTZ	Value of session at end of history interval.
IS_ACTIVE	BOOLEAN	Denotes if the session is still running.
SESSION_DURATION_MS	INTEGER	Duration of the session in milliseconds.

Permissions

Must be a superuser.

Example

```
=> SELECT * FROM system_sessions;

-[ RECORD 1 ]-----------+-----------------------------------------------------
node_name               | v_vmart_node0002
user_name               | dbadmin
session_id              | xxxx02.verticacorp.-23295:0x2b7
transaction_id          | 49539595901220372
statement_id            |
session_type            | MERGEOUT
runtime_priority        |
description             | Txn: b0000000023614 'Mergeout: Tuple Mover'
session_start_timestamp | 2012-12-10 06:41:03.025615-05
session_end_timestamp   | 2012-12-10 06:41:03.030063-05
is_active               | f
session_duration_ms     | 1
-[ RECORD 2 ]-----------+-----------------------------------------------------
node_name               | v_vmart_node0003
user_name               | dbadmin
session_id              | xxxx03.verticacorp.-22620:0x2bd
transaction_id          | 54043195528590868
statement_id            |
session_type            | MOVEOUT
runtime_priority        |
description             | Txn: c0000000023614 'Moveout: Tuple Mover'
session_start_timestamp | 2012-12-10 06:41:03.007496-05
session_end_timestamp   | 2012-12-10 06:41:03.00844-05
is_active               | f
session_duration_ms     | 1
-[ RECORD 3 ]-----------+-----------------------------------------------------
node_name               | v_vmart_node0001
user_name               | dbadmin
session_id              | xxxx01.verticacorp.-30972:0x1f3
transaction_id          | 45035996273960317
statement_id            |
session_type            | REBALANCE_CLUSTER
runtime_priority        |
description             | Txn: a000000003e57d 'rebalance_cluster(background)'
session_start_timestamp | 2012-12-10 06:37:26.015479-05
session_end_timestamp   | 2012-12-10 06:37:26.033779-05
is_active               | f
session_duration_ms     | 13
-[ RECORD 4 ]-----------+-----------------------------------------------------
...
```

See also

CURRENT_SESSION (page 999)

SESSION_PROFILES (page 1093)

SESSIONS (page 1095)

USER_SESSIONS (page 1126)

TRANSACTIONS

Records the details of each transaction.

Column Name	Data Type	Description
START_TIMESTAMP	TIMESTAMPTZ	Beginning of history interval.
END_TIMESTAMP	TIMESTAMPTZ	End of history interval.
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_ID	INTEGER	Unique numeric ID assigned by the Vertica catalog, which identifies the user.
USER_NAME	VARCHAR	Name of the user for which transaction information is listed.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
TRANSACTION_ID	INTEGER	Identifier for the transaction within the session, if any; otherwise NULL.
DESCRIPTION	VARCHAR	Textual description of the transaction.
START_EPOCH	INTEGER	Number of the start epoch for the transaction.
END_EPOCH	INTEGER	Number of the end epoch for the transaction
NUMBER_OF_STATEMENTS	INTEGER	Number of query statements executed in this transaction.
ISOLATION	VARCHAR	Denotes the transaction mode as "READ COMMITTED" or "SERIALIZABLE".
IS_READ_ONLY	BOOLEAN	Denotes "READ ONLY" transaction mode.
IS_COMMITTED	BOOLEAN	Determines if the transaction was committed. False means ROLLBACK.
IS_LOCAL	BOOLEAN	Denotes transaction is local (non-distributed).

IS_INITIATOR	BOOLEAN	Denotes if the transaction occurred on this node (t).
IS_DDL	BOOLEAN	Distinguishes between a DDL transaction (t) and non-DDL transaction (f).

