Informatica

Informatica® Data Archive
6.4.3 HotFix 1

Data Vault SQL Reference

Informatica Data Archive Data Vault SQL Reference
6.4.3 HotFix 1
June 2017

Publication Date: 2018-07-02

Table of Contents

Preface

The Data Vault SQL Reference provides information about the SQL commands that can be used to access the Data Vault in Data Archive. The SQL Reference is written for users who use query tools or reporting tools to access data in an optimized archive.

This guide assumes that you have knowledge of SQL and relational database concepts, and the database, flat file, or mainframe systems in your environment.

Informatica Resources

Informatica Network

Informatica Network hosts Informatica Global Customer Support, the Informatica Knowledge Base, and other product resources. To access Informatica Network, visit https://network.informatica.com.

As a member, you can:

- Access all of your Informatica resources in one place.
- Search the Knowledge Base for product resources, including documentation, FAQs, and best practices.
- View product availability information.
- Review your support cases.
- Find your local Informatica User Group Network and collaborate with your peers.

Informatica Knowledge Base

Use the Informatica Knowledge Base to search Informatica Network for product resources such as documentation, how-to articles, best practices, and PAMs.

To access the Knowledge Base, visit https://kb.informatica.com. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team at KB_Feedback@informatica.com.

Informatica Documentation

To get the latest documentation for your product, browse the Informatica Knowledge Base at https://kb.informatica.com/_layouts/ProductDocumentation/Page/ProductDocumentSearch.aspx.

If you have questions, comments, or ideas about this documentation, contact the Informatica Documentation team through email at infa_documentation@informatica.com.

Informatica Product Availability Matrixes

Product Availability Matrixes (PAMs) indicate the versions of operating systems, databases, and other types of data sources and targets that a product release supports. If you are an Informatica Network member, you can access PAMs at https://network.informatica.com/community/informatica-network/product-availability-matrices.

Informatica Velocity

Informatica Velocity is a collection of tips and best practices developed by Informatica Professional Services. Developed from the real-world experience of hundreds of data management projects, Informatica Velocity represents the collective knowledge of our consultants who have worked with organizations from around the world to plan, develop, deploy, and maintain successful data management solutions.

If you are an Informatica Network member, you can access Informatica Velocity resources at http://velocity.informatica.com.

If you have questions, comments, or ideas about Informatica Velocity, contact Informatica Professional Services at ips@informatica.com.

Informatica Marketplace

The Informatica Marketplace is a forum where you can find solutions that augment, extend, or enhance your Informatica implementations. By leveraging any of the hundreds of solutions from Informatica developers and partners, you can improve your productivity and speed up time to implementation on your projects. You can access Informatica Marketplace at https://marketplace.informatica.com.

Informatica Global Customer Support

You can contact a Global Support Center by telephone or through Online Support on Informatica Network.

To find your local Informatica Global Customer Support telephone number, visit the Informatica website at the following link:
http://www.informatica.com/us/services-and-training/support-services/global-support-centers.

If you are an Informatica Network member, you can use Online Support at http://network.informatica.com.

CHAPTER 1

Introduction to SQL Reference for Informatica Data Vault

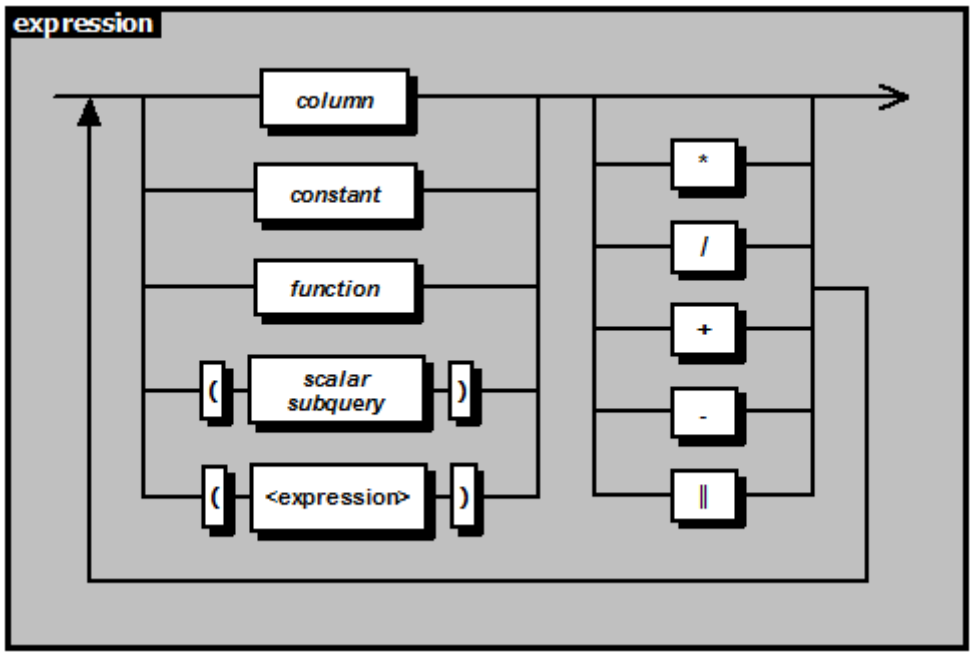
This chapter includes the following topics:

Value Expressions

Value expressions can be used anywhere Data Vault Service SQL syntax calls for a *value expression*, for example, in the projection list of the SELECT statement. A value expression is an argument that evaluates to a single numeric, character string, or date/time value (the datatype of the expression must be appropriate to the statement in which it is used). Value expression *primaries*, which are the "building blocks" of the value expression argument, consist of direct column references, character or numeric constants, functions, the arithmetic operators **+**, **–**, *****, **/**, and the string concatenation operator ||. A value expression may also contain another value expression enclosed in parentheses.

Note that only numeric values may be part of an arithmetic operation; only character values can be used with the concatenation operator.

The value expression argument is described in the following syntax diagram:



column

The column argument specifies the name of a column belonging to the archived table.

constant

The constant argument specifies a character or numeric literal. A numeric value may be preceded by a unary operator (that is, a + or –) to indicate whether it is a positive or negative value. A character string must be surrounded by single quotes. Only numeric values may be part of an arithmetic operation; only character values can be used with the concatenation operator.

function

A function can be applied to all input values, resulting in a single output value.

scalar subquery

The scalar subquery argument must return a single value only. If more than one value is returned, an error condition is generated. If the subquery result set is empty, a null value is returned instead.

Note that the entire SELECT statement must be contained in parentheses. Also, a correlated subquery is not permitted at this time.

Numeric Value Expressions

Arithmetic operators are evaluated within an expression from left to right, in the following order of precedence:

1. Unary Plus (+) and Minus (-)
2. Multiplication (*) and Division (/)
3. Addition (+) and Subtraction (-)

Parentheses may be employed, however, to override the order of evaluation. Expressions within parentheses are evaluated before other parts of the expression. Within a level of precedence, operations are performed in order from left to right. Expressions that use arithmetic operators require numeric operands, and always produce numeric values.

String Value Expressions

A string value expression is an expression that evaluates to a character string using some combination of direct references to CHAR or VARCHAR columns, string literals, and string functions. The concatenation operator (||) is used to combine the string value expression primaries (as described above). The maximum size of the result string is **4056** characters.

For example, the following expression

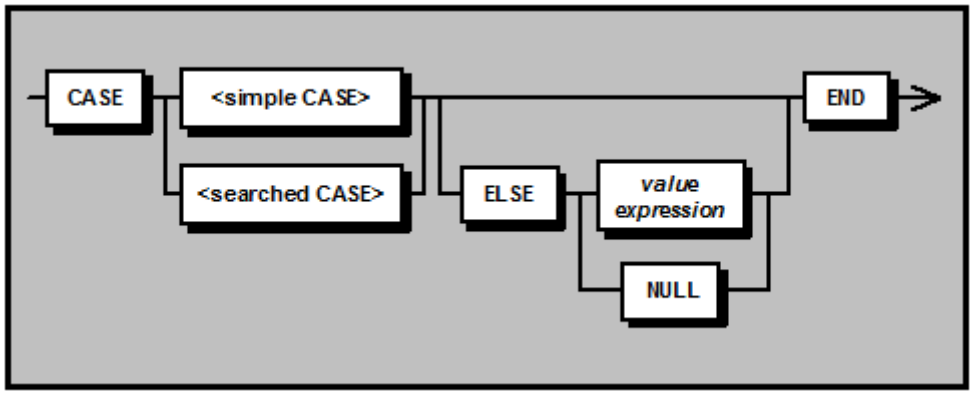
```
UPPER ("hello" || "world")
```

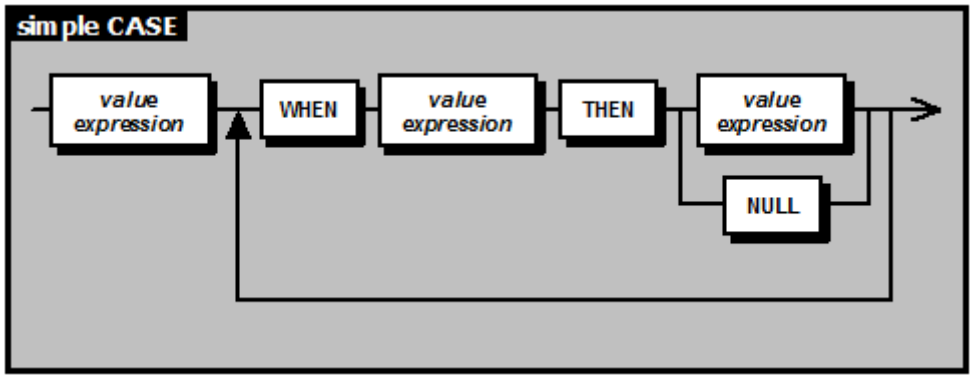
returns the result string "HELLOWORLD".

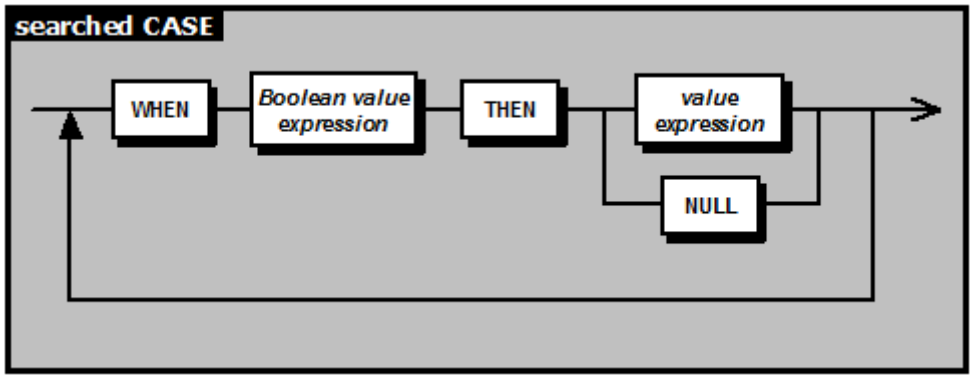
Case Expressions

The CASE statement is a conditional expression that can be used anywhere a value expression is used. It allows for the evaluation of multiple conditions, returning a specified value when a condition is true, or a default value when none of the conditions is true. There are three functions, NULLIF, COALESCE, and IFNULL, that are designed to handle a subset of the CASE functionality.

The following syntax diagrams describe the CASE statement:







As indicated in the syntax diagrams, there are two types of CASE statements: *simple* and *searched*.

Simple CASE

In the simple CASE, a value expression follows the CASE keyword. This expression is tested for equality against the value expression in each WHEN clause. If a truth condition is encountered, testing of any remaining conditions is halted and the value expression for the current THEN clause is returned. If no comparison evaluates to true, then the default value expression in the ELSE clause is returned. If there is no ELSE clause, a null value is returned. At least one of the THEN clauses must specify a value other than NULL. All comparisons must involve compatible datatypes.

Example of a Simple CASE

```
SELECT lastname,
       CASE dept
          WHEN 'A' THEN 'Administration'
          WHEN 'D' THEN 'Development'
          WHEN 'Q' THEN 'Quality Assurance'
          WHEN 'T' THEN 'Technical Writing'
          ELSE '(unknown department)'
       END
FROM employees;
```

In this example, the *employees* table is searched for employee last name and department. But instead of returning the single character that represents a department, a CASE statement is set up to return the full (or default) department name. If the *dept* value is 'A', the string 'Administration' is returned; if the *dept* value is 'D', 'Development' is returned; and so on. If the *dept* value matches none of the single characters tested in the CASE statement, the default string '(unknown department)' is returned instead.

Searched CASE

In contrast to the simple CASE, the searched CASE does not use a single test expression to compare for equality. Instead, the searched CASE permits any Boolean condition, using any comparison operator(s), to appear in each WHEN clause. As in the simple CASE, the first condition to evaluate to true returns the value expression in the associated THEN clause. If no comparison evaluates to true, then the default value expression in the ELSE clause is returned. If there is no ELSE clause, a null value is returned. At least one of the THEN clauses must specify a value other than NULL. All comparisons must involve compatible datatypes.

Note that a simple CASE can always be rewritten as a searched CASE in the following manner:

You can rewrite it to be a simple CASE expression as follows:

```
CASE <value-exp-0>
  WHEN <compare-value-exp-1> THEN <return-value-exp-1>
  WHEN <compare-value-exp-2> THEN <return-value-exp-2>
  ...
  WHEN <compare-value-exp-n> THEN <return-value-exp-n>
  ELSE <value-exp-x>
END
```

You can rewrite it to be a searched CASE expression as follows:

```
CASE
  WHEN <value-exp-0> = <compare-value-exp-1> THEN <return-value-exp-1>
  WHEN <value-exp-0> = <compare-value-exp-2> THEN <return-value-exp-2>
  ...
  WHEN <value-exp-0> = <compare-value-exp-n> THEN <return-value-exp-n>
  ELSE <value-exp-x>
END
```

Example of a Searched CASE

```
SELECT name,
       CASE
          WHEN age < 0 THEN '(unknown)'
          WHEN age < 18 THEN 'Youth'
          WHEN age >= 18 AND age < 65 THEN 'Adult'
          ELSE 'Senior'
       END
FROM people;
```

In this example, the *people* table is searched for name; as well, a character string that varies according to the age of the individual is returned with the name. Since ranges of ages are tested in the CASE statement in this example, as opposed to exact ages, a searched CASE statement is preferable to a simple CASE. Here, if the *age* value is less than zero, the character string '(unknown)' is returned; if the age is between 0 and 17 inclusive, 'Youth' is returned; and if the age is between 18 and 64 inclusive, 'Adult' is returned. If the age value does not fall into any of those ranges, then it must be 65 or greater, so the default value 'Senior' is returned instead. Note that the order of processing of the conditions can sometimes be important. In the above example, the condition "age < 0" must appear before "age < 18" in the CASE structure, otherwise a value of -1 will return 'Youth' instead of '(unknown)'.

SQL Syntax Diagrams

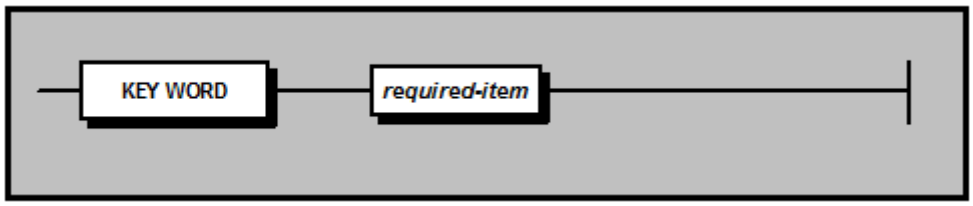
The following conventions are used in the SQL syntax diagrams:

- Uppercase letters indicate keywords (for example, INSERT). Keywords must be spelled exactly as presented, but are not case-sensitive, except where noted.

- Italicized words represent variables or arguments (for example, *table-name.column-name*) for which the user must substitute database object names or data values. In usage, these are only case-sensitive where specifically indicated.
- Non-italicized words or phrases between angle brackets (**< >**) indicate that the syntax for this element is expanded in a subsequent diagram labeled with the same word or phrase. When uppercase keywords are followed by the word “statement” (for example, <SELECT statement>), the syntax element is another SQL command statement entirely, and should be replaced by the appropriate syntax diagram.
- Partial diagrams are labeled with a word or phrase in the upper-left corner. These named diagrams should be substituted for the word or phrase wherever it is referenced in another syntax diagram.
- Parentheses **( )** appearing within the statement syntax are part of the syntax, and must be typed as shown. Punctuation marks such as commas (**,**) and colons (**:**) must be entered literally wherever they appear in syntax diagrams. Mathematical symbols, such as plus signs (**+**) and multiplication operators (*****), must also be entered literally.
- Underlined items indicate a default option. For example, if the options for SELECT are ALL and DISTINCT, SELECT by itself is the equivalent of SELECT ALL.

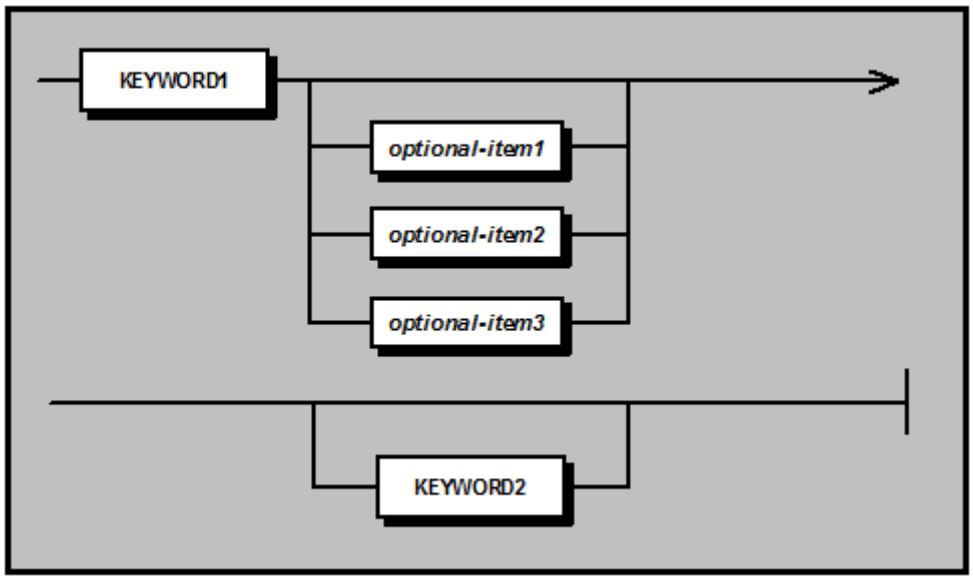
Required Syntax Elements

Required elements such as invocation commands and required arguments appear on the main path of the syntax diagram. A vertical bar signals the end of the syntax diagram, whereas an arrow signals continuation.