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> SELECT * FROM transactions LIMIT 4;

-[ RECORD 1 ]--------+--------------------------------------------------
start_timestamp      | 2012-03-30 17:25:16.025136-04
end_timestamp        | 2012-03-30 17:25:16.029179-04
node_name            | v_myvdb_node0004
user_id              | 45035996273704962
user_name            | dbadmin
session_id           | raster-s1-10295:0x389d
transaction_id       | 58546795155820007
description          | Txn: d0000000000de7 'RemoteNodeLocalState_RowCounts'
start_epoch          | 4
end_epoch            | 4
number_of_statements | 1
isolation            | SERIALIZABLE
is_read_only         | f
is_committed         | f
is_local             | t
is_initiator         | t
is_ddl               | f
-[ RECORD 2 ]--------+--------------------------------------------------
start_timestamp      | 2012-03-30 17:25:14.001833-04
end_timestamp        | 2012-03-30 17:25:14.001915-04
node_name            | v_myvdb_node0004
user_id              | 45035996273704962
user_name            | dbadmin
session_id           | raster-s4-22870:0x3b4b
transaction_id       | 58546795155820006
description          | Txn: d0000000000de6 'Check LGE'
start_epoch          | 4
end_epoch            | 4
number_of_statements | 1
isolation            | SERIALIZABLE
is_read_only         | f
is_committed         | f
is_local             | t
is_initiator         | t
is_ddl               | f

-[ RECORD 3 ]--------+--------------------------------------------------
start_timestamp      | 2012-03-30 17:22:16.0105-04
end_timestamp        | 2012-03-30 17:22:16.014883-04
node_name            | v_myvdb_node0004
user_id              | 45035996273704962
user_name            | dbadmin
session_id           | raster-s1-10295:0x3839
transaction_id       | 58546795155819998
description          | Txn: d0000000000dde 'RemoteNodeLocalState_RowCounts'
start_epoch          | 4
end_epoch            | 4
number_of_statements | 1
```

```
isolation            | SERIALIZABLE
is_read_only         | f
is_committed         | f
is_local             | t
is_initiator         | t
is_ddl               | f
-[ RECORD 4 ]--------+-------------------------------------------------
start_timestamp      | 2012-03-30 17:22:10.0063-04
end_timestamp        | 2012-03-30 17:22:10.006571-04
node_name            | v_myvdb_node0004
user_id              | 45035996273704962
user_name            | dbadmin
session_id           | raster-s1-10295:0x3835
transaction_id       | 58546795155819997
description          | Txn: d0000000000ddd 'ProjUtil::getLocalNodeLGE'
start_epoch          | 4
end_epoch            | 4
number_of_statements | 1
isolation            | SERIALIZABLE
is_read_only         | f
is_committed         | f
is_local             | t
is_initiator         | t
is_ddl               | f
```

See Also

Transactions in the Concepts Guide

TUNING_RECOMMENDATIONS

Returns the tuning recommendation results from the last ***ANALYZE_WORKLOAD()*** (page 443) call. This information is useful for letting you build filters on the Workload Analyzer result set.

Column	Data type	Description
observation_count	INTEGER	Integer for the total number of events observed for this tuning recommendation. For example, if you see a return value of 1, WLA is making its first tuning recommendation for the event in 'scope'.
first_observation_time	TIMESTAMPTZ	Timestamp when the event first occurred. If this column returns a null value, the tuning recommendation is from the current status of the system instead of from any prior event.
last_observation_time	TIMESTAMPTZ	Timestamp when the event last occurred. If this column returns a null value, the tuning recommendation is from the current status of the system instead of from any prior event.

tuning_parameter	VARCHAR	Objects on which you should perform a tuning action. For example, a return value of: ▪ **public.t** informs the DBA to run Database Designer on table t in the public schema ▪ **bsmith** notifies a DBA to set a password for user bsmith
tuning_description	VARCHAR	Textual description of the tuning recommendation from the Workload Analyzer to perform on the tuning_parameter object. Examples of some of the returned values include, but are not limited to: ▪ Run database designer on table `schema.table` ▪ Create replicated projection for table `schema.table` ▪ Consider query-specific design on query ▪ Reset configuration parameter with `SELECT set_config_parameter('parameter', 'new_value')` ▪ Re-segment projection `projection-name` on high-cardinality column(s) ▪ Drop the projection `projection-name` ▪ Alter a table's partition expression ▪ Reorganize data in partitioned table ▪ Decrease the MoveOutInterval configuration parameter setting

tuning_command	VARCHAR	Command string if tuning action is a SQL command. For example, the following example statements recommend that the DBA:
		Update statistics on a particular schema's table.column:
		`SELECT ANALYZE_STATISTICS('public.table.column');`
		Resolve mismatched configuration parameter 'LockTimeout':
		`SELECT * FROM CONFIGURATION_PARAMETERS WHERE parameter_name = 'LockTimeout';`
		Set the password for user bsmith:
		`ALTER USER (user) IDENTIFIED BY ('new_password');`

tuning_cost	VARCHAR	Cost is based on the type of tuning recommendation and is one of: ▪ LOW—minimal impact on resources from running the tuning command ▪ MEDIUM—moderate impact on resources from running the tuning command ▪ HIGH—maximum impact on resources from running the tuning command Depending on the size of your database or table, consider running high-cost operations after hours instead of during peak load times.

Permissions

Must be a superuser.

Examples

For examples, see ***ANALYZE_WORKLOAD()*** (page 443)

See Also

Monitoring and analyzing workloads and WLA's triggering conditions and recommendations in the Administrator's Guide

TUPLE_MOVER_OPERATIONS

Monitors the status of the Tuple Mover (TM) on each node.

Column Name	Data Type	Description
OPERATION_START_TIMESTAMP	TIMESTAMP	Start time of a Tuple Mover operation.
NODE_NAME	VARCHAR	Node name for which information is listed.

OPERATION_NAME	VARCHAR	One of the following operations:
		`Moveout`
		`Mergeout`
		`Analyze Statistics`

OPERATION_STATUS	VARCHAR	Returns `Running` or an empty string to indicate 'not running.'
TABLE_SCHEMA	VARCHAR	Schema name for the specified projection.
TABLE_NAME	VARCHAR	Table name for the specified projection
PROJECTION_NAME	VARCHAR	Name of the projection being processed.
PROJECTION_ID	INTEGER	Unique numeric ID assigned by the HP Vertica catalog, which identifies the projection.
COLUMN_ID	INTEGER	Identifier for the column for the associated

		projection being processed.
EARLIEST_CONTAINER_START_EPOCH	INTEGER	Populated for mergeout, purge and merge_partitions operations only. For an ATM-invoked mergeout, for example, the returned value represents the lowest epoch of containers involved in the mergeout.
LATEST_CONTAINER_END_EPOCH	INTEGER	Populated for mergeout, purge and merge_partitions operations only. For an ATM-invoked mergeout, for example, the returned value represents the highest epoch of containers involved in the mergeout.
ROS_COUNT	INTEGER	Number of ROS containers.
TOTAL_ROS_USED_BYTES	INTEGER	Size in bytes of all ROS containers in the mergeout operation. (Not applicable for other operations.)
PLAN_TYPE	VARCHAR	One of the following values: Moveout Mergeout Analyze Replay Delete
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
IS_EXECUTING	BOOLEAN	Distinguishes between actively-running (t) and completed (f) tuple mover operations.
RUNTIME_PRIORITY	VARCHAR	Determines the amount of run-time resources (CPU, I/O bandwidth) the Resource Manager should dedicate to running queries in the resource pool. Valid values are: ▪ HIGH ▪ MEDIUM ▪ LOW

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Notes

Manual mergeouts are invoked using one of the following APIs:

- ***DO_TM_TASK*** (page 471)()
- ***PURGE*** (page 517)
- ***MERGE_PARTITIONS*** (page 513)

Example

```
=> SELECT node_name, operation_status, projection_name, plan_type
```

```
   FROM TUPLE_MOVER_OPERATIONS;