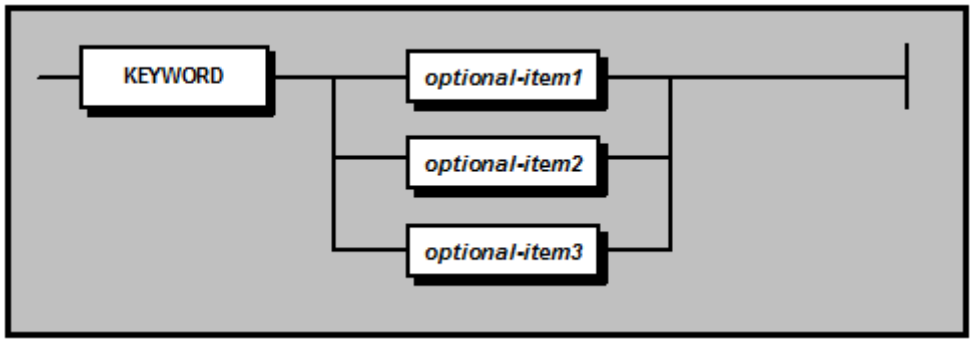


Optional Syntax Elements

Optional elements appear below the main path of the syntax diagram.

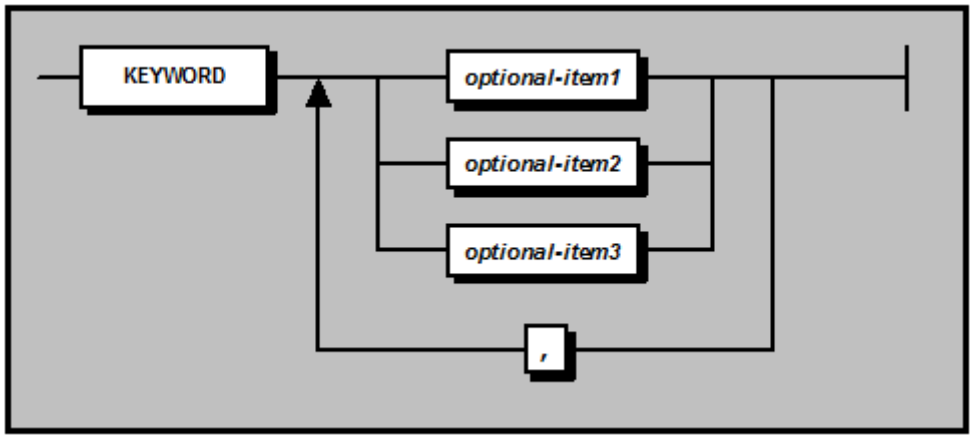


If there are multiple choices for a required element, one of the elements will be displayed on the main path of the diagram.



Repetitive Constructs

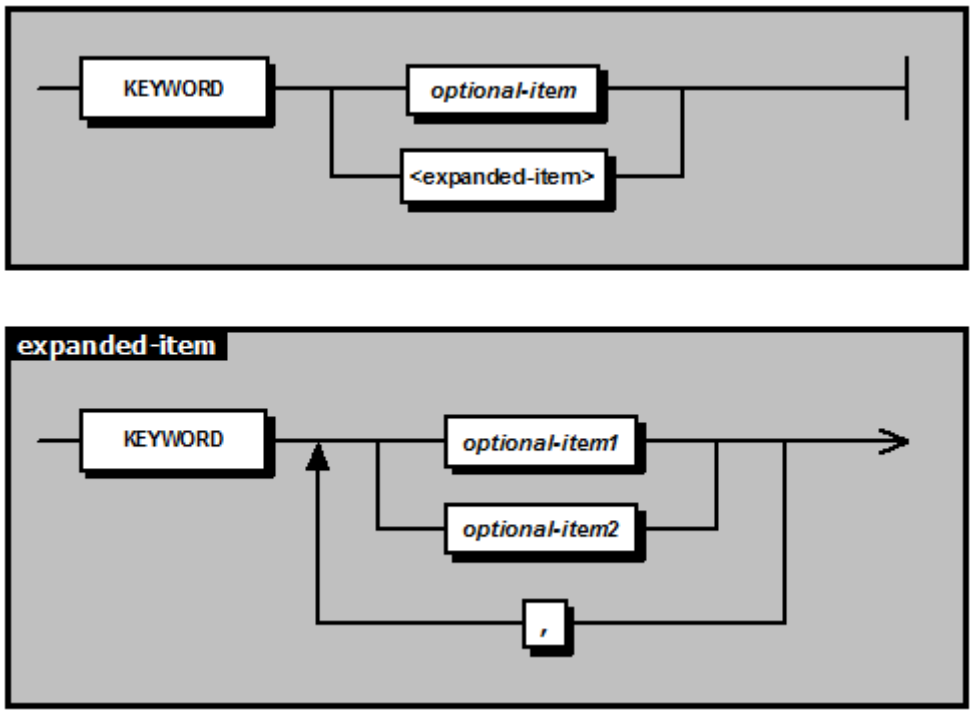
An arrow appears below a syntax element (or list of optional elements) when more than one element may be entered or when a particular element may be repeated. If the elements in the list must be separated by commas, a comma is shown in the syntax box.



Expanded Elements

Elements that are expanded in a subsequent syntax diagram are labeled with a word or phrase between angle brackets. The partial diagram that expands the element is labeled in the top left corner with the same word or

phrase found between angle brackets in the preceding diagram.



Usage Examples

Usage examples are shown in Data Vault Service SQL tool. In the SQL tool, statements must be terminated with a semicolon (;).

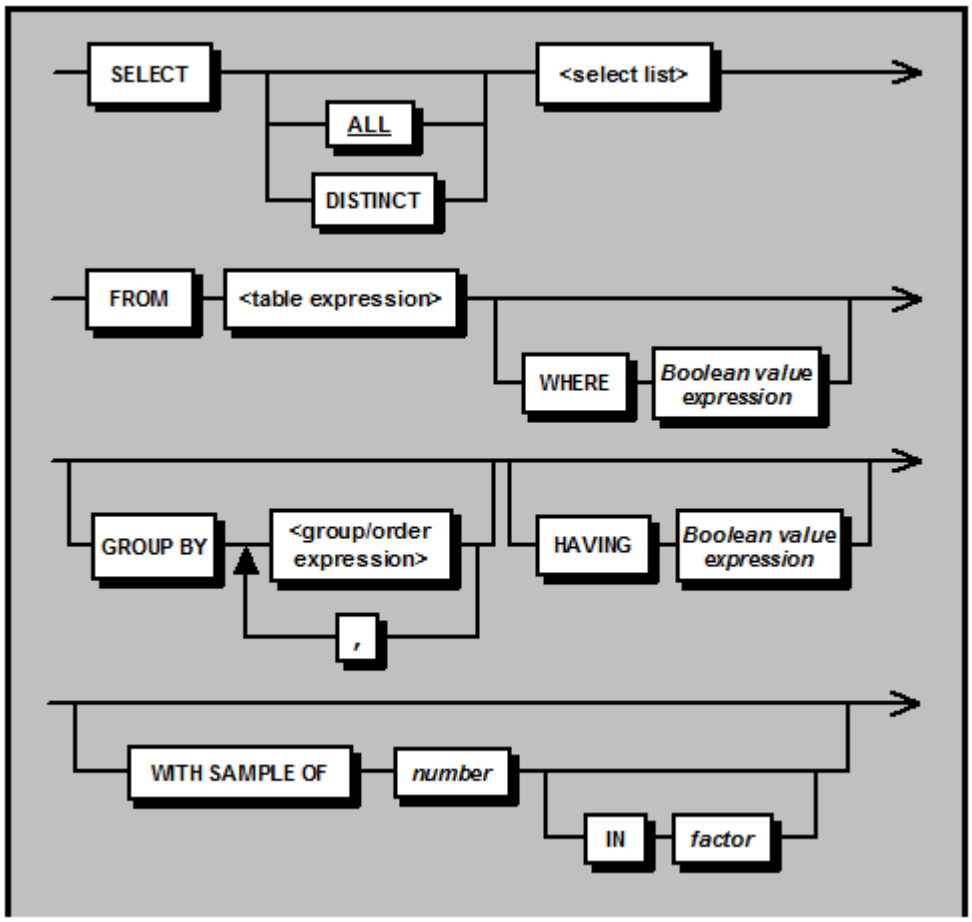
SELECT Statement Syntax

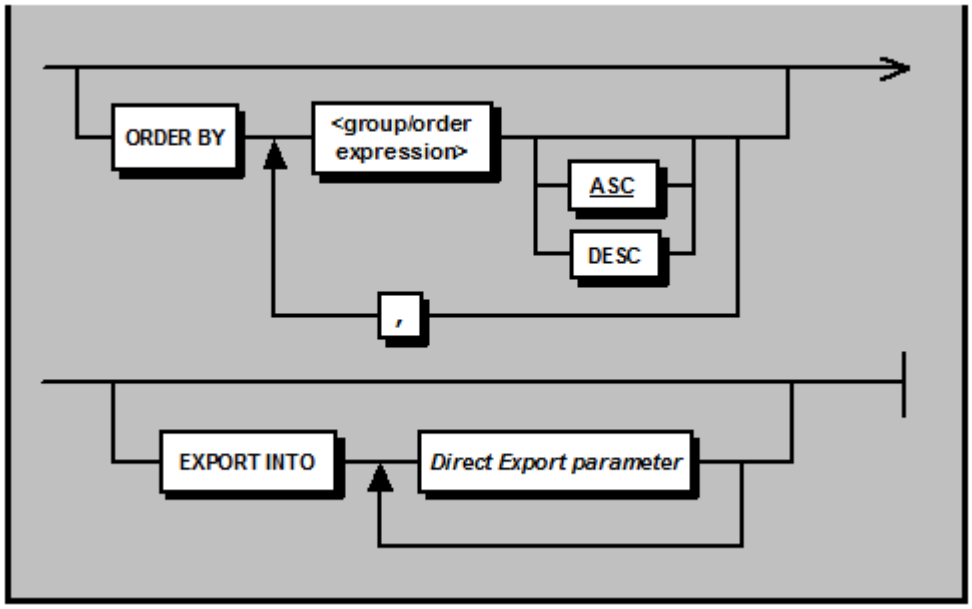
A SELECT statement, or query, is executed against a registered archived table to retrieve data from registered Data Vault data files, displaying this information in the form of a results table. In the case of a Direct Export, the results of the query are instead written to one or more output flat files.

Required Privileges

In order to select data from an archived table, the user authorization must own the table, own or possess OWNER privileges on the schema to which the table belongs, possess SELECT privileges on the table, or possess DBA privileges.

Syntax





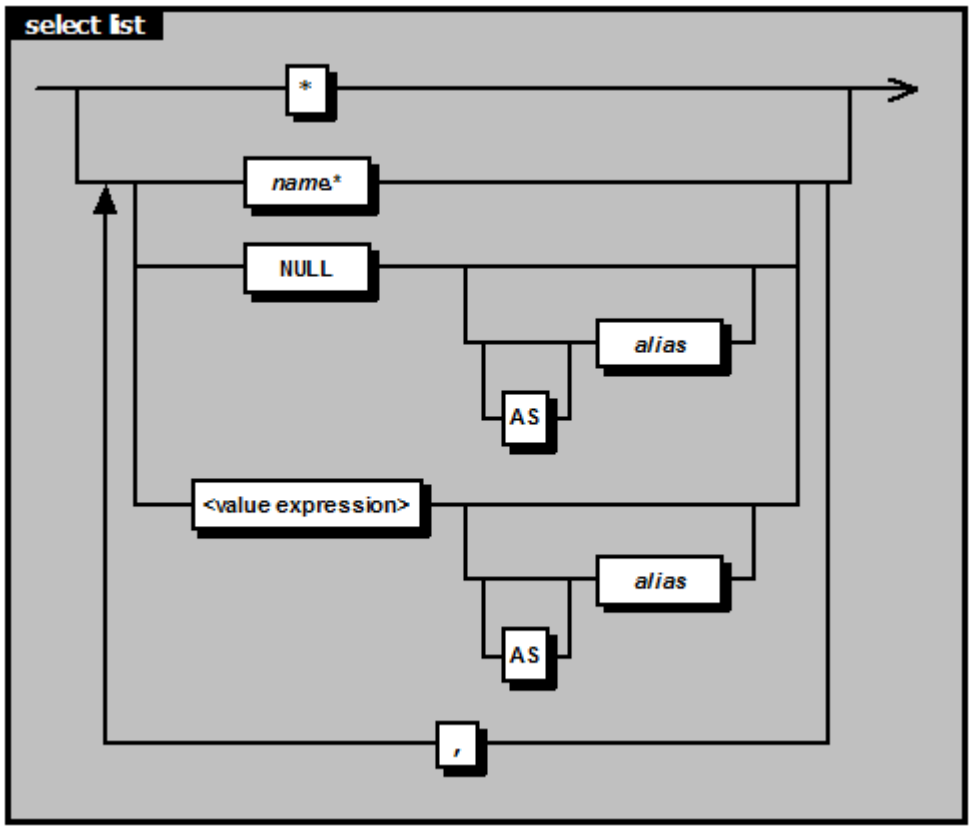
- Boolean value expression.

- Direct Export parameter.
  A parameter that describes the Direct Export operation. Each parameter can be set only once in the SQL statement. The PATHS and FILEPREFIX parameters must be specified; all other parameters are optional. The parameters can appear in any order.

  The following table describes the Direct Export parameters:

Parameter	Description
PATHS	Is the list of computer/directory locations where the flat files will be written in round robin fashion. Multiple paths can be specified, separated by semicolons (**;**) or by new lines. The whole parameter value must be contained in single quotation marks (' '). Note that paths can have spaces, but if the last directory in the path ends with a space, terminate it with a slash or backslash Each location must be accessible to all of the network computers running Data Vault Agent processes, otherwise the export operation will almost certainly fail.
FILEPREFIX (or FP)	Is the root name of the output flat file(s). Each flat file will be given the name *fileprefix*.*nnn*, where *nnn* is a number that increases incrementally with each new file from **000** to **NUMBEROFFILES-1**. The FILEPREFIX value must be contained in single quotation marks (' ').
NUMBEROFFILES (or NOF)	Is the number of flat files that will be created during the export (maximum: **1000**). Note that the size of each file will not necessarily be equal, as there are a number of factors influencing flat file size. Default. the number of active Data Vault Agent processes, which excludes those Data Vault Agents dedicated to internal tasks
ESCAPE (or ESC)	Is the character used to "escape" the column delimiter, row delimiter, null character, or the escape character itself in the exported data. When the escape character immediately precedes one of those special characters, that character should be interpreted as part of the data. The escape character is a single character, defined in hexadecimal form, and contained in single quotation marks (' '). For example, the hexadecimal '\x5c' represents the backslash character "\". Default is `\x1b` (the ESC character).
COLUMNDELIMITER (or CD)	Is the character that establishes column boundaries in the exported data. The column delimiter is a single character, defined in hexadecimal form, and contained in single quotation marks (' '). Default is `\x1f`.
ROWDELIMITER (or RD)	Is the character that signifies the end of a row and the start of a new one in the exported data. The row delimiter is a single character, defined in hexadecimal form, and contained in single quotation marks (' '). Default is `\x0a` (the linefeed character).
NULL	Is the character that represents a null value in the exported data. The null character is a single character, defined in hexadecimal form, and contained in single quotation marks (' '). Default is `\x7f`.

Select List Clause



*****

To retrieve all columns from the archived table, enter a single asterisk (*) as the sole argument of the *select list* clause.

name*.

When selecting columns from one or more tables or views, you can select all columns from a particular table by qualifying the special asterisk character with the appropriate table or correlation name specified in the FROM clause.

NULL

A derived column consisting entirely of null values can be specified using the NULL keyword.

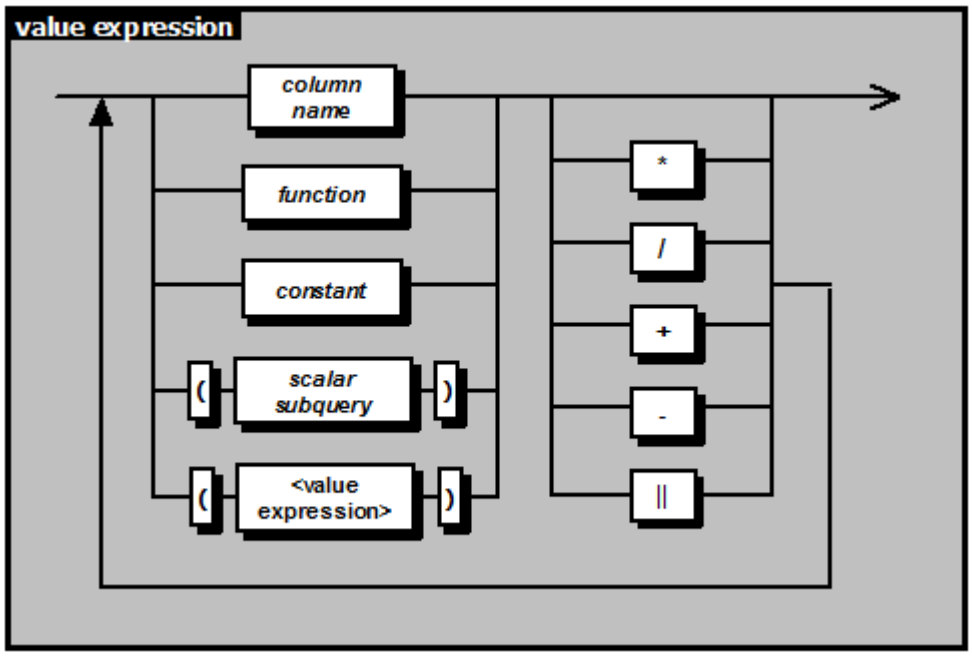
value expression

Typically, unless * is used (see above), one or more value expressions, separated by commas, make up the *select list* portion of the statement. These value expressions can be column names, functions, character or numeric constants, or some arithmetic combination of columns, functions, and constants. Expressions may be optionally enclosed by parentheses and nested.

alias

A name for the output of the value expression or the NULL column. If no alias is specified and the value expression is a direct column reference, the name of the output column will be the column name referenced. If no alias is specified and the value expression is not a direct column reference, the name of the output column will be its position number in the output table, counting from left to right.