 node_name | operation_status | projection_name  | plan_type
-----------+------------------+------------------+-----------
 node0001  | Running          | p1_b2            | Mergeout
 node0002  | Running          | p1               | Mergeout
 node0001  | Running          | p1_b2            | Replay Delete
 node0001  | Running          | p1_b2            | Mergeout
 node0002  | Running          | p1_b2            | Mergeout
 node0001  | Running          | p1_b2            | Replay Delete
 node0002  | Running          | p1               | Mergeout
 node0003  | Running          | p1_b2            | Replay Delete
 node0001  | Running          | p1               | Mergeout
 node0002  | Running          | p1_b1            | Mergeout
```

See Also

DO_TM_TASK (page 471), ***MERGE_PARTITIONS*** (page 513), and ***PURGE*** (page 517)

Understanding the Tuple Mover and Partitioning Tables in the Administrator's Guide

UDX_FENCED_PROCESSES

Provides information about processes HP Vertica uses to run user-defined extensions in fenced mode.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
PROCESS_TYPE	VARCHAR	Indicates what kind of side process this row is for and can be one of the following values: ▪ `UDxZygoteProcess` — Master process that creates worker side processes, as needed, for queries. There will be, at most, 1 UP UDxZygoteProcess for each HP Vertica instance. ▪ `UDxSideProcess` — Indicates that the process is a worker side process. There could be many UDxSideProcesses, depending on how many sessions there are, how many queries, and so on.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.
LANGUAGE	VARCHAR	The language of the UDx. For example 'R' or 'C++';

PID	INTEGER	Linux process identifier of the side process (UDxSideProcess).
PORT	VARCHAR	For HP Vertica internal use. The TCP port that the side process is listening on.
STATUS	VARCHAR	Can be one of "UP" or "DOWN", depending on whether the process is alive or not. **Note:** If a process fails, HP Vertica will reap it some time in the future, but not necessarily immediately, so the status could appear as "DOWN." Also, once a process fails, HP Vertica restarts it only on demand. So after a process failure, there could be periods of time when no side processes are running.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> select * from udx_fenced_processes;
   node_name   |   process_type   |            session_id            |
language |  pid  | port  | status
---------------+------------------+----------------------------------+--------
--+-------+-------+--------
 v_db_node0001 | UDxZygoteProcess |                                  |
|  3137 | 56667 | UP
 v_db_node0001 | UDxSideProcess   | localhost.localdoma-3117:0x15746 | R
| 41821 | 34040 | UP
(2 rows)
```

USER_LIBRARIES

Lists the user libraries that are currently loaded.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	The name of the schema containing the library
LIB_NAME	VARCHAR	The name of the library
LIB_OID	INTEGER	The object ID of the library
OWNER_ID	INTEGER	The object ID of the library's owner
LIB_FILE_NAME	VARCHAR	The name of the shared library file
MD5_SUM	VARCHAR	The MD5 checksum of the library file, used to ensure that the file was correctly copied to each node

SDK_VERSION	VARCHAR	The version of the HP Vertica SDK used to compile the library.
REVISION	VARCHAR	The revision of the HP Vertica SDK used to compile the library.

USER_LIBRARY_MANIFEST

Lists the User Defined Functions contained in all of the loaded user libraries.

Column Name	Data Type	Description
SCHEMA_NAME	VARCHAR	The name of the schema containing the function.
LIB_NAME	VARCHAR	The name of the library containing the UDF.
LIB_OID	INTEGER	The object ID of the library containing the function.
OBJ_NAME	VARCHAR	The name of the constructor class in the library for a function.
OBJ_TYPE	VARCHAR	The type of user defined function (scalar function, transform function)
ARG_TYPES	VARCHAR	A comma-delimited list of data types of the function's parameters.
RETURN_TYPE	VARCHAR	A comma-delimited list of data types of the function's return values.

USER_SESSIONS

Returns user session history on the system.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	Name of the node that is reporting the requested information.
USER_NAME	VARCHAR	Name of the user at the time HP Vertica recorded the session.
SESSION_ID	VARCHAR	Identifier for this session. This identifier is unique within the cluster at any point in time but can be reused when the session closes.

TRANSACTION_ID	VARCHAR	Identifier for the transaction within the session, if any. If a session is active but no transaction has begun, TRANSACTION_ID returns NULL.
STATEMENT_ID	VARCHAR	Unique numeric ID for the currently-running statement. NULL indicates that no statement is currently being processed. The combination of TRANSACTION_ID and STATEMENT_ID uniquely identifies a statement within a session.
SESSION_START_TIMESTAMP	TIMESTAMPTZ	Value of session at beginning of history interval.
SESSION_END_TIMESTAMP	TIMESTAMPTZ	Value of session at end of history interval.
IS_ACTIVE	BOOLEAN	Denotes if the operation is executing.
CLIENT_HOSTNAME	VARCHAR	IP address of the client system
CLIENT_PID	INTEGER	Linux process identifier of the client process that issued this connection. **Note:** The client process could be on a different machine from the server.
CLIENT_LABEL	VARCHAR	User-specified label for the client connection that can be set when using ODBC. See Label in DSN Parameters in Programmer's Guide.
SSL_STATE	VARCHAR	Indicates if HP Vertica used Secure Socket Layer (SSL) for a particular session. Possible values are: ▪ None – Vertica did not use SSL. ▪ Server – Sever authentication was used, so the client could authenticate the server. ▪ Mutual – Both the server and the client authenticated one another through mutual authentication. See Implementing Security and Implementing SSL in the Administrator's Guide.