Value Expression Clause



column name

This specifies a table column from which data values are to be retrieved by the query.

function

The *function* argument can be an *aggregate*, *cast*, *string*, *math*, or *date* function. The function is applied to a value expression argument, producing a derived column as its result. Consult the Functions section for a description of the available functions.

constant

A numeric or string constant can also be specified in the select list. The effect on the output table is a derived column where the constant is the value for each row in that column. This might be useful, for example, in labeling the output. String literals must be contained between single quotes.

(*scalar subquery*)

A complete SELECT statement that returns a single value. If more than one row is returned by this subquery, an error will be generated and the outer query will fail. If the subquery result set is empty, a null value is returned. The entire scalar subquery must be contained in parentheses.

Note that a *correlated* subquery is not permitted at this time.

expression operators

The + (addition), - (subtraction), * (multiplication), and / (division) operators are used to construct arithmetic expressions. The || (concatenation) operator is used to append one string value expression to the end of another. An exception is that aggregate functions cannot be part of an arithmetic expression.

Table Expression Clause

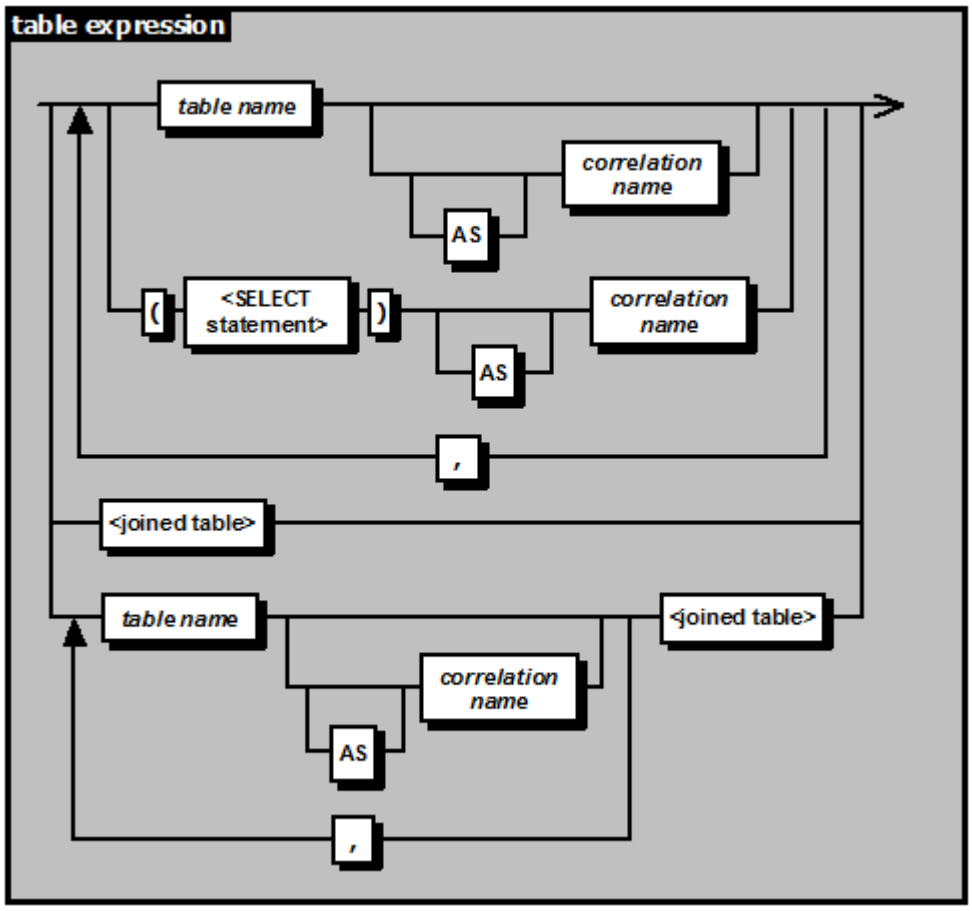


table name

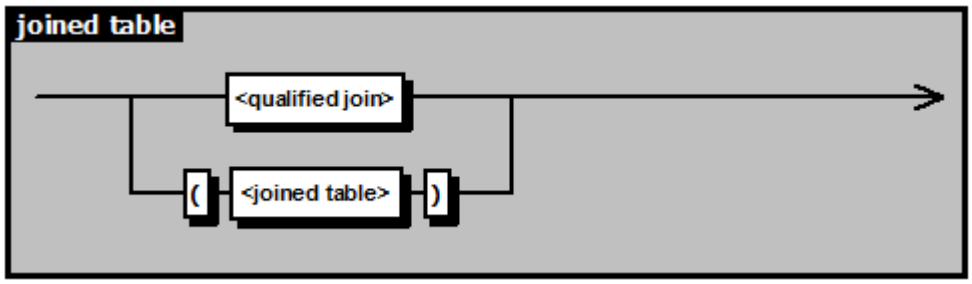
This identifies the archived table from which data is retrieved for the query results. Generally, the columns in the projection list are, or are based on, columns from this table.

correlation name

This is an alias for the archived table or subquery (nested table expression), used to identify the database object in qualified column references within the SELECT statement. A correlation name is optional for an archived table, since the actual name can be used to qualify column references. However, a correlation name is required for a subquery, which cannot be referenced in any other way.

Note that, in addition to requiring a correlation name, each subquery must be contained entirely in parentheses.

Joined Table Clause



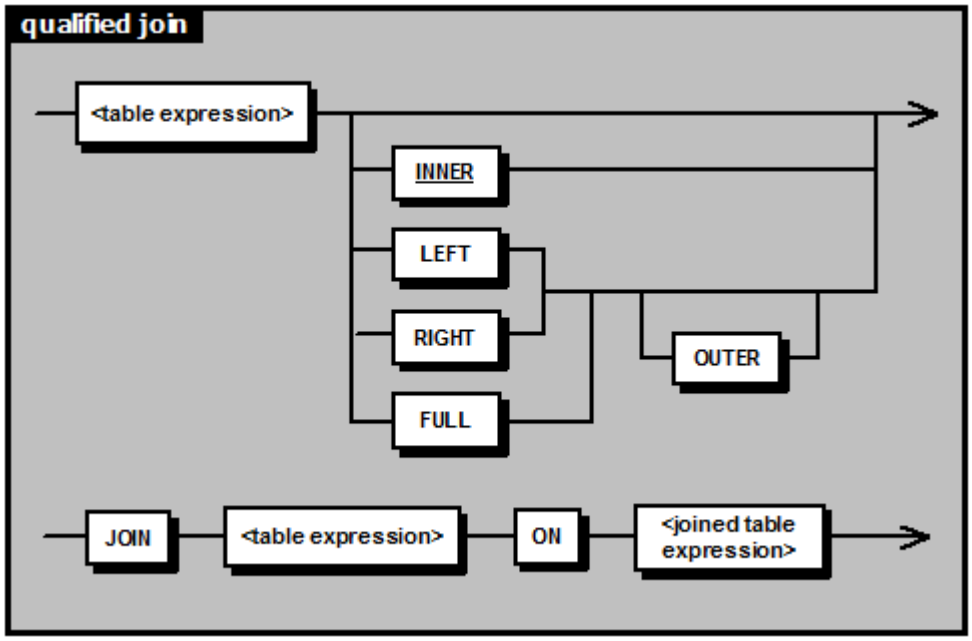
qualified join

A join may be performed on two or more tables or subqueries. Join types include INNER JOIN (the equivalent of simply specifying JOIN) and OUTER JOIN (LEFT, RIGHT, or FULL).

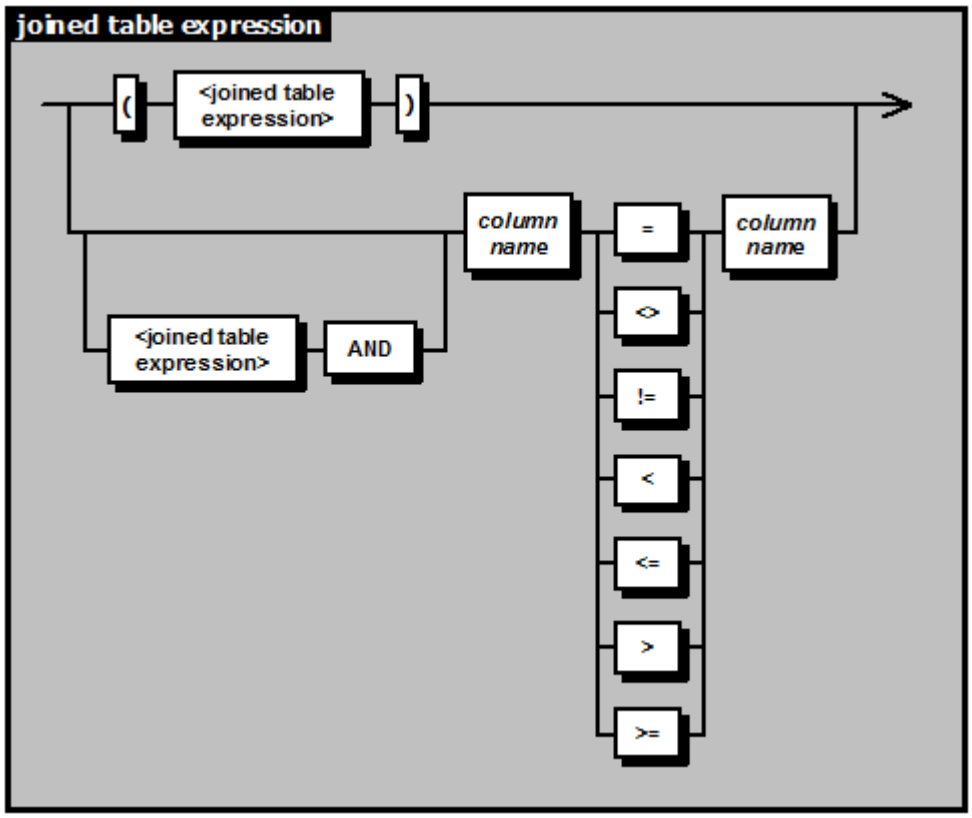
(*joined table*)

A table expression (table name or qualified join specification) can be included in parentheses to specify that it should be evaluated before other table expressions in the SELECT statement.

Qualified Join Clause



Joined Table Expression Clause



(joined table expression)

A joined table expression can optionally be enclosed by parentheses. The use of parentheses does not affect the order of processing in this case, since only the AND operator can be used to combine join predicates.

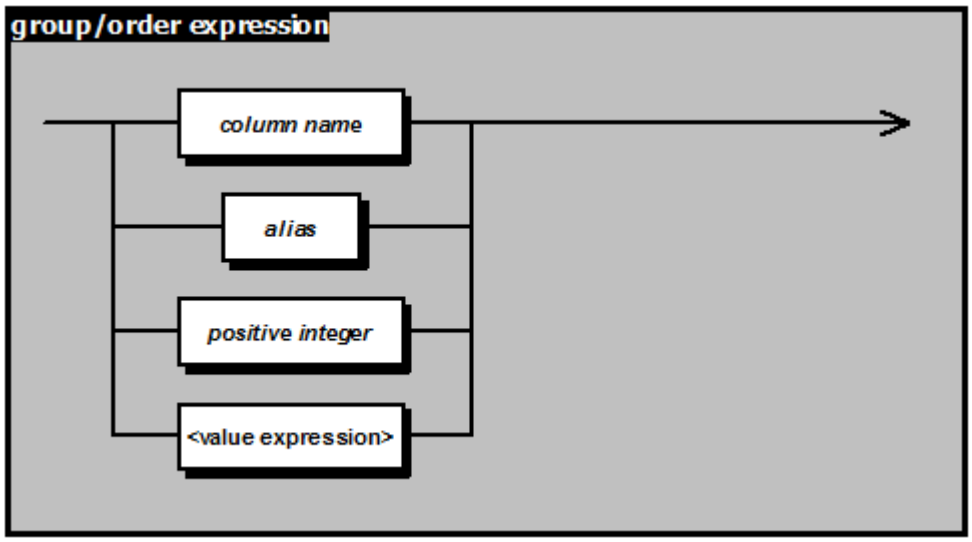
column name

The first *column name* argument is a reference to a column from one of the joined tables; the *column name* argument following the comparison operator must refer to a column from the other joined table. Comparisons to constants cannot appear in a join predicate.

AND

Multiple join predicates can be combined using the AND Boolean operator only. That is, OR conditions are not permitted in the joined table expression clause.

Group/Order Expression Clause



column name

The *column name* argument specifies a column in the projection list by which to group or order the results of the query.

alias

The *alias* argument specifies an alias given to a column or derived column in the projection list, which will be used to group or order the results of the query.

positive integer

The *positive integer* argument identifies a column or derived column by its position in the projection list. That is, the first item in the projection list is *1*, the second is *2*, and so on. The position number cannot refer to a constant, or an expression composed entirely of constants, in the projection list.

<value expression>

Grouping and ordering can be based on a value expression from the projection list by including the same expression in the GROUP BY or ORDER BY clause. The value expression in the GROUP BY or ORDER BY clause must match the one in the projection list exactly (excluding whitespace differences), otherwise an error will be returned.

SELECT Clauses

The following list provides SELECT clauses:

DISTINCT

The DISTINCT keyword dictates that duplicate rows be omitted from query results. If DISTINCT is specified, the projection list must consist only of named columns, with no functions or expressions. Note that this requirement also applies to all Column Rules defined on columns in the queried table.

AS

The AS keyword can be used to name result columns in a SELECT clause. This is particularly useful for a column derived from an expression or from a function applied to a column value.

The name of a result column is determined as follows:

- If the AS clause is specified, the name of the result column is the correlation name.
- If the AS clause is not specified and the result column is derived from a column name, the result column name is the unqualified name of that column.
- If the AS clause is not specified and the result column is derived from a function or value expression, the result column name is the number of the column, counting from left to right.

  Note that the AS keyword is optional when naming result columns.

FROM

This keyword introduces the archived tables or derived tables from which data is retrieved. The FROM clause *must* be included in a SELECT statement.

Correlation names can be used in place of the actual table name when qualifying columns within the query statement. The correlation name appears immediately after the corresponding table in the FROM clause (separated by whitespace) or after the keyword AS. Correlation names must satisfy the same naming conventions as table names (except that they cannot be delimited by quotation marks). The correlation name qualifies columns that appear in either SELECT or WHERE clauses.

Apart from individual archived table names, the FROM clause can include explicit join syntax.

Explicit Joins

You can use the following explicit join types:

- INNER JOIN

  Inner joins return matched rows from both tables.
- LEFT [OUTER] JOIN

  Left outer joins include non-matching rows only from the table named before (that is, to the left of) the LEFT OUTER JOIN clause. Missing values in a row are filled with nulls.
- RIGHT [OUTER] JOIN

  Right outer joins include non-matching rows only from the table named after (that is, to the right of) the RIGHT [OUTER] JOIN clause. Missing values in a row are filled with nulls.
- FULL [OUTER] JOIN

  The result of the full outer join contains all matched as well as unmatched rows from both tables.

An explicit join is allowed in the FROM clause with the following restrictions:

- a single explicit join tree can be specified (that is, multiple explicit joins, separated by commas, are not permitted in the query)
- the explicit join must appear after all implicit joins (individual archived table names, separated by commas) in the FROM clause, if implicit joins are included in the query.

The FROM clause must therefore comprise **one** of the following:

- one or more comma-delimited table names
- a single explicit join tree
- one or more comma-delimited table names, followed by a single explicit join tree

```
For example, the following query constructions are syntactically legal:
SELECT * FROM t1, t2, t3 WHERE ... ;
SELECT * FROM t4 INNER JOIN t1
ON t4.c1 = t1.c1 AND
t4.c2 != t1.c2 AND
t1.c3 > t4.c3
```

```
WHERE ... ;
SELECT * FROM t1 JOIN t2 ON t1.c1 < t2.c1
FULL OUTER JOIN t3 ON t2.c2 != t3.c2
WHERE ... ;
SELECT * FROM t1, t2, t7,
(t3 INNER JOIN t4 ON t3.c2 = t4.c2) FULL OUTER JOIN
(t5 INNER JOIN t6 ON t5.col1 != t6.c1) ON t3.c3 < t6.c3
WHERE ... ;
```

while the following are syntactically illegal queries:

```
-- implicit joins (t3, t4) follow explicit join:
SELECT * FROM t1 INNER JOIN t2 ON t1.c1 > t2.c1, t3, t4 WHERE ... ;
-- multiple join trees:
SELECT * FROM t1 INNER JOIN t2 ON t1.c1 > t2.c1,
t3 RIGHT JOIN t4 ON t3.c3 != t4.c3 WHERE ... ;
```

WHERE

The optional WHERE clause sets conditions that each record must meet before being retrieved by a query. If the SELECT statement does not include a WHERE clause, every row in the table being queried will appear in the result.

In the WHERE clause, a *Boolean value expression* defines a selection criterion (or criteria) applied to each row of data values in the table being queried by the SELECT statement. If the data in a particular record matches the selection criteria, then that record is retrieved by the query; otherwise the record is passed over.

Consult the next section for more information about the possible Boolean value expressions.

GROUP BY

The optional GROUP BY clause allows query results from aggregate functions on columns to be grouped in terms of the distinct values appearing in specified columns. The GROUP BY clause collects the rows that meet the specified search criteria into groups containing common values in the specified columns. There must be at least one expression in the SELECT column list that represents a group sharing a single value.

The grouping-column argument(s) must consist of a column name (or names) from among those specified in the column list following the SELECT command keyword. Multiple grouping-column arguments may be specified, separated by commas. A direct column reference need not correspond to a column in the projection list; the query output can be grouped by any column from the archived table.

An alternative to specifying a column by name is to use a *positive integer* to identify the column by its position in the projection list of the SELECT statement. That is, the first item in the projection list is *1*, the second is *2*, and so on. One restriction, however, is that a position number cannot refer to a constant, or an expression composed entirely of constants, in the projection list.

Additionally, grouping can be done by specifying a correlation name from the projection list, which is useful for grouping by a named value expression. For instance, if the SELECT list contains the item `"col1*col2 AS derived_col"`, the GROUP BY clause would reference `derived_col` to group by that derived column.

Grouping can also be based on a value expression from the projection list by including the same expression in the GROUP BY clause. The value expression in the GROUP BY clause must match the one in the projection list exactly (excluding whitespace differences and extraneous parentheses), otherwise an error will be returned.

A HAVING clause can be applied to qualify further the results of a GROUP BY clause.

HAVING

The HAVING clause further qualifies groups in the same way that the WHERE clause qualifies rows. Groups of rows from the tables referenced in the FROM clause that satisfy all the selection criteria arguments are used in composing the query result. Note that a GROUP BY clause must appear in the SELECT statement if there is a HAVING clause.

A HAVING clause may contain Boolean value expressions, simply or compounded together using AND or OR operators. Each Boolean predicate value expression must consist of a grouped column (that is, a column referenced in the GROUP BY clause), an aggregate function, a constant, or some arithmetic combination of grouped columns, aggregate functions, and constants. To optimize query execution speeds, avoid using compound expressions in the HAVING clause and in the projection list; it is best to specify columns only.

Subqueries may not appear in the HAVING clause.

WITH SAMPLE OF

The WITH SAMPLE OF clause returns a random sampling of the result set produced by the SELECT statement. That is, if the same SELECT...WITH SAMPLE OF statement is issued multiple times, a different subset of the total result set is returned to the caller each time.

The number of rows in the returned sample is proportional to the total number of rows in the result set. The "*number* IN *factor*" part of the clause indicates the proportion of returned rows to total rows, which is calculated as *number*:*factor*. For example, the following SELECT statement returns approximately a 10% subset, (a 1:10 proportion) of the entire result set:

```
SELECT p_no FROM products WITH SAMPLE OF 1 IN 10;
```

If 100 rows were contained in the result set, then about 10 of those rows, determined randomly, would be returned to the caller.

The syntax rules for *number* and *factor* are as follows:

- *number* and *factor* must be integers greater than zero.
- *number* must be less than or equal to *factor*.

If *number* is equal to *factor*, then no sampling is performed: the entire result set is returned. If more than zero rows are selected but the sample size is calculated as zero, then one row will be randomly chosen. The sampling logic will never return zero rows, unless the result set itself contains zero rows.

ORDER BY

The optional ORDER BY clause sorts the rows returned by the query according to the values in the specified columns, in the order in which they are specified. That is, results are sorted according to the values in the first column argument first; the values in the second column argument next, and so on. The column argument can be a column from among those returned by the query (that is, one of the select list expressions), or it can be any column from the archived table (regardless of whether the column is included in the select list). Multiple column arguments may be specified, separated by commas. Results can be sorted in ascending order (the default) or descending order by specifying the ASC or DESC keywords, respectively.

As an alternative to specifying a column by name, a *positive integer* may be used to identify the column by its position in the projection list of the SELECT statement. That is, the first item in the projection list is *1*, the second is *2*, and so on. In the case of SELECT *, the positive integer refers to the ordinal position of a column in the archived table.

Ordering can also be specified using a correlation name from the projection list, which is useful for ordering results by derived column.

Another way to sort by a value expression from the projection list is to include the same expression in the ORDER BY clause. The value expression in the ORDER BY clause must match the one in the projection list exactly (excluding whitespace differences and extraneous parentheses), otherwise an error will be returned.

EXPORT INTO

The "Direct Export" query option is a method of exporting data from an archived table to one or more flat files. By including the EXPORT INTO clause at the end of a standard SELECT statement, the data fetched by the query will be written to the specified files rather than displayed at the console. The Direct Export method is highly scalable, so increasing the number of computers with active Data Vault Agent processes should decrease the total amount of time required for the export operation. The key principle is to maximize the number of Data Vault Agents writing data to separate flat files in parallel with minimal hardware contention.

As the SELECT...EXPORT INTO command is executing, the fetched data will not be displayed at the console. Instead, as each Data Vault data file is processed, the data file name will appear on the screen. By default, this output is buffered by the client, so the data file names will actually be written in clusters sporadically during the export operation. To have the data file names displayed as they are processed, issue the following command prior to the SELECT...EXPORT INTO command:

```
ALTER SESSION SET FETCHROWS=1;
```

This does not affect the export operation performance in any way, only the presentation of information on the client side. (To restore the default data display buffering, execute the same command with FETCHROWS=0.)

Note that if the export operation fails before completion, the flat files written up to that point will most likely be incomplete.

Example of EXPORT INTO

```
SELECT id, ddate, price, qty, sku FROM ssa.demossa WHERE ddate > '2004-01-01'
  EXPORT INTO
    PATHS='\\ALPHA1\sct ; \\BETA1\work 7\sct
           \\DELTA1\usr\tmp'
    NOF=3
    FP='sct_export1'
    ESC='\x5c'
    CD='\x7c'
    RD='\x0a'
    NULL='\x87' ;
```

The above SELECT statement fetches all records from the DemoSSA table where the ddate value is after January 1, 2004, and exports this data to four flat files that will be created on the three specified computers (ALPHA1, BETA1, DELTA1). The flat files will have the names "sct_export1.000", "sct_export1.001", and "sct_export1.002". Columns in the flat files will be delimited by the pipe ("|") symbol; rows will be delimited by the linefeed character; null values will be indicated by the "double dagger" symbol ("‡"); and the escape character will be a backslash ("\").

The equivalent SELECT statement with the longer parameter aliases is as follows:

```
SELECT id, ddate, price, qty, sku FROM ssa.demossa WHERE ddate > '2004-01-01'
  EXPORT INTO
    PATHS='\\ALPHA1\sct ; \\BETA1\work 7\sct
           \\DELTA1\usr\tmp'
    NUMBEROFFILES=3
    FILEPREFIX='sct_export1'
    ESCAPE='\x5c'
    COLUMNDELIMITER='\x7c'
    ROWDELIMITER='\x0a'
    NULL='\x87' ;
```

CHAPTER 2

Date and Time Arithmetic

This chapter includes the following topics:

Date and Time Arithmetic Overview

Basic arithmetic operations can be performed with DATE, TIME, and TIMESTAMP values. Subtracting one date/time value from another produces an *interval*. An interval can also be produced by labeling an expression as a duration (a *labeled duration*). These intervals can be added to or subtracted from the date/time datatypes and other intervals, and can also be multiplied or divided by numeric expressions. Further, intervals of the same type can be grouped together using a subset of the aggregate functions, or compared in Boolean value expressions using comparison operators. There is implicit type casting when quoted string literals or numeric expressions are involved in date/time arithmetic or Boolean expressions.

Intervals

Subtracting one temporal type from another of the same type results in an *interval*. There are three types of intervals: date, time, and timestamp. A date interval is characterized by an integral number of *days*, ranging between -99,999,999 and 99,999,999 inclusive. A time interval is characterized by an integral number of *seconds*, ranging between -999,999 and 999,999 inclusive. And a timestamp interval is represented by a decimal number of *seconds*, anywhere between +/-99,999,999,999,999.999999 inclusive. Note that a negative interval is produced if the value being subtracted is greater than the value from which it is being subtracted. "Greater than", in the case of temporal intervals, means "more recent".

Intervals can be involved in arithmetic expressions involving other intervals or labeled durations (see below), date/time datatypes, or numeric expressions. If required, a date/time interval can be converted to an integer value using the INTEGER casting function.

The following table summarizes characteristics of these intervals:

Operation	Interval Type	Size	Unit
<DATE> – <DATE>	date interval	DEC (8,0)	days
<TIME> – <TIME>	time interval	DEC (6,0)	seconds
<TIMESTAMP> – <TIMESTAMP>	timestamp interval	DEC (20,6)	seconds

Examples of Intervals

The following table provides expressions for intervals:

Expression	Result
'01.01.1954'-'01.01.1953'	365 (days)
'1602-12-31'-'2001-01-01'	-145368 (days)
'00:00:00'-'10:10:10'	-36610 (seconds)
'23.59.59'-'22.59.00'	3659 (seconds)
'1954-01-01-00.00.00'- '1953-01-01-00.00.00'	31536000.000000 (seconds)
'2003-12-16-09.51.16.073000'- '2003-12-19-09.00.00.000000'	-256123.927000 (seconds)

Labeled Durations

Expressions can be cast into an interval type by labeling the expression as a duration. This *labeled duration* is represented by a numeric expression followed by one of the duration keywords. The supported labeled durations are DAY(S), MONTH(S), YEAR(S), HOUR(S), MINUTE(S), SECOND(S), and MICROSECOND(S).

The following table provides the possible labeled durations and resulting interval types:

Label	Result Type
<integer expression> DAYS (or DAY)	date interval
<integer expression> MONTHS (or MONTH)	date interval
<integer expression> YEARS (or YEAR)	date interval
<integer expression> HOURS (or HOUR)	time interval
<integer expression> MINUTES (or MINUTE)	time interval

Label	Result Type
<integer expression> SECONDS (or SECOND)	time interval
<decimal/float expression> SECONDS (or SECOND)	timestamp interval
<numeric expression> MICROSECONDS (or MICROSECOND)	timestamp interval

Examples

```
277 DAYS
(1024*1024-100) MINUTES
10e2 SECONDS
```

Intervals and Date/Time Values

Intervals can be added to or subtracted from date/time datatypes. Such arithmetic expressions will always evaluate to the same datatype as the date/time value in the expression. The base arithmetic expressions and their result types are summarized in the following tables:

DATE

The following table provides the DATE expressions:

Expression	Result Type
<DATE> – <date interval>	DATE
<DATE> + <date interval>	DATE
<date interval> + <DATE>	DATE

TIME

The following table provides the TIME expressions:

Expression	Result Type
<TIME> – <time interval>	TIME
<TIME> + <time interval>	TIME
<time interval> + <TIME>	TIME

TIMESTAMP

The following table provides the TIMESTAMP expressions:

Expression	Result Type
<TIMESTAMP> – <timestamp interval>	TIMESTAMP
<TIMESTAMP> + <timestamp interval>	TIMESTAMP
<timestamp interval> + <TIMESTAMP>	TIMESTAMP
<TIMESTAMP> – <date interval>	TIMESTAMP
<TIMESTAMP> + <date interval>	TIMESTAMP
<date interval> + <TIMESTAMP>	TIMESTAMP
<TIMESTAMP> – <time interval>	TIMESTAMP
<TIMESTAMP> + <time interval>	TIMESTAMP
<time interval> + <TIMESTAMP>	TIMESTAMP

Examples

The following table provides examples for expressions:

Expression	Result
13 DAYS + '2003-12-19'	2004-01-01
'11:30:45' - 156 SECONDS	11:28:09
'2003-12-19-12.35.52.544000' - 75 HOURS	2003-12-16-09.35.52.544000

Interval Arithmetic

Intervals can be added to and subtracted from other intervals in most cases.

The following table shows the possible combinations of addition and subtraction between two intervals, and the resulting interval types:

+ | –	DAYS	MONTHS	YEARS	HOURS	MINUTES	SECONDS	MICRO-SECONDS
DAYS	DAYS	*	*	SECONDS	SECONDS	SECONDS	SECONDS
MONTHS	*	MONTHS	MONTHS	*	*	*	*
YEARS	*	MONTHS	MONTHS	*	*	*	*

+ | –	DAYS	MONTHS	YEARS	HOURS	MINUTES	SECONDS	MICRO-SECONDS
HOURS	SECONDS	*	*	SECONDS	SECONDS	SECONDS	SECONDS
MINUTES	SECONDS	*	*	SECONDS	SECONDS	SECONDS	SECONDS
SECONDS	SECONDS	*	*	SECONDS	SECONDS	SECONDS	SECONDS
MICRO-SECONDS	SECONDS	*	*	SECONDS	SECONDS	SECONDS	SECONDS

Note: * You cannot perform an arithmetic operation between the two interval types.

Intervals can also be involved in addition, subtraction, multiplication, and division operations with numeric expressions as operands. The interval can appear on either side of an operator (**+**, **–**, *****, **/**) in all cases except one: a numeric expression cannot be divided by an interval. For example, "`12/(3 days)`" is an invalid arithmetic operation.

The following table lists the interval type resulting from the arithmetic interaction of a numeric expression and an interval of a given type (the result is the same for each operator):

Interval	Result Type
DAYS	DAYS
MONTHS	MONTHS
YEARS	MONTHS
HOURS	SECONDS
MINUTES	SECONDS
SECONDS	SECONDS
MICROSECONDS	SECONDS

Examples of Interval Arithmetic

The following table provides examples for interval arithmetic:

Expression	Result
1 DAY + 1 HOUR	90000.000000 (seconds)
60 SECONDS + 500 MICROSECONDS	60.000500 (seconds)
12 MONTHS + 1 YEAR	24 (months)
1 YEAR * 2	24 (months)
28 DAYS / 7	4 (days)

Expression	Result
8 * 2 HOURS	57600 (seconds)
20 MICROSECONDS + 2	2.000020 (seconds)
1 DAY - (30 + 30 MINUTES)	84570.000000 (seconds)

Interval Aggregation

Intervals of the same type can be grouped together using a subset of the aggregation functions. When intervals are aggregated in this manner, the resulting value is the same interval type.

The following aggregate functions can be used to group interval values:

- MIN
- MAX
- AVG
- SUM

Examples

given SMALLINT column *c1*:

c1

10

90

55

33

48

- MIN (C1 * 10 DAYS) = 100 (days)
- MAX (C1 - 5 SECONDS) = 85 (seconds)
- AVG (C1 + 10 HOURS) = 36047 (seconds)
- SUM (C1 - 1 MINUTE) = -64 (seconds)

Interval Comparisons

Intervals can be compared to intervals of the same type using any of the following comparison operators: **=**, **<**, **>**, **>=**, **<=**, **<>**. When such a comparison is made, the expression evaluates to a Boolean value (TRUE or FALSE).

Examples

```
x=3
y=1
```

Expression	Result
x SECONDS <= y SECONDS	FALSE
x HOURS <> y HOURS	TRUE
y DAYS < x DAYS	TRUE

Implicit Casting

Certain implicit type casting is performed on quoted string literals to DATE, TIME, and TIMESTAMP values, when they are part of an arithmetic expression or comparison operation. There is also implicit type casting on some numeric expressions to interval types in the same situations.

The following list provides the implicit date/time type castings:

- A quoted literal in a subtraction expression with a DATE value, or a quoted literal compared to a DATE value, is cast as a DATE value.
- A quoted literal in a subtraction expression with a TIME value, or a quoted literal compared to a TIME value, is cast as a TIME value.
- A quoted literal in a subtraction expression with a TIMESTAMP value, or a quoted literal compared to a TIMESTAMP value, is cast as a TIMESTAMP value.
- A numeric value in an addition or subtraction expression with a time or timestamp interval, or a numeric value compared to a time or timestamp interval, is cast as a SECONDS labeled duration.
- A numeric value in an addition or subtraction expression with a date interval, or a numeric value compared to a date interval, is cast as a DAYS labeled duration.
- A quoted literal that is added to a date interval, or a quoted literal that has a date interval subtracted from it, is cast as a DATE value.
- A quoted literal that is added to a time interval, or a quoted literal that has a time interval subtracted from it, is cast as a TIME value.
- A quoted literal that is added to a timestamp interval, or a quoted literal that has a timestamp interval subtracted from it, is cast as a TIMESTAMP value.
- A numeric value that is added to a DATE value, or a numeric value subtracted from a DATE value, is cast as a DAYS labeled duration.
- A numeric value that is added to a TIME or TIMESTAMP value, or a numeric value subtracted from a TIME or TIMESTAMP value, is cast as a SECONDS labeled duration.
- A numeric value subtracted from a quoted literal, or a numeric value added to a quoted literal, involves a double casting: if the quoted literal is a valid date, it is cast as a DATE value and the numeric is cast as a DAYS labeled duration; if the quoted literal is a valid time, it is cast as a TIME value and the numeric is cast as a SECONDS labeled duration; or if the quoted literal is a valid timestamp, it is cast as a TIMESTAMP value and the numeric is cast as a SECONDS labeled duration

- A quoted literal subtracted from a quoted literal involves a double casting: if both quoted literals are valid dates, they are both cast as DATE values. If both quoted literals are valid times, they are both cast as TIME values; or if both quoted literals are valid timestamps, they are both cast as TIMESTAMP values.

The following table displays examples for implicit casting:

Expression	Result
'01:01:01' + 59	01:02:00
'1945-09-02' - '1939-09-10'	2184 (days)
'2003-12-19' - 19	2003-11-30

CHAPTER 3

WHERE Clauses

This chapter includes the following topics:

WHERE Clauses Overview

Selection criteria arguments are used in the WHERE clauses of SELECT statements. These selection criteria consist of a collection of predicates and expressions that determine which rows of the table will be returned by a SELECT statement.