AUTHENTICATION_METHOD	VARCHAR	Type of client authentication used for a particular session, if known. Possible values are: ▪ Unknown ▪ Trust ▪ Reject ▪ Kerberos ▪ Password ▪ MD5 ▪ LDAP ▪ Kerberos-GSS ▪ Ident See Implementing Security and Implementing Client Authentication.

Permissions

No explicit permissions are required; however, users see only the records that correspond to tables they have permissions to view.

Example

```
=> SELECT * FROM USER_SESSIONS;

-[ RECORD 2 ]-----------+---------------------------------
node_name               | v_vmart_node0001
user_name               | dbadmin
session_id              | 000000.verticacorp.-30972:0x15
transaction_id          |
statement_id            |
session_start_timestamp | 2012-12-10 06:22:39.539826-05
session_end_timestamp   | 2012-12-10 06:22:39.844873-05
is_active               | f
client_hostname         | 10.20.100.62:41307
client_pid              | 31517
client_label            |
ssl_state               | None
authentication_method   | Password
```

See also

CURRENT_SESSION (page 999)

SESSION_PROFILES (page 1093)

SESSIONS (page 1095)

SYSTEM_SESSIONS (page 1116)

Implementing Security in the Administrator's Guide

WOS_CONTAINER_STORAGE

Monitors information about WOS storage, which is divided into regions. Each region allocates blocks of a specific size to store rows.

Column Name	Data Type	Description
NODE_NAME	VARCHAR	The node name for which information is listed.
WOS_TYPE	VARCHAR	Returns one of the following: ▪ `system` – for system table queries ▪ `user` – for other user queries
WOS_ALLOCATION_REGION	VARCHAR	The block size allocated by region in KB. The summary line sums the amount of memory used by all regions.
REGION_VIRTUAL_SIZE_KB	INTEGER	The amount of virtual memory in use by region in KB. Virtual size is greater than or equal to allocated size, which is greater than or equal to in-use size.
REGION_ALLOCATED_SIZE_KB	INTEGER	The amount of physical memory in use by a particular region in KB.
REGION_IN_USE_SIZE_KB	INTEGER	The actual number of bytes of data stored by the region in KB.
REGION_SMALL_RELEASE_COUNT	INTEGER	Internal use only.
REGION_BIG_RELEASE_COUNT	INTEGER	Internal use only.
EXTRA_RESERVED_BYTES	INTEGER	The amount of extra memory allocated to maintain WOS sort information.
EXTRA_USED_BYTES	INTEGER	The amount of memory in use currently to maintain the WOS sort information.

Notes

- The WOS allocator can use large amounts of virtual memory without assigning physical memory.
- To see the difference between virtual size and allocated size, look at the `REGION_IN_USE_SIZE` column to see if the WOS is full. The summary line tells you the amount of memory used by the WOS, which is typically capped at one quarter of physical memory per node.

Examples

```
=>\pset expanded
Expanded display is on.
=> SELECT * FROM WOS_CONTAINER_STORAGE;
```

```
-[ RECORD 1 ]--------------+---------------------------
node_name                  | host01
wos_type                   | user
wos_allocation_region      | 16 KB Region
region_virtual_size_kb     | 2045408
region_allocated_size_kb   | 0
region_in_use_size_kb      | 0
region_small_release_count | 656
region_big_release_count   | 124
-[ RECORD 2 ]--------------+---------------------------
node_name                  | host01
wos_type                   | system
wos_allocation_region      | 16 KB Region
region_virtual_size_kb     | 1024
region_allocated_size_kb   | 960
region_in_use_size_kb      | 0
region_small_release_count | 78
region_big_release_count   | 9
-[ RECORD 3 ]--------------+---------------------------
node_name                  | host01
wos_type                   | system
wos_allocation_region      | 64 KB Region
region_virtual_size_kb     | 1024
region_allocated_size_kb   | 64
region_in_use_size_kb      | 0
region_small_release_count | 19
region_big_release_count   | 0
-[ RECORD 4 ]--------------+---------------------------
node_name                  | host01
wos_type                   | system
wos_allocation_region      | Summary
region_virtual_size_kb     | 2048
region_allocated_size_kb   | 1024
region_in_use_size_kb      | 0
region_small_release_count | 97
region_big_release_count   | 9
-[ RECORD 5 ]--------------+---------------------------
node_name                  | host01
wos_type                   | user
wos_allocation_region      | Summary
region_virtual_size_kb     | 2045408
region_allocated_size_kb   | 0
region_in_use_size_kb      | 0
region_small_release_count | 656
region_big_release_count   | 124
-[ RECORD 6 ]--------------+---------------------------
node_name                  | host02
wos_type                   | user
wos_allocation_region      | 16 KB Region
region_virtual_size_kb     | 2045408
region_allocated_size_kb   | 0
region_in_use_size_kb      | 0
region_small_release_count | 666
region_big_release_count   | 121
```

```
-[ RECORD 7 ]--------------+---------------------------
node_name                  | host02
wos_type                   | system
wos_allocation_region      | 16 KB Region
region_virtual_size_kb     | 1024
region_allocated_size_kb   | 960
region_in_use_size_kb      | 0
region_small_release_count | 38
region_big_release_count   | 2
-[ RECORD 8 ]--------------+---------------------------
node_name                  | host02
wos_type                   | system
wos_allocation_region      | 64 KB Region
region_virtual_size_kb     | 1024
region_allocated_size_kb   | 64
region_in_use_size_kb      | 0
region_small_release_count | 10
region_big_release_count   | 0
-[ RECORD 9 ]--------------+---------------------------
...
```

Appendix: Compatibility with Other RDBMS

This section describes compatibility of HP Vertica with other relational database management systems.

Information in this appendix is intended to simplify database migration to HP Vertica.

Data Type Mappings Between Vertica and Oracle

Oracle uses proprietary data types for all main data types (for example, VARCHAR, INTEGER, FLOAT, DATE), if you plan to migrate your database from Oracle to HP Vertica, HP strongly recommends that you convert the schema—a simple and important exercise that can minimize errors and time lost spent fixing erroneous data issues.

The following table compares the behavior of Oracle data types to HP Vertica data types.

Oracle	Vertica	Notes
NUMBER (no explicit precision)	INT, NUMERIC or FLOAT	In Oracle, the NUMBER data type with no explicit precision stores each number N as an integer M, together with a scale S. The scale can range from -84 to 127, while the precision of M is limited to 38 digits. So N = M * 10^S. When precision is specified, precision/scale applies to all entries in the column. If omitted, the scale defaults to 0. For the common case where Oracle's NUMBER with no explicit precision data type is used to store only integer values, INT is the best suited and the fastest Vertica data type. However, INT (the same as BIGINT) is limited to a little less than 19 digits, with a scale of 0; if the Oracle column contains integer values outside of the range [-9223372036854775807, +9223372036854775807], use the Vertica data type NUMERIC(p,0) where p is the maximum number of digits required to represent the values of N. Even though no explicit scale is specified for an Oracle NUMBER column, Oracle allows non-integer values, each with its own scale. If the data stored in the column is approximate, Vertica recommends using the Vertica data type FLOAT, which is standard IEEE floating point, like ORACLE BINARY_DOUBLE. If the data is exact with fractional places, for example dollar amounts, Vertica recommends NUMERIC(p,s) where p is the precision (total number of digits) and s is the maximum scale (number of decimal places). Vertica conforms to standard SQL, which requires that p >= s and s >= 0. Vertica's NUMERIC data type is most effective for p=18, and increasingly expensive for p=37, 58, 67, etc., where p <= 1024. Vertica recommends against using the data type NUMERIC(38,s) as a default "failsafe" mapping to guarantee no loss of precision.