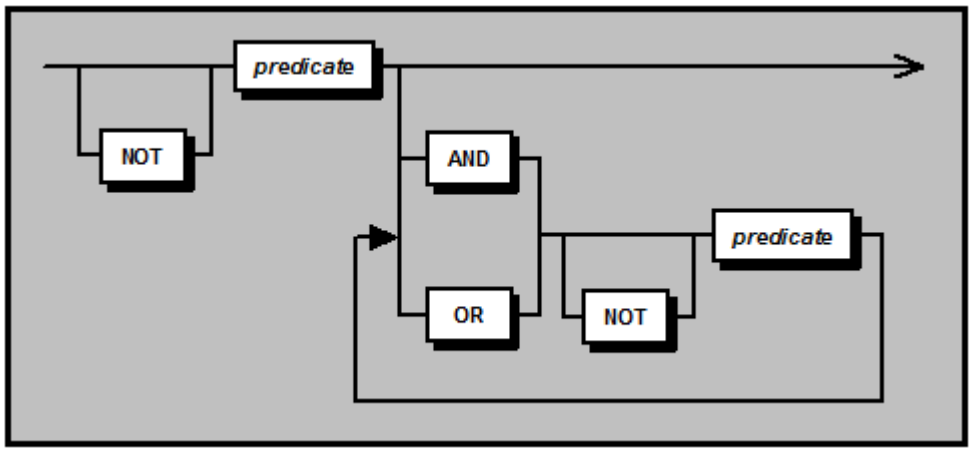
Boolean Value Expressions (Search Conditions)

Boolean value expressions are used in Data Vault Service SQL to specify a *search condition* that can be used in SELECT statements. These expressions use various *predicates* to test a particular operand (for example, the contents of a column) for membership in a specified set of values. If the operand satisfies the condition, it is TRUE; if it does not satisfy the condition, it is FALSE; a NULL value can force a Boolean expression to evaluate to UNKNOWN.

When the NOT operator is used, the following Boolean values are equivalent:

- NOT TRUE is FALSE
- NOT FALSE is TRUE
- NOT UNKNOWN is UNKNOWN

Syntax



Boolean value expressions can specify a single search condition or can combine multiple conditions linked by the AND / OR keywords. The order in which Boolean operators are evaluated within an expression is as follows, unless altered by the presence of parentheses: NOT, AND, then OR.

Note that parentheses `(...)` can be placed around individual predicates, groups of predicates, and even the whole Boolean value expression. If a predicate (or collection of predicates) is enclosed by parentheses, it will be evaluated first. If parenthetical expressions are nested, the innermost condition surrounded by parentheses is evaluated before all others. Placing parentheses around logical operations also enhances the readability of the SQL, especially if there are many predicates in the expression.

The following table provides evaluation of different Boolean expressions combined with the AND operator:

Boolean Expression	true	false	unknown
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

The following table provides evaluation of different Boolean expressions combined with the OR operator:

Boolean Expression	true	false	unknown
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

Predicates

WHERE clauses can contain simple or compound predicates.

Simple Predicates

A simple predicate uses comparison operators in selection conditions:

The following table lists the simple predicates:

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

The following query displays all rows in the Supplier table containing a value of 20 in their *status* column:

```
SELECT *
FROM supplier
WHERE status = 20;
```

The following table displays the query results:

sno	sname	status	city
S1	SMITH	20	LONDON
S4	CLARK	20	LONDON

The WHERE clause in the next example compares a character string 'PARIS' with the *city* column. The query below retrieves *sno* (supplier number) and *status* column data for all suppliers in Paris:

```
SELECT sno, status
FROM supplier
WHERE city = 'PARIS';
```

Note that character strings must be enclosed by single quotation marks ('*x*').

The following table displays the results:

sno	status
S2	10
S3	30

When one column value is compared against another column value in a query and the column datatypes are different, the Data Vault Service tries to convert one of the data values to a datatype compatible with the other data value before performing the comparison. For example, if integer and float column values are compared, the INT data value is converted to a FLOAT value before the Data Vault Service performs the comparison.

Note that BLOB columns cannot be compared with other BLOB columns, or with columns of any other datatype.

Spaces in data values are considered when evaluating retrieval conditions. Right-justified data with leading spaces does not equal left-justified data with trailing spaces. For example, the string 'Jones' is read as a completely different value, depending on the number of preceding or trailing blank spaces attached.

The LIKE predicate operator allows string pattern matching without strict adherence to character string content and order. See the **LIKE Predicates** section below for more information.

The **< >** (not equal) comparison operator specifies records for retrieval that do not meet the specified condition. For example, the following query retrieves the *pno* (part number) column values of all Part table records with a *color* column value other than 'RED'.

```
SELECT pno
FROM part
WHERE color <> 'RED';
```

Compound Predicates

Compound predicates use the operators OR and AND. For example, to display the *sno* column values for all suppliers in "PARIS" with a *status* column value greater than 20, use the following query:

```
SELECT sno
FROM supplier
WHERE city = 'PARIS'
AND status > 20;
```

This SELECT command statement retrieves *pno* column values for those parts that are either 'RED', or weigh more than 15 pounds:

```
SELECT pno
FROM part
WHERE color = 'RED'
OR weight > 15;
```

In complex queries, parentheses can be used to indicate the order of evaluation. The condition(s) surrounded by the innermost pair of parentheses are applied first.

```
SELECT pno, pname
FROM part
WHERE color = 'green'
OR (city = 'LONDON'
AND weight < 15);
```

This last query retrieves *pno* and *pname* column values of all parts that are either green, or are both made in London and have a weight less than 15.

The following table displays the query result:

pno	pname
P1	NUT
P2	BOLT
P4	SCREW

BETWEEN Predicates

The BETWEEN operator can evaluate whether or not data values fall within the range of values indicated in the predicate.

Each occurrence of the expression *x* is evaluated to determine if it sorts between and including the range values indicated by *y* and *z* (the range values in the BETWEEN clause are also part of the result set). The three variables above can consist of any combination of value expressions, although *x* is typically a column name.

Note that the value of *y* should be less than that of *z*. Otherwise, in the case of literals, an error will be returned; in the case of numeric expressions, the predicate will always return FALSE.

The following query selects all Part table records with *weight* column values between 15 and 18:

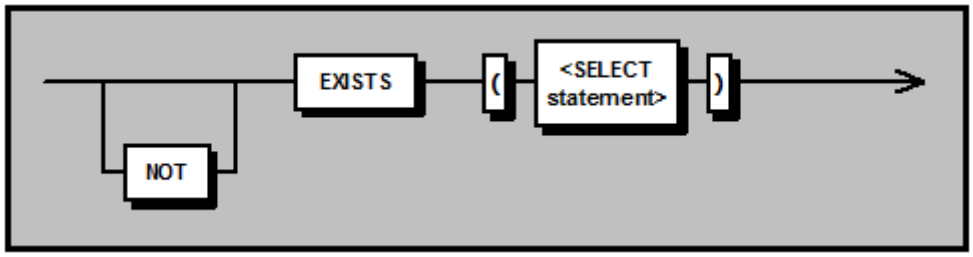
```
SELECT pno, pname
FROM part
WHERE weight BETWEEN 15 AND 18;
```

The following table displays the records retrieved:

pno	pname
P2	BOLT

EXISTS Predicate

The [NOT] EXISTS predicate tests a subquery for the presence of a value. If the subquery returns at least one value, the EXISTS predicate evaluates to TRUE, while a NOT EXISTS predicate evaluates to FALSE. Its syntax is as follows:



Note that a *correlated* subquery is supported with some restrictions:

- A correlated EXISTS predicate cannot appear in the HAVING clause.
- Correlation can be used in the WHERE and HAVING clauses of the subquery.
- Only one correlated column is allowed in the subquery, but it can be part of multiple conditions in the WHERE and HAVING clauses.

For instance, the following clauses are allowed in an EXISTS subquery ('outer' is a table reference from the outer query):

```
WHERE outer.c1 != inner.c1
WHERE outer.c1 > 15 AND inner.c2 > outer.c1
WHERE outer.c1 > 15 AND outer.c1*3 <= inner.c1 OR outer.c1 = 0
WHERE outer.c1 > 15 ... HAVING MAX(outer.c1) > 40
```

But the following are disallowed:

```
WHERE outer.c1 > 15 AND outer.c2 <= inner.c2
WHERE outer.c1 > 15 ... HAVING MAX(outer.c2) > 10
```

Example of EXISTS Predicate

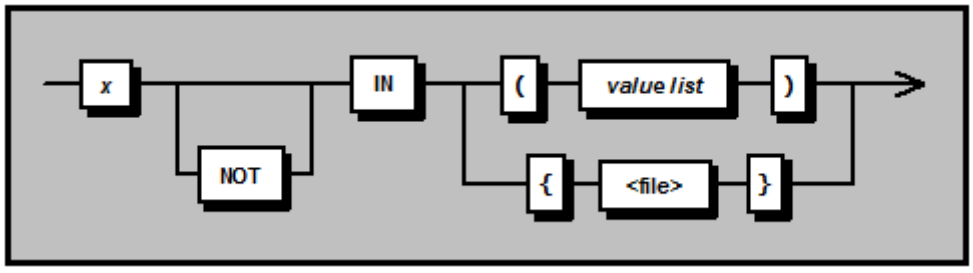
The following query returns the last and first names (*lname*, *fname*) of all personnel in the 'patients' table if the same patient ID (*pid*) record has the *infected* attribute set to 'TRUE' in the 'diagnosis' table:

```
SELECT P.lname, P.fname
FROM patients P
WHERE EXISTS (SELECT *
FROM diagnosis D
WHERE P.pid = D.pid
AND D.infected = 'TRUE');
```

IN Predicates

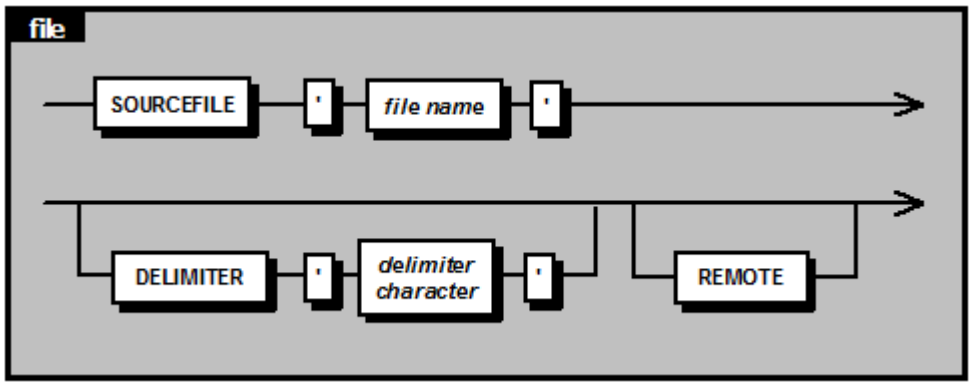
The IN and NOT IN operators cause the values appearing in the WHERE clause predicate to be tested for membership in a collection of values, which can be contained in a specified file, represented as a value list, or the result of a noncorrelated subquery.

The syntax for the IN predicate is as follows:



The set of values contained in the *value list* must be compatible with the datatype of *x* (which is typically a column name). The NULL keyword cannot be included in the value list (for comparisons against null, use a NULL predicate instead). If a SELECT statement (subquery) is used with the IN predicate, the subquery may

only return values from a single column. The individual values contained in the value list must be separated by commas.



When the alternative IN-file option is used, the parameters following the "IN" keyword must all be contained within a set of braces **{ }**. The one required element of the IN-file option is the path of a single file containing the list of values in the test set, enclosed by single quotation marks and preceded by the keyword "SOURCEFILE". If the specified file is missing, an error is returned and the whole query is cancelled.

One of the two optional elements of the IN-file clause is the delimiter character, which is a single ASCII character that will act as the list delimiter in the specified text file. This character must be contained in single quotation marks and preceded by the keyword "DELIMITER". By default, if the delimiter is not specified, it is assumed to be a newline (or carriage return plus newline) character: **\n** or **\r\n**. Hexadecimal characters are not supported at this time.

The other optional element in the IN-file clause is the keyword "REMOTE", which (when present) indicates that the specified file is on the Data Vault Service side. If this keyword is omitted, the file is assumed to be on the **client** side: that is, the machine from where the query originated.

Note that multiple files cannot be specified in a single IN-file clause. However, multiple IN-file clauses, each referencing a different file, can be included in the same query.

Also note that each file name specified by the IN-file clauses in the same query must be unique, even if they are in different locations. This is because all of the files will be copied to the same SHAREDIR folder for processing, and so the names must not conflict.

Examples of IN Predicates

The following is an example of an IN clause with a value list:

```
SELECT sno, sname
FROM supplier
WHERE sno IN ('S1', 'S2', 'S3');
```

All data values in the *sno* column must match one of the three string constants in order for their corresponding records to be retrieved.

This query retrieves the following data values:

sno	sname
S1	SMITH
S2	JONES
S3	BLAKE

The following is an example of IN-file usage:

```
SELECT col2
FROM sct1
WHERE col1 IN { SOURCEFILE 'C:\ssa\tmp\infile.csv' DELIMITER ',' }
OR col1 IN { SOURCEFILE '\\ALPHA\Public\in.txt' REMOTE } ;
```

In this example, an archived table named *sct1* is queried. Each value in column *col1* is compared against the set of values in the client-side file, *infile.csv*, where separate values are separated by commas. The values in the same column are also checked against the contents of a remote, server-side file named *in.txt* (on a machine named ALPHA), whose individual values are assumed to be delimited by newline, or newline plus carriage return, characters. The query returns the *col2* value for each record that satisfies the IN-file membership testing.

The final example shows an IN clause containing a noncorrelated subquery:

```
SELECT SUM(S.sales) AS Total_UK_Sales
FROM stores S
WHERE S.storeno IN (SELECT L.storeno
FROM locations L
WHERE L.country = 'UK');
```

Here, the outer query calculates the total sales of stores whose *storeno* value is contained in the result set of the inner query. The inner query (the noncorrelated subquery) returns only the *storeno* values of stores located in the UK. Hence, the value returned by the overall query is the sum of sales for UK stores only.

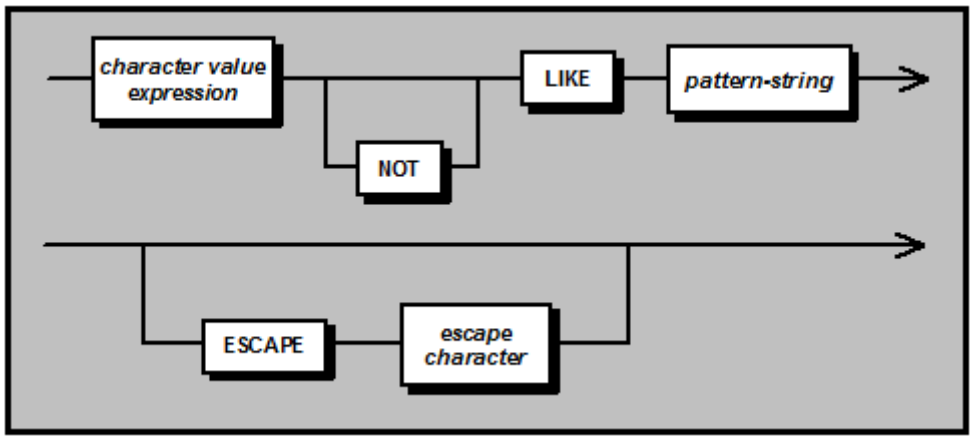
LIKE Predicates

The LIKE condition specifies a test that involves pattern matching. You can use the LIKE predicate with placeholders.

The following text is an example of a query that uses the LIKE predicate:

```
Select * from employee where first_name like “John%”
```

The following image shows the syntax for a LIKE (pattern-match) predicate:



character value expression

The *character value expression* parameter is compared to the pattern string. The expression can consist of a direct column reference, a string function, a string constant, or any combination of these using the concatenation operator (||).

pattern-string

All string-constant arguments must be enclosed by single quotation marks (' '). The underscore character (_) is a wildcard character that matches any single character; the percent character (%) is a wildcard character that matches any number (zero or more) characters. Note that wildcard characters can only appear at the end of the string.

ESCAPE

The optional ESCAPE clause designates a special escape character, which if placed immediately before a percent (%) or underline (_) character causes that percent or underline character to be interpreted literally as part of the pattern string.

escape character

An escape character must appear after the ESCAPE keyword in the statement, if the ESCAPE clause is used. The escape character can be any single character, enclosed by single quotation marks (' '). It is recommended that the chosen escape character not be a common string component (such as a letter or digit), or one of the wildcard characters (the percent or underline character), as the query expression might not produce the expected results.

Description

A particular row satisfies the LIKE predicate if the value of the column specified as the *character value expression* preceding the LIKE predicate matches the pattern specified in the *pattern-string* argument. Otherwise, the row is disqualified. The following examples demonstrate pattern matching:

The LIKE predicate is satisfied in the following cases:

- Part LIKE 'Q%'. All part values beginning with Q.
- Part LIKE '%Q'. All part values ending with Q.
- Part LIKE 'Q_ _ _9'. All part values beginning with Q and having a 9 as the fifth and final character.
- Part LIKE '_ _6%'. All part values having a 6 as the third character.
- Part LIKE '%QQQ%'. All part values containing the string QQQ (for example, QQQAB, AQQQB, and ABQQQ).

ESCAPE character

If either the percent (%) character or underscore (_) character is to be interpreted literally within the *pattern-string* argument of a LIKE predicate, an escape character must precede the percent character or underscore character within the string constant.

In order to define an escape character, the ESCAPE keyword and its character argument must follow the *pattern-string* argument within the LIKE predicate. The character specified for the character argument must precede occurrences of the percent character or underscore character within the *pattern-string* argument. Consider the following example:

```
desc LIKE '%40!%%' ESCAPE '!'
```

This LIKE predicate is satisfied by rows containing the string **40%** in the *desc* column.

LOOKUP Predicates

As part of an archived table query, the Lookup predicate allows the insertion of an array of values into placeholders in a template Boolean expression. Each row of the array represents a different instance of the same Boolean expression. When the Lookup predicate is processed, all of those individual expressions are combined by logical OR conjunctions into a larger conditional expression.