		NUMERIC(18,s) is better, and INT or FLOAT are better yet, if one of these data types will do the job.
NUMBER (P,0), P <= 18	INT	In Oracle, when precision is specified the precision/scale applies to all entries in the column. If omitted the scale defaults to 0. For the Oracle NUMBER data type with 0 scale, and a precision less than or equal to 18, use INT in Vertica.
NUMBER (P,0), P > 18	NUMERIC (p,0)	An Oracle column precision greater than 18 is often more than an application really needs. If all values in the Oracle column are within the INT range [-9223372036854775807,+9223372036854775807], use INT for best performance. Otherwise, use the Vertica data type NUMERIC(p, 0), where p = P.
NUMBER (P,S) all cases other than previous 3 rows	NUMERIC (p,s) or FLOAT	When P >= S and S >= 0, use p = P and s = S, unless the data allows reducing P or using FLOAT as discussed above. If S > P, use p = S, s = S. If S < 0, use p = P – S, s = 0.
NUMERIC (P,S)	See notes -->	Rarely used in Oracle. See notes for the NUMBER type.
DECIMAL (P,S)	See notes -->	DECIMAL is a synonym for NUMERIC. See notes for the NUMBER type.
BINARY_FLOAT	FLOAT	Same as FLOAT(53) or DOUBLE PRECISION.
BINARY_DOUBLE	FLOAT	Same as FLOAT(53) or DOUBLE PRECISION.
RAW	VARBINARY(RAW)	The maximum size of RAW in Oracle is 2,000 bytes. The maximum size of CHAR/BINARY in Vertica is 65000 bytes. In Vertica, RAW is a synonym for VARBINARY.
LONG RAW	VARBINARY(RAW)	The maximum size of Oracle's LONG RAW is 2GB. The maximum size of Vertica's VARBINARY is 65000 bytes. Vertica user should exercise caution to avoid truncation during data migration from Oracle.
CHAR(n)	CHAR(n)	The maximum size of CHAR in Oracle is 2,000 bytes. The maximum size of CHAR in Vertica is 65000 bytes.
NCHAR(n)	CHAR(n*3)	Vertica supports national characters with CHAR(n) as variable-length UTF8-encoded UNICODE character string. UTF-8 represents ASCII in 1 byte, most European characters in 2 bytes, and most oriental and Middle Eastern characters in 3 bytes.
VARCHAR2(n)	VARCHAR(n)	The maximum size of VARCHAR2 in Oracle is 4,000 bytes. The maximum size of VARCHAR in Vertica is 65000 . **Note:** The behavior of Oracle's VARCHAR2 and Vertica's VARCHAR is semantically different. Vertica's VARCHAR exhibits standard SQL behavior, whereas Oracle's VARCHAR2 is not completely consistent with standard behavior – it treats an empty string as NULL value and uses non-padded comparison if one operand is VARCHAR2.
NVARCHAR2 (n)	VARCHAR(n*3)	See notes for NCHAR().

DATE	TIMESTAMP or possibly DATE	Oracle's DATE is different from the SQL standard DATE data type implemented by Vertica. Oracle's DATE includes the time (no fractional seconds), while Vertica DATE type includes only date per SQL specification.
TIMESTAMP	TIMESTAMP	TIMESTAMP defaults to six places, that is, to microseconds
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE	TIME ZONE defaults to the currently SET or system time zone.
INTERVAL YEAR TO MONTH	INTERVAL YEAR TO MONTH	Per the SQL standard, INTERVAL can be qualified with YEAR TO MONTH sub-type in Vertica.
INTERVAL DAY TO SECOND	INTERVAL DAY TO SECOND	In Vertica, DAY TO SECOND is the default sub-type for INTERVAL.

Copyright Notice

Hewlett-Packard
150 CambridgePark Drive
Cambridge, MA 02140
Phone: +1 617 386 4400
E-Mail: info@vertica.com
Web site: ***http://www.vertica.com***
(***http://www.vertica.com***)

Trademarks

HP Vertica™, the HP Vertica Analytics Platform™, and FlexStore™ are trademarks of Hewlett-Packard.

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.

DataDirect® and DataDirect Connect® are registered trademarks of Progress Software Corporation in the U.S. 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.

Information on third-party software used in HP Vertica, including details on open-source software, is available in the guide Third-Party Software Acknowledgements.