For example, the following inline Lookup instance:

```
Lookup( c1 = %1 AND c2 > %2 OR c3 < %3,
('N', 1, 100),
('Y', 2, 150),
('N', 3, 200)
)
```

is equivalent to the following conditional expression:

```
(c1 = 'N' AND c2 > 1 OR c3 < 100) OR
(c1 = 'Y' AND c2 > 2 OR c3 < 150) OR
(c1 = 'N' AND c2 > 3 OR c3 < 200)
```

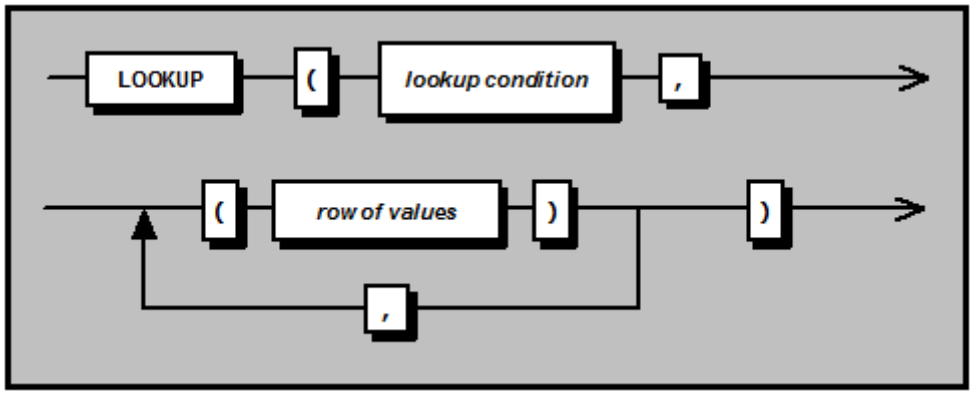
In this example, the three rows of lookup values are inserted into placeholders in the lookup expression when the query is executed, producing three distinct Boolean expressions, connected by ORs.

Inline Lookup

With the inline form of Lookup, all of the comparison values are specified as inline parameters within the Lookup instance in the SQL statement. By contrast, the CSV file version of Lookup specifies the location of an external file that contains all of the comparison values.

Other than the manner of specifying lookup values, there are no substantial differences between the two types of Lookup. However, inline Lookup might be preferred over CSV file Lookup if the array of values is not too large, or if it is desirable to avoid an external dependency in the form of a CSV file.

The syntax for an inline LOOKUP predicate appears below:



lookup condition

The conditional expression containing placeholders. Multiple sub-conditions can be connected by the logical AND or OR conjunctions. Parentheses can be used to group sub-conditions for readability and/or to override the logical order of processing.

The placeholders take the form of "%p", where p is an integer ranging from 1 to the total number of values in each row of the data array. The number of distinct placeholders can be less than the total number of values in each array row, which means that one or more columns of values in the array will be

ignored when testing the Lookup conditions. For instance, in the expression "col1 > %1 AND col3 = %3 AND col7 < %7", only the 1st, 3rd, and 7th values in each row of the array are used.

The placeholders do not have to appear in sequential order from right to left within the expression. For example, "col1 = %3 OR col2 = %2 OR col3 = %1" is a perfectly valid expression.

Also, the same array value can be used multiple times in the expression, simply by repeating the specific placeholder. For example, we could have this expression: "col1 = %1 AND col2 > %1". This means that the same value will be compared against both col1 and col2.

Note that a comma (,) must follow the *lookup condition* parameter.

row of values

A row of values in the "array" that will be inserted into the lookup expression. Each row must have the same number of values, separated by commas, and the datatype at the same position in each row must be compatible.

Currently, there is no way to specify null values. An empty string can be specified using two single quotation marks ('').

Each row of values must be contained in parentheses, and separated from the next row by a comma.

Any individual character value can be specified using the hexadecimal format. For example, the uppercase letter "Z" (ASCII hexadecimal code: 5A) can be represented as:

```
x'5A'
```

Similarly, the unprintable line feed control character can be specified as:

```
x'0A'
```

A full string can also be represented in hexadecimal form by concatenating the ASCII codes for all the characters in the string. For example, the string "Crüe" can be written as:

```
x'4372FC65'
```

String concatenation is supported for inline Lookup strings, using the concatenation (||) operator.This makes it possible to specify the hexadecimal form only for extended or unprintable characters, instead of for the whole string. For instance, the string "Crüe" can also be written as:

```
'Cr'||x'FC'||'e'
```

Note that literal single quotation marks (') can be referenced by escaping the quote; that is, specifying two consecutive single quotation marks (''). Alternatively, single quotation marks can be specified using the hexadecimal representation:

```
x'27'
```

The following two Lookup strings are therefore equivalent:

```
'''Drive,'', she said.'
x'27'||'Drive,'||x'27'||' she said.'
```

They both specify the following string:

```
'Drive,' she said.
```

CSV File Lookup

The CSV file form of the Lookup predicate generally works the same way as the inline form, except the array of values is contained in an external file referenced by the Lookup instance. The values in the external file are typically in the CSV (Comma-Separated Values) format, although Lookup can also read some custom formats.

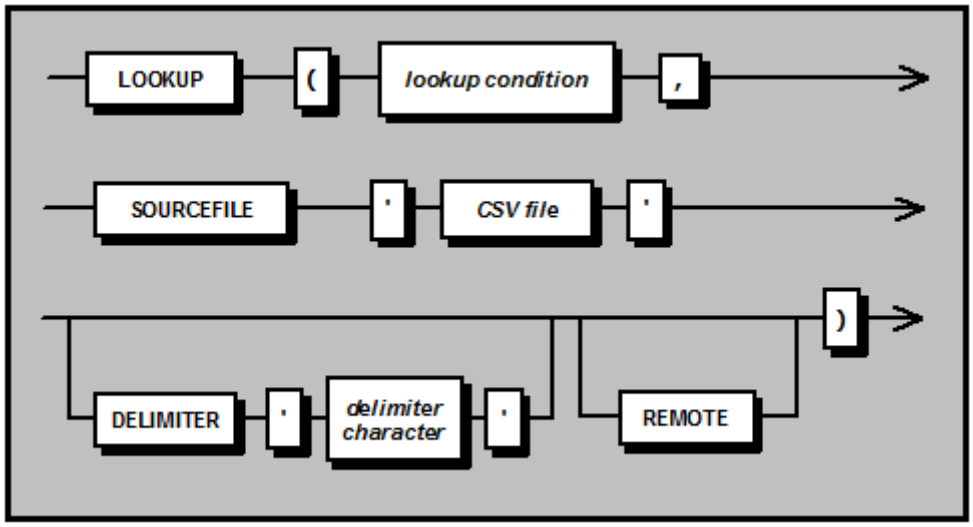
The Lookup predicate adheres to the following CSV file characteristics:

- Each row of values is located on a separate line, delimited by a line break.
- The last record in the file may or may not have an ending line break.
- Within each record, there may be one or more fields, separated by commas, and each line contains the same number of fields throughout the file.
- Each field may or may not be enclosed in double quotation marks ("). If fields are not enclosed by double quotation marks, then double quotation marks may not appear inside the fields.
- Fields that contain line breaks, double quotes, and commas should be enclosed by double quotation marks.
- If double quotation marks are used to enclose fields, then a double quotation mark appearing inside a field must be escaped by preceding it with another double quotation mark.

One CSV file feature not supported by Lookup is the optional header line. A CSV file referenced by Lookup must not contain the field names on the first line.

Note that there are no limitations on the size of a CSV file, except those imposed by hardware resources or the operating system.

The syntax for a CSV file LOOKUP predicate appears below:



lookup condition

The conditional expression containing placeholders for the CSV values. Note that a comma (,) must follow the *lookup condition* parameter.

The following table describes the lookup condition parameters.

Parameter	Description
SOURCEFILE '*CSV file*'	Specifies the CSV file containing an array of values that will be used in the lookup condition. The CSV file path, which must be enclosed by single quotation marks, can be absolute or relative.
DELIMITER '*delimiter character*'	Optionally specifies which single character separates the values in the CSV file. By default, the *delimiter character* is a comma (**,**), as is standard in CSV files. Note that the rows in the CSV file are delimited by line breaks, and this row delimiter cannot be changed.
REMOTE	Indicates that the specified CSV file is on the Data Vault Service side. If this keyword is omitted, the file is assumed to be on the client side (the machine where the query originated).

While the Lookup predicate is capable of processing a typical CSV file, it is flexible enough to also handle nonstandard CSV files or non-CSV files. As mentioned above, the DELIMITER parameter in the Lookup syntax allows the specification of any other single character as the value delimiter in the file. For example, the following CSV rows use the default comma, pound symbol (**#**), pipe symbol (**|**), and space character, respectively, as value delimiters:

```
1,"aubergine",-6e23,"Y"
1#"aubergine"#-6e23#"Y"
1|"aubergine"|-6e23|"Y"
1 "aubergine" -6e23 "Y"
```

As with the inline form of Lookup, columns in the array of values in the CSV file can be ignored by omitting the associated placeholders in the conditional expression.

Unlike the inline form of Lookup, hexadecimal values cannot be specified in the CSV file. However, this should not be a problem, since any literal character can appear in the CSV file. The CSV file can even include control characters like the escape character (ASCII code 27), or special whitespace characters like the vertical tab (ASCII code 11), as long as the containing string is enclosed by double quotation marks.

Line breaks that are part of a string appear as line breaks in the CSV file, in exactly the same way that it would appear if the string were printed out. For instance:

```
24786,"Dear Joan,
Get well soon.
Regards,
Tim",1997
24787,"shipping notice",2004
```

In the above example, the string "Dear Joan,<line break><line break>Get well soon.<line break><line break>Regards,<line break>Tim" is interpreted as a single string in the same record as the values "24786" and "1997" that appear on either side of the string. The row that follows ('24787,"shipping notice",2004') is a more typical record.

Optionally, any field value can be enclosed by double quotation marks in the CSV file, whether the value is string or numeric. However, certain characters require that the whole string be contained in quotes:

- control characters (ASCII codes 1-31 and 127)
- space character (ASCII code 32)
- comma (**,**)

- double quotation marks ("), if they are at the start of the string

  Note that if control characters and spaces are not part of a quoted string, they are ignored or treated as delimiters.

Repeated delimiter characters (for example, ",,,,,") are counted as a single delimiter.

Literal double quotation marks in a CSV file string must be escaped by doubling the character (""). For example:

```
24788,"She said, ""OUCH!""",1998
```

The string in the above example is interpreted as:

```
She said, "OUCH!"
```

As with the inline form of Lookup, null values cannot be specified. An empty string can be specified using "".

Warning: If the null control character (ASCII code 00) is present in the CSV file, either as part of a string literal or as a delimiter, the query will hang when executed against the Data Vault Service.

Inline Lookup Example

```
SELECT c1, c2, c3
FROM S1.T1
WHERE
Lookup ( c1 <= %1 AND c2 = %2 AND c3 >= %3,
( '200712220721320215402412 63', 158.97, '20071223091422020670233655'),
( '20071218163352021130070420', 12.26 , '20071218163352021130070420'),
( '20071217125938020250714927', 22.20 , '20071217125938020250714927'),
( '20071222085214020220173289', 148.91, '20071222085214020220173289'),
( '20071217121333021600203214', 56.48 , '20071223091422020670233670')
) ;
```

CSV File Lookup Example

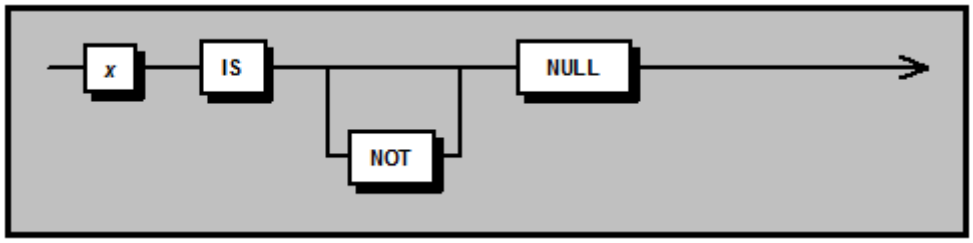
```
SELECT c1, c2, c3
FROM S1.T1
WHERE
Lookup ( c1 <= %1 AND c2 = %2 AND c3 >= %3,
SOURCEFILE 'C:/prod/csv/20090116.csv' DELIMITER ','
) ;
       where the CSV file 20090116.csv contains the following data:
"200712220721320215402412 63",158.97,"20071223091422020670233655"
"20071218163352021130070420",12.26,"20071218163352021130070420"
"20071217125938020250714927",22.20,"20071217125938020250714927"
"20071222085214020220173289",148.91,"20071222085214020220173289"
"20071217121333021600203214",56.48,"20071223091422020670233670"
```

Both of the Lookup examples above are effectively identical to the following SQL statement:

```
SELECT c1, c2, c3
FROM S1.T1
WHERE
(c1 <= '200712220721320215402412 63' AND c2 = 158.97 AND
c3 >= '20071223091422020670233655') OR
(c1 <= '20071218163352021130070420' AND c2 =  12.26 AND
c3 >= '20071218163352021130070420') OR
(c1 <= '20071217125938020250714927' AND c2 =  22.20 AND
c3 >= '20071217125938020250714927') OR
(c1 <= '20071222085214020220173289' AND c2 = 148.91 AND
c3 >= '20071222085214020220173289') OR
(c1 <= '20071217121333021600203214' AND c2 =  56.48 AND
c3 >= '20071223091422020670233670') ;
```

NULL Predicates

The IS [NOT] NULL operator can be used in WHERE clause predicates to match or exclude null values occurring in a specified data column. Its syntax is as follows (where *x* is typically a column name):



NULL Predicates Example

The following query retrieves all Part table records that do not have a null value in their *color* data column:

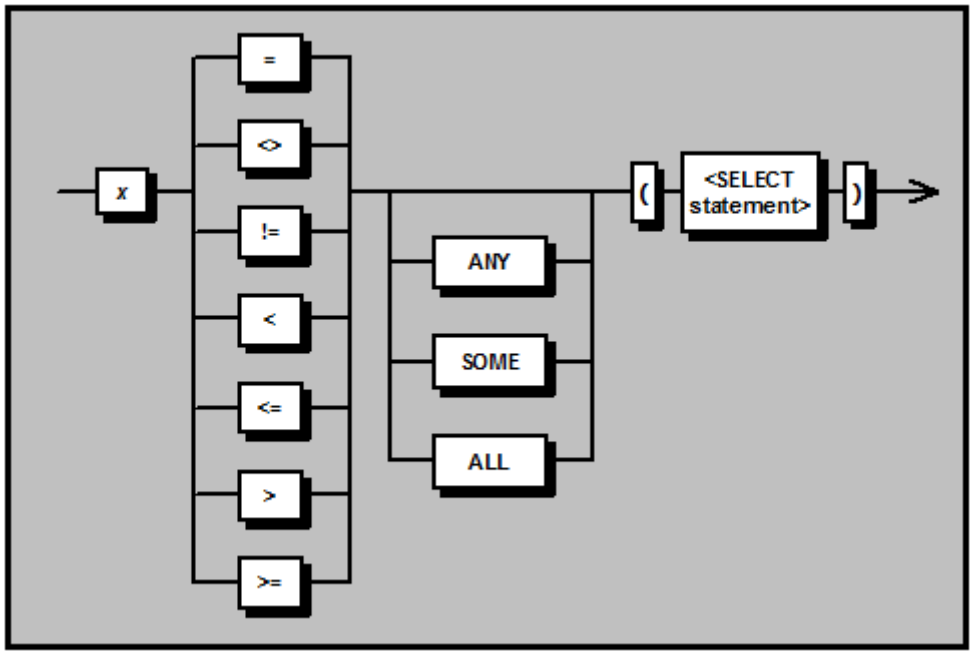
```
SELECT *
FROM part
WHERE color IS NOT NULL;
```

When used in a predicate, the NULL operator keyword must be preceded with either the IS or IS NOT keywords. Operators such as = or <> cannot be used with the NULL keyword.

Quantified Comparisons

The quantified comparison predicate compares the specified value (*x*) with the values returned by a subquery. If ALL is specified, then the value is tested for its appearance in all records returned by the subquery. If either ANY or SOME is specified (these two keywords are equivalent), the value is tested for its appearance in at least one of the records returned by the subquery. If no keyword is supplied, the subquery is interpreted as a *scalar* subquery, which must return exactly one row and one value.

If the value *x* is null, or the subquery returns no rows, the predicate will evaluate to UNKNOWN. The SELECT statement (subquery) may only return values from a single column, or in the case of a scalar subquery, only a single value.



Note that a *correlated* subquery is not permitted at this time.

Example of Quantified Comparison

```
SELECT * FROM inventory
WHERE cost >= ALL (SELECT I.price
FROM items I, orders O
WHERE I.o_no = O.o_no);
```

Each *cost* value in the inventory table is tested against all the *price* values returned by the subquery. Since the operator/keyword is **`>= ALL`** , if a particular row's cost value is greater than every single value returned by the subquery, the value expression returns TRUE, and that row is included in the result set.

CHAPTER 4

UNION Operator

This chapter includes the following topics:

UNION Operator Overview

The UNION operator combines the result sets of two or more SELECT statements. By default, the Data Vault Service evaluates the SELECT statements from left to right. You can use parentheses to explicitly set the order of evaluation.

The UNION operator has the following forms:

- UNION
- UNION ALL

UNION

The UNION operator combines the result sets of multiple SELECT statements but does not include duplicate rows in the final result set. The final result set contains only distinct rows from all the SELECT statements.

Syntax

The UNION operator has the following syntax:

```
SELECT column1 [, columnN ]
FROM table1 [, tableN ]
[WHERE clause]

UNION

SELECT column1 [, columnN ]
FROM table1 [, tableN ]
[WHERE clause]
```

Examples

The following UNION query selects the same number of columns from different tables:

```
SELECT location_id, department_name,
   FROM departments
   UNION
   SELECT location_id, warehouse_name
   FROM warehouses;
```

The following UNION query selects all columns from multiple tables that have the same number of columns of the same datatype:

```
SELECT *
   FROM departments
   UNION
   SELECT *
   FROM warehouses
   UNION
   SELECT *
   FROM offices;
```

The following UNION query includes an ORDER BY clause:

```
SELECT City
FROM Customers
UNION
SELECT City
FROM Suppliers
ORDER BY City;
```

UNION ALL

The UNION ALL query combines the result sets of multiple SELECT statements. It returns all rows from the queries, including rows returned from more than one SELECT statement. It does not remove duplicate rows from the final result set.

Syntax

The UNION ALL operator has the following syntax:

```
SELECT column1 [, columnN ]
FROM table1 [, tableN ]
[WHERE clause]

UNION ALL

SELECT column1 [, columnN ]
FROM table1 [, tableN ]
[WHERE clause]
```

Examples

The following UNION ALL query returns all product IDs from the first and second SELECT statements:

```
SELECT product_id FROM order_items
UNION ALL
SELECT product_id FROM inventories;
```

The following UNION ALL query uses parentheses to specify the precedence of evaluation :

```
SELECT *
   FROM departments
   UNION ALL
   (SELECT *
   FROM warehouses
```

```
UNION
SELECT *
FROM offices);
```

All three tables must have the same number of columns of the same data type.

Guidelines for Using the UNION Operator

Unless otherwise noted in the guidelines, the Data Vault Service follows the standard SQL rules for the UNION operator.

Use the following guidelines when you include UNION or UNION ALL in a query:

- You can use parentheses to specify the order of evaluation.
  The following example is a valid UNION query for use with the Data Vault Service:

  ```
  SELECT c1, c2, c3 FROM Table1
  UNION
  (SELECT c1, c2, c3 FROM Table2
  UNION
  SELECT c1, c2, c3 FROM Table3);
  ```

  In this example, the Data Vault Service evaluates the union between Table2 and Table3. Then the Data Vault Service evaluates the union between Table1 and the result set of the union between Table2 and Table3.
- If you use the Data Vault Service ssasql command line program to run queries, you cannot use parentheses to enclose the first SELECT statement in a UNION query.
  The following example is not a valid use of parentheses in a UNION query that you run from the Data Vault Service ssasql command line program:

  ```
  (SELECT c1, c2, c3 FROM Table1)
  UNION
  SELECT c1, c2, c3 FROM Table2;
  ```
- The number of columns selected from a table must be the same for each SELECT statement. If you select all columns from a table with SELECT *, the number of columns in the table must be the same as the number of columns selected for other tables.
- The data types of the selected columns must be compatible. For example, integer and decimal data types are compatible. In the following example, LastName from Customers is compatible with VendorName from Suppliers, City is compatible with City, MembershipStart is compatible with LastUpdated.

  ```
  SELECT LastName, City, MembershipStart FROM Customers
  UNION
  SELECT VendorName, City, LastUpdated FROM Suppliers
  ORDER BY City;
  ```
- The columns in the final result set take the names of the columns in the first SELECT statement. For example, the columns in the final result set for the following query will be named LastName and City.

  ```
  SELECT LastName, City FROM Customers
  UNION
  SELECT VendorName, City FROM Suppliers
  ORDER BY City;
  ```
- The columns in the final result set take the largest column size in all the SELECT statements. For example, if the SELECT statements include a column with small integer, integer, and decimal data types, the datatype for the column in final result set will be decimal.
- You can use the UNION or UNION ALL operator in a subquery.

CHAPTER 5

Parameterized Query

This chapter includes the following topics:

Parameterized Query Overview

A parameterized query uses parameters or placeholders in place of values. When you define the SQL statement, you can use placeholders in place of the values. When you run the SQL statement, supply the values for the placeholders in a separate statement. Parameterized queries are also known as prepared statements.

The SQL client that you use must support parameterized queries. The SQL client determines the syntax for the parameterized queries. Each SQL client has its own method for preparing and running parameterized queries.

Unless otherwise noted in the guidelines, the Data Vault Service follows the standard SQL rules for parameterized queries.

Parameterized Query Usage

You can use parameters for values used with operators, predicates, and functions. You must use a question mark (?) as a placeholder for the value.

Operators

You can use placeholders for values in mathematical operations (+ - * /) and the string concatenation operation (||).

For example, the following query uses a placeholder for one of the values in the addition operation:

```
SELECT *
FROM customer
WHERE age + ? > 100;
```

Predicates

You can use placeholders with the following predicates: > < >= <= = != BETWEEN IN LIKE

For example, the following query uses placeholders for values in the >= and <= predicates:

```
SELECT *
FROM employee
 WHERE emp_no >= ? and emp_no <= ?;
```

The following query uses a placeholder for the value in the LIKE predicate:

```
Select * from employee where first_name like ?;
```

Functions

You can use placeholders with Data Vault Service SQL functions except aggregate functions and the COALESCE, IFNULL, and NULLIF functions.

When you use a parameter in place of a function argument, verify that the datatype of the parameter matches the datatype required for the argument.

You cannot use parameters in the following functions:

- AVG
- COALESCE
- COUNT
- IFNULL
- MAX
- MIN
- NULLIF
- SUM

For more information about the functions available for Data Vault Service queries, see Chapter 6, "Functions" on page 59.

Guidelines for Using Parameterized Queries

You can include parameters in any type of query.

Use the following guidelines when you include parameters in your queries:

- You must use a question mark (?) as a placeholder.
- When you use a placeholder with an operator or a predicate, only one side of the expression can use a placeholder. One side of the operation must always contain a value for which the Data Vault Service can determine the datatype.
  For example, the following query is valid:

  ```
  SELECT account_balance
  FROM users
  WHERE user_name = ?;
  ```

  The following query is not valid:

  ```
  SELECT account_balance
  FROM users
  WHERE ? <= ?;
  ```

CHAPTER 6

Functions

This chapter includes the following topics:

Functions Overview

A function is a named, specialized operation performed on zero or more input values, that returns a single value when executed successfully. A function can appear anywhere a value expression is permitted. The function is invoked by its name, followed by the input arguments in parentheses.

The following table describes the types of functions that you can use:

Function Type	Description
Aggregate	An operation applied to the values contained in a particular column (or in a derived column created by a value expression), producing a single value that summarizes the (derived) column. The *value expression* argument cannot itself contain an aggregate function.
Cast	The cast functions are used to convert input values from one format to another. A cast function can appear anywhere a value expression is permitted in SQL syntax.
String	The string functions are used to manipulate character string values. A string function can be used anywhere a string value expression is permitted.

Function Type	Description
Math	The math functions are used for manipulating numeric values. A math function can appear wherever a numeric value expression is allowed.
Date/Time	The date/time functions are used to manipulate DATE, TIME, and TIMESTAMP values, and can appear wherever a value expression is permitted.
NULLIF, COALESCE, IFNULL	The three functions that are designed to handle a subset of the CASE functionality.

ABS

A math function that returns the absolute value of the input expression.

ABS Syntax

```
ABS (double)
```

ABS Examples

Input	Output
-1	1
1	1

```
CREATE TABLE num_table (col1 SMALLINT);
INSERT INTO num_table VALUES (-1);
1 row affected
INSERT INTO num_table VALUES (1);
1 row affected
SELECT ABS(col1) FROM num_table;
2 rows selected

1
------
     1
```

AVG

An aggregate function that returns the average of the non-null values in the specified or derived column (numeric data only).

AVG Syntax

```
AVG (value-expression)
AVG (DISTINCT value-expression)
```

CEILING

A math function that returns the smallest integer that is greater than or equal to the input expression.

CEILING Syntax

```
CEILING (double)
```

CEILING Example

Input	Output
9.3	10

```
CREATE TABLE num_table (col1 DEC(5,2));
INSERT INTO num_table VALUES (9.3);
1 row affected
SELECT CEILING(col1) FROM num_table;
1 row selected
1
----------------------
  1.00000000000000e+01
```

CHAR

A cast function that converts the input value to the CHAR datatype equivalent and returns the value. DATE and TIME values are converted according to the picture format in the nucleus.ini file, unless a second parameter that specifies a particular date or time picture (between quotation marks) is included.

CHAR Syntax

```
CHAR (char)
CHAR (char [, 'picture format'])
CHAR (timestamp [, 'picture format'])
```

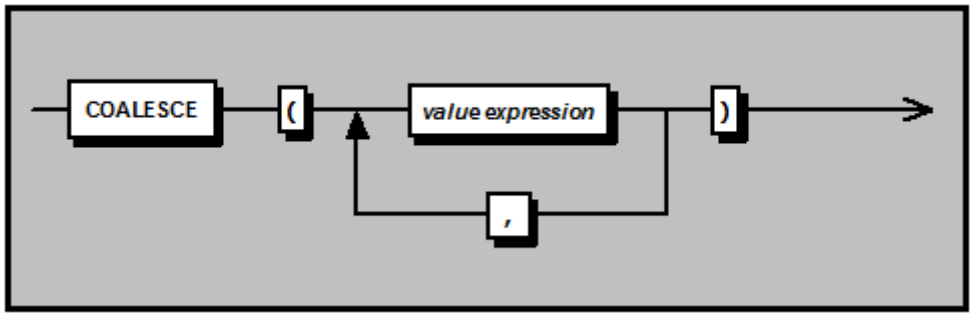
CHAR Examples

Input	Output
'2004-01-31', 'mmmm dd, yyyy'	'January 31, 2004'
1000 + 200 + 30 + 4	'1234'

```
CREATE TABLE string_table (col1 VARCHAR(20));
CREATE TABLE date_table (c1 DATE);
INSERT INTO date_table VALUES ('2004-01-31');
1 row affected
INSERT INTO string_table SELECT CHAR(c1, 'mmmm dd, yyyy') FROM date_table;
1 row affected
INSERT INTO string_table VALUES (CHAR(1000+200+30+4));
1 row affected
SELECT * FROM string_table;
2 rows selected
COL1
--------------------
January 31, 2004
  1234
```

COALESCE

Returns the first argument that is not null. The arguments are evaluated in the order in which they are specified. The result is null only if all the arguments are null.



COALESCE Example

```
SELECT lastname,
       job_desc,
       COALESCE(salary, contract, commission, subsistence)
FROM payroll;
```

In this example, a worker on the payroll is paid either a regular salary, contract pay, a commission, or subsistence wages. The COALESCE function returns the value for the appropriate pay type, assuming all but the applicable pay field store null values. If none of the pay types apply, a null value is returned.

The equivalent (searched) CASE expression for this example would be the following:

```
SELECT lastname,
       job_desc,
       CASE
          WHEN salary IS NOT NULL THEN salary
          WHEN contract IS NOT NULL THEN contract
          WHEN commission IS NOT NULL THEN commission
          WHEN subsistence IS NOT NULL THEN subsistence
          ELSE NULL
       END
FROM payroll;
```

CONCAT

A string function that is the explicit function counterpart to the concatenation operator ("||"). CONCAT() returns the character string produced by concatenating the first argument with the second argument. An error is generated if either of the input arguments does not evaluate to a character string. The maximum size of the result string is 4056 characters.

CONCAT Syntax

```
CONCAT (string, string)
```

CONCAT Example

Input	Output
'hello', 'world'	'helloworld'

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES (CONCAT('hello', 'world'));
1 row affected
SELECT col1 FROM string_table;
1 row selected
col1
--------------------
helloworld
```

COUNT

An aggregate function.

COUNT Syntax

- Returns the number of rows in the table or virtual table.

  ```
  COUNT (*)
  ```

- Returns the number of non-null values in the specified or derived column, when duplicates are omitted.

  ```
  COUNT (DISTINCT column)
  ```

- Returns the number of non-null values in the specified or derived column.

  ```
  COUNT (value-expression)
  ```

DATE

A cast function that converts a properly formatted input expression to the DATE datatype and returns the value. The input expression can evaluate to the CHAR, VARCHAR, TIMESTAMP, or DATE datatype. For the character input types, the following formats are permitted:

- 'YYYY-MM-DD'
- 'MM/DD/YYYY'
- 'DD.MM.YYYY'
- Any legal format that matches the date picture (DatePic) specified in the [CLIENT] section of the nucleus.ini file.

If the input type is TIMESTAMP, the DATE portion of the value is returned. If the input type is DATE, the value is returned unchanged.

The DATE (month, day, year) form of the function constructs a DATE value from integer values.

DATE Syntax

```
DATE (char)
DATE (timestamp)
DATE (month, day, year)
```

DATE Example

Input	Output
'22.11.1975'	1975-11-22

```
CREATE TABLE emp_data (eyear CHAR(4), emonth CHAR(2), eday CHAR(2));
INSERT INTO emp_data VALUES ('1975', '11', '22');
1 row affected
CREATE TABLE birthdates (d1 DATE);
INSERT INTO birthdates SELECT DATE(eday || '.' || emonth || '.' || eyear) FROM emp_data;
1 row affected
SELECT d1 FROM birthdates;
1 row selected
d1
----------
1975-11-22
```

DATE (Date and Time)

A date and time function that returns the equivalent DATE datatype value for the specified month, day, and year. The input values must be integers, and they must represent a valid date.

DATE Syntax

```
DATE (month, day, year)
```

DATE Result

Input	Output
12, 31, 2003	2003-12-31

```
CREATE TABLE emp_data (emonth INT, eday INT, eyear INT);
INSERT INTO emp_data VALUES (12, 31, 2003);
1 row affected
SELECT DATE(emonth, eday, eyear) FROM emp_data;
1 row selected
1
----------
2003-12-31
```

DAY

A date and time function that returns the day portion of a DATE or TIMESTAMP value as an INTEGER value.

DAY Syntax

```
DAY (char)
```

DAY Examples

Input	Output
'11/27/1969'	27

Input	Output
'2004-02-13-14.14.59.624000'	13

```
CREATE TABLE dt_table (date_col DATE, timestamp_col TIMESTAMP);
INSERT INTO dt_table VALUES ('11/27/1969', '2004-02-13-14.14.59.624000');
1 row affected
SELECT date_col, DAY(date_col) AS day FROM dt_table;
1 row selected


date_col    day
----------  -----------
1969-11-27           27
SELECT timestamp_col, DAY(timestamp_col) AS day FROM dt_table;
1 row selected
timestamp_col               day
--------------------------  -----------
2004-02-13-14.14.59.624000           13
```

DEC

A cast function that converts a numeric or string input to the DECIMAL datatype and returns the value. Optionally, the precision and scale for the converted DECIMAL value can be set. If omitted, the following default precisions are used, depending on the input type:

- SMALLINT: **5**
- INTEGER: **11**
- UNSIGNED: **19**
- other: **15**

The default scale is **0** in all cases.

DEC Syntax

```
DEC ( char value-expression [ , precision [ , scale ] ] )
DECIMAL ( char value-expression [ , precision [ , scale ] ] )
```

DEC Examples

Input	Output
'1234.567', 10, 5	1234.56700
5.67192e3	5672

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES ('1234.567');
1 row affected
SELECT DEC(col1, 10, 5) FROM string_table;
1 row selected
1
------------
  1234.56700
CREATE TABLE num_table (col1 FLOAT);
INSERT INTO num_table VALUES (5.67192e3);
1 row affected
SELECT DEC(col1) FROM num_table;
1 row selected
1
```

```
-----------------
             5672
```

DIGITS

A cast function that returns a string representation of the numeric input expression, excluding non-digits. The input value's numeric sign, decimal point, and/or exponent character (if applicable) are not part of the return value. The length of the returned string is the precision of the input value, padded with leading zeroes if necessary.

DIGITS Syntax

```
DIGITS (decimal)
```

DIGITS Example

Input	Output
-123456.789	'0123456789'

```
CREATE TABLE dec1 (col1 DEC(10, 3));
INSERT INTO dec1 VALUES (-123456.789);
1 row affected
CREATE TABLE varchar1 (c1 VARCHAR(10));
INSERT INTO varchar1(c1) SELECT DIGITS(col1) FROM dec1;
1 row affected
SELECT c1 FROM varchar1;
1 row selected
c1
----------
0123456789
```

EXP

A math function that returns a FLOAT value of the logarithmic constant ***e*** (2.718281828) raised to the power of the parameter.

EXP Syntax

```
EXP (double)
```

EXP Example

Input	Output
100	2.68811714181614e+43

```
CREATE TABLE num_table (col1 SMALLINT);
INSERT INTO num_table VALUES (100);
1 row affected
SELECT EXP(col1) FROM num_table;
1 row selected
1
----------------------
  2.68811714181614e+43
```

EXTRACT

A date and time function that returns the specified *component* from the input date/time expression as an INTEGER value.

Use the following keywords to indicate the date component to extract:

- DAY
- MONTH
- YEAR
- HOUR
- MINUTE
- SECOND
- MICROSECOND
  Note that DATE components cannot be extracted from TIME values.

EXTRACT Syntax

```
EXTRACT (DateComponentKeyword FROM char)
```

EXTRACT Examples

Input	Output
DAY FROM '2003-12-29'	29
MONTH FROM '2003-12-29'	12
YEAR FROM '2004-02-17-14.20.06.131000'	2004
HOUR FROM '02:20:34'	2
MINUTE FROM '2004-02-17-14.20.06.131000'	20
SECOND FROM '02:20:34'	34
MICROSECOND FROM '2004-02-17-14.20.06.131000'	131000

```
CREATE TABLE dt_table (date_col DATE, timestamp_col TIMESTAMP, time_col TIME);
INSERT INTO dt_table VALUES ('2003-12-29', '2004-02-17-14.20.06.131000', '02:20:34');
1 row affected
SELECT date_col, EXTRACT(DAY FROM date_col) AS result FROM dt_table;
1 row selected
date_col    result
----------  -----------
2003-12-29           29
SELECT date_col, EXTRACT(MONTH FROM date_col) AS result FROM dt_table;
1 row selected
date_col    result
----------  -----------
2003-12-29           12
SELECT timestamp_col, EXTRACT(YEAR FROM timestamp_col) AS result FROM dt_table;
1 row selected

timestamp_col               result
--------------------------  -----------
2004-02-17-14.20.06.131000         2004
SELECT time_col, EXTRACT(HOUR FROM time_col) AS result FROM dt_table;
1 row selected
time_col  result
--------  -----------
```

```
02:20:34           2
SELECT timestamp_col, EXTRACT(MINUTE FROM timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-14.20.06.131000           20
SELECT time_col, EXTRACT(SECOND FROM time_col) AS result FROM dt_table;
1 row selected
time_col  result
--------  -----------
02:20:34           34
SELECT timestamp_col, EXTRACT(MICROSECOND FROM timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-14.20.06.131000       131000
```

FLOOR

A math function that returns the largest integer that is less than or equal to the input expression.

FLOOR Syntax

```
FLOOR (double)
```

FLOOR Example

Input	Output
9.3	9

```
CREATE TABLE num_table (col1 DEC(5,2));
INSERT INTO num_table VALUES (9.3);
1 row affected
SELECT FLOOR(col1) FROM num_table;
1 row selected
1
----------------------
  9.00000000000000e+00
```

HOUR

A date and time function that returns the hours portion of a TIME or TIMESTAMP value as an INTEGER value.

HOUR Syntax

```
HOUR (char)
```

HOUR Examples

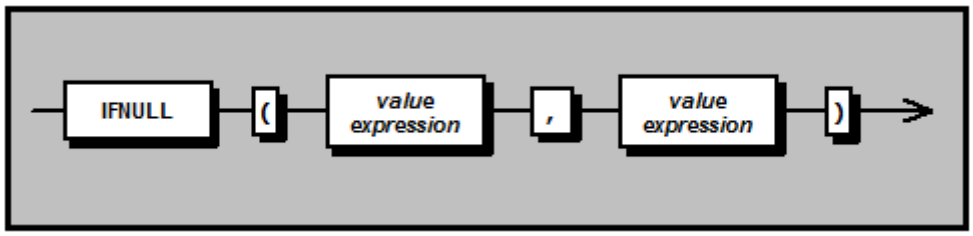
Input	Output
'15:08:37'	15

Input	Output
'2004-02-17-15.08.37.588000'	15

```
CREATE TABLE dt_table (time_col TIME, timestamp_col TIMESTAMP);
INSERT INTO dt_table VALUES ('15:08:37', '2004-02-17-15.08.37.588000');
1 row affected
SELECT time_col, HOUR(time_col) AS result FROM dt_table;
1 row selected
time_col  result
--------  -----------
15:08:37           15
SELECT timestamp_col, HOUR(timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-15.08.37.588000           15
```

IFNULL

Returns the first of two arguments that is not null. This is the same as calling the COALESCE function with only two arguments. As with the COALESCE function, the IFNULL arguments are evaluated in the order in which they are specified, and the result is null only if both arguments are null.



IFNULL Example

```
SELECT item, IFNULL(price, -1) AS Price
FROM inventory;
```

In this example, a SELECT statement returns a list each item and its associated price from the *inventory* table. The IFNULL function is used here to intercept item prices that have not been set (and are therefore null) and return a value of **-1** instead. For items whose price is not null, the IFNULL function returns the price value unchanged.

The equivalent (searched) CASE expression for this example would be the following:

```
SELECT item,
       CASE
          WHEN price IS NOT NULL THEN price
          ELSE -1
       END
       AS Price
FROM inventory;
```

INT

A cast function that returns the INTEGER datatype representation of a numeric or character input expression. The value produced by the conversion must fall within the range of the INTEGER type, or else an error condition will result. If present, the decimal part of the input value is rounded.

INT Syntax

```
INT (char)
INTEGER (char)
```

INT Examples

Input	Output
'1.024e4'	10240
1.024099999999e+04	10241

```
CREATE TABLE temp_int (col1 INT);
INSERT INTO temp_int values (INTEGER('1.024e4'));
1 row affected
INSERT INTO temp_int values (INTEGER(1.024099999999e+04));
1 row affected
SELECT * FROM temp_int;
2 rows selected
COL1
-----------
      10240
      10241
```

LASTDAY

Returns the date of the last day of the month in the DateString.

Syntax

```
LASTDAY ( DateString )
```

Argument	DataType	Description
DateString	char, date, or timestamp	Date string from which to get the last day. The format of the date string can be a date or a time stamp.

Return Value

Date of the last day of the month specified in the DateString.

Example

The function call lastday('2013-09-12') returns the last day of September: 2013-09-30

LEFT

A string function that returns a substring from *value-expression* consisting of the number of leftmost characters indicated by *length*. The *length* parameter is a numeric expression that evaluates to an integer value. If the value of *length* is greater than or equal to the length of the input string, the whole string is returned.

LEFT Syntax

```
LEFT ( char value-expression, integer length)
```

LEFT Example

Input	Output
'isopropyl', 3	'iso'

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES (LEFT('isopropyl', 3));
1 row affected
SELECT col1 FROM string_table;
1 row selected


col1
--------------------
iso
```

LEN

A string function that returns the number of characters in the input expression.

LEN Syntax

```
LEN (char)
LENGTH (char)
```

LEN Example

Input	Output
'sea bass'	8

```
CREATE TABLE string_table (col1 VARCHAR(10));
INSERT INTO string_table VALUES ('sea bass');
1 row affected
SELECT LENGTH(col1) FROM string_table;
1 row selected
1
-----------
          8
```

LN

A math function that returns the natural logarithm of the input expression. If the input expression evaluates to a negative number or **0**, an error message is returned.

LN Syntax

```
LN (double)
LOG (double)
```

LN Example

Input	Output
120	4.78749174278205e+00

```
CREATE TABLE num_table (col1 SMALLINT);
INSERT INTO num_table VALUES (120);
1 row affected
SELECT LN(col1) FROM num_table;
1 row selected
1
----------------------
  4.78749174278205e+00
```

LOG10

A math function that returns the base 10 logarithm of the input expression. If the input expression evaluates to a negative number or 0, an error message is returned.

LOG10 Syntax

```
LOG10 (double)
```

LOG10 Example

Input	Output
2	3.01029995663981e-01

```
CREATE TABLE num_table (col1 SMALLINT);
INSERT INTO num_table VALUES (2);
1 row affected
SELECT LOG10(col1) FROM num_table;
1 row selected
1
----------------------
  3.01029995663981e-01
```

LOWER

A string function that returns the input expression with all uppercase characters converted to their lowercase equivalents.

LOWER Syntax

```
LOWER (char)
LCASE (char)
```

LOWER Example

Input	Output
'LOUD'	'loud'

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES (LOWER('LOUD'));
1 row affected
SELECT col1 FROM string_table;
1 row selected
col1
--------------------
loud
```

LTRIM

A string function that returns the input string with leading spaces removed.

LTRIM Syntax

```
LTRIM (char)
```

LTRIM

Input	Output
' snipped'	'snipped'

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES (LTRIM('    snipped'));
1 row affected
SELECT col1 FROM string_table;
1 row selected
col1
--------------------
snipped
```

MAX

An aggregate function that returns the maximum value in the specified column or derived column.

MAX Syntax

```
MAX (value-expression)
```

MICROSECOND

A date and time function that returns the fractional second portion of a time stamp value in microsecond precision. The function returns the microsecond value as an integer.

Syntax

```
MICROSECOND (char)
```

Example

Input	Output
'2004-02-17-15.08.37.588000'	588000
'2004-02-17-15.08.37.588000123456'	588000

```
CREATE TABLE dt_table (timestamp_col TIMESTAMP);
INSERT INTO dt_table VALUES ('2004-02-17-15.08.37.588000');
1 row affected
SELECT timestamp_col, MICROSECOND(timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-15.08.37.588000       588000
```

MIN

An aggregate function that returns the minimum value in the specified or derived column.

MIN Syntax

```
MIN (value-expression)
```

MINUTE

A date and time function that returns the minutes portion of a TIME or TIMESTAMP value as an INTEGER value.

MINUTE Syntax

```
MINUTE (char)
```

MINUTE Examples

Input	Output
'15:08:37'	8
'2004-02-17-15.08.37.588000'	8

```
CREATE TABLE dt_table (time_col TIME, timestamp_col TIMESTAMP);
INSERT INTO dt_table VALUES ('15:08:37', '2004-02-17-15.08.37.588000');
1 row affected
SELECT time_col, MINUTE(time_col) AS result FROM dt_table;
```

```
1 row selected
time_col  result
--------  -----------
15:08:37            8
SELECT timestamp_col, MINUTE(timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-15.08.37.588000            8
```

MOD

A math function that returns the remainder of *value-expression1* divided by *value-expression2*. The return value is negative only if *value-expression1* is negative. If *value-expression2* is 0, the function returns *value-expression1*. If either of the arguments is a null value, the result is a null value.

MOD Syntax

```
MOD (double value-expression1, double value-expression2)
```

MOD Examples

Input	Output
8, 5	3
127,62	3
-4,2	0
-4,3	-1
NULL,1	NULL

```
CREATE TABLE num_table (exp1 SMALLINT, exp2 SMALLINT, result SMALLINT);
INSERT INTO num_table VALUES (8, 5, );
1 row affected
INSERT INTO num_table VALUES (127, 62, );
1 row affected
INSERT INTO num_table VALUES (-4, 2, );
1 row affected
INSERT INTO num_table VALUES (-4, 3, );
1 row affected
INSERT INTO num_table VALUES ( , 1, );
1 row affected
UPDATE num_table SET result = MOD(exp1, exp2);
5 rows affected
.NULLS *****
SELECT * FROM num_table;
5 rows selected
EXP1    EXP2    RESULT
------  ------  ------
     8       5       3
   127      62       3
    -4       2       0
    -4       3      -1
*****        1  *****
```

MONTH

A date and time function that returns the month portion of a DATE or TIMESTAMP value as an INTEGER value.

MONTH Syntax

```
MONTH (char)
```

MONTH Examples

Input	Output
'02/17/2004'	2
'2004-02-17-15.08.37.588000'	2

```
CREATE TABLE dt_table (date_col DATE, timestamp_col TIMESTAMP);
INSERT INTO dt_table VALUES ('02/17/2004', '2004-02-17-15.08.37.588000');
1 row affected
SELECT date_col, MONTH(date_col) AS result FROM dt_table;
1 row selected
date_col    result
----------  -----------
2004-02-17            2
SELECT timestamp_col, MONTH(timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-15.08.37.588000            2
```

NANOSECOND

A date and time function that returns the fractional second portion of a time stamp value in nanosecond precision. The function returns the nanosecond value as an integer.

Syntax

```
NANOSECOND (char)
```

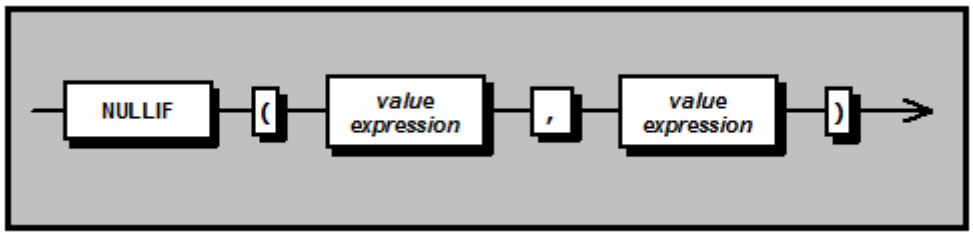
Example

Input	Output
'2004-02-17-15.08.37.588000'	588000000000
'2004-02-17-15.08.37.588000123456'	588000123456

```
CREATE TABLE dt_table (timestamp_col TIMESTAMP);
INSERT INTO dt_table VALUES ('2004-02-17-15.08.37.588000');
1 row affected
SELECT timestamp_col, NANOSECOND(timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-15.08.37.588000       588000000000
```

NULLIF

Returns a null value if the arguments are equal, otherwise it returns the value of the first argument.



NULLIF Example

```
SELECT item, NULLIF(cost, -1)
FROM inventory;
```

For this example, say that a cost value of -1 indicates no cost at all. With the .NULLS system variable set to 'N/A', it might be more informative to return a null value instead of -1. Using the NULLIF function, a null value can be substituted for every return of -1.

The equivalent (simple) CASE expression for this example would be the following:

```
SELECT item,
       CASE cost
          WHEN -1 THEN NULL
          ELSE cost
       END
FROM inventory;
```

PI

A math function that returns the constant **pi** (that is, **3.14159265...**) as a FLOAT value.

PI Syntax

```
PI ( )
```

PI Example

```
CREATE TABLE num_table (col1 SMALLINT);
INSERT INTO num_table VALUES (2);
1 row affected
SELECT PI(), PI()*col1 FROM num_table;
1 row selected
1                      2
----------------------  ----------------------
  3.14159265358979e+00    6.28318530717958e+00
```

POSITION

A string function that returns the starting position of a string *value-expression1* within another string *value-expression2*. The Data Vault Service evaluates the position of the string from left to right, starting at position 1. If *value-expression1* is not found, POSITION returns 0. If *value-expression1* occurs more than once within *value-expression2*, POSITION returns the starting position of the last occurrence of *value-expression1*.

The POSITION function is similar to the POSSTR function. Both functions return the position of a substring within a string.

POSITION Syntax

```
POSITION ( char value-expression1 IN char value-expression2 )
```

POSITION Example 1

In this example, *value-expression1* occurs once within *value-expression2*. POSITION returns the starting position of *value-expression1*.

Input	Output
'fun' IN 'malfunction'	4

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES ('malfunction');
1 row affected
SELECT POSITION('fun' IN col1) FROM string_table;
1 row selected
1
-----------
          4
```

POSITION Example 2

In this example, *value-expression1* occurs more than once within *value-expression2*. POSITION returns the starting position of the last occurrence of *value-expression1*.

Input	Output
'4' IN '4234.23423'	8

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES ('4234.23423');
1 row affected
SELECT POSITION('4' IN col1) FROM string_table;
1 row selected
1
-----------
          8
```

POSSTR

A string function that returns the starting position of a string *value-expression1* within another string *value-expression2*. The Data Vault Service evaluates the position of the string from left to right, starting at position 1. If *value-expression1* is not found, POSSTR returns 0. If *value-expression1* occurs more than once within *value-expression2*, POSSTR returns the starting position of the first occurrence of *value-expression1*.

The POSSTR function is similar to the POSITION function. Both functions return the position of a substring within a string.

POSSTR Syntax

```
POSSTR (char value-expression2, char value-expression1)
```

POSSTR Example 1

In this example, *value-expression1* occurs once within *value-expression2*. POSSTR returns the starting position of *value-expression1*.

Input	Output
'solstice', 'ice'	6

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES ('solstice');
1 row affected
SELECT POSSTR(col1, 'ice') FROM string_table;
1 row selected
1
-----------
          6
```

POSSTR Example 2

In this example, *value-expression1* occurs more than once within *value-expression2*. POSSTR returns the starting position of the last occurrence of *value-expression1*.

Input	Output
'4234.23423', '4'	8

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES ('4234.23423');
1 row affected
SELECT POSSTR(col1, '4') FROM string_table;
1 row selected
1
-----------
          8
```

POWER

A math function that returns a value that is calculated as *value-expression1* raised to the power of *value-expression2*. If both input values are integers, then the result is INTEGER; otherwise the result is a FLOAT value.

POWER Syntax

```
POWER (double value-expression1, double value-expression2)
POW (double value-expression1, double value-expression2)
```

POWER Examples

Input	Output
4, 8	65536
4.0, 8	6.55360000000000e+04
4, 0.5	2.00000000000000e+00

Input	Output
4, -2.0	6.25000000000000e-02

```
CREATE TABLE num_table (col1 SMALLINT, col2 SMALLINT);
INSERT INTO num_table VALUES (4, 8);
1 row affected
SELECT POWER(col1, col2) FROM num_table;
1 row selected
1
-----------
      65536
SELECT POWER(4.0, col2) FROM num_table;
1 row selected
1
----------------------
  6.55360000000000e+04
SELECT POWER(col1, 0.5) FROM num_table;
1 row selected
1
----------------------
  2.00000000000000e+00
SELECT POWER(col1, -2.0) FROM num_table;
1 row selected
1
----------------------
  6.25000000000000e-02
```

REPLACE

Replaces characters in a string with a specified substring. REPLACE searches the input string for all instances of the substring you specify and replaces them with the replacement string you specify.

Syntax

```
REPLACE ( InputString, Substring[, ReplacementString] )
```

Argument	DataType	Description
InputString	char	String in which to search for a substring to be replaced by another string.
Substring	char	Substring to search for that you want to replace with ReplacementString.
ReplacementString	char	Optional. String to replace Substring. If you do not pass this parameter, REPLACE deletes all occurrences of Substring.

Return Value

String with the specified substring replaced or deleted.

Example

If a column named col1 contains the string "The fox sees another fox in the meadow.", the following expression replaces all instances of the substring "fox" with the string "rabbit":

```
REPLACE ( col1, 'fox', 'rabbit' )
```

The function call returns the following string:

```
The rabbit sees another rabbit in the meadow.
```

RIGHT

A string function that returns a substring from *value-expression* consisting of the number of rightmost characters indicated by *length*. The *length* parameter is a numeric expression that evaluates to an integer value. If the value of *length* is greater than or equal to the length of the input string, the whole string is returned.

RIGHT Syntax

```
RIGHT ( char value-expression, integer length)
```

RIGHT Example

Input	Output
'isopropyl'	'pyl'

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES ('isopropyl');
1 row affected
SELECT RIGHT(col1, 3) FROM string_table;
1 row selected
1
---
pyl
```

ROUND

A math function that returns the value produced by rounding *value-expression1* to *value-expression2* digits to the right of the decimal point. A negative number of digits indicates rounding to the left of the decimal point. Note that *value-expression1* can be any numeric value, but *value-expression2* must be an integer value.

ROUND Syntax

```
ROUND (double value-expression1, integer value-expression2 )
```

ROUND Example

Input	Output
234.56789, 2	234.57
234.56789, -2	200

```
CREATE TABLE num_table (col1 DEC(10, 2));
INSERT INTO num_table VALUES (234.56789);
1 row affected
SELECT ROUND(col1, 2) FROM num_table;
1 row selected
1
-------------
       234.57
SELECT ROUND(col1, -2) FROM num_table;
1 row selected
1
-------------
       200.00
```

RTRIM

A string function that returns the input string with trailing spaces removed.

RTRIM Syntax

```
RTRIM (char)
```

RTRIM Example

Input	Output
'pruned '	'pruned'

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES ('pruned        ');
1 row affected
SELECT RTRIM(col1) FROM string_table;
1 row selected
1
--------------
pruned
```

SECOND

A date and time function that returns the seconds portion of a TIME or TIMESTAMP value as an INTEGER value.

SECOND Syntax

```
SECOND (char)
```

SECOND Examples

Input	Output
'15:08:37'	37
'2004-02-17-15.08.37.588000'	37

```
CREATE TABLE dt_table (time_col TIME, timestamp_col TIMESTAMP);
INSERT INTO dt_table VALUES ('15:08:37', '2004-02-17-15.08.37.588000');
1 row affected
SELECT time_col, SECOND(time_col) AS result FROM dt_table;
1 row selected
time_col  result
--------  -----------
15:08:37           37
SELECT timestamp_col, SECOND(timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-15.08.37.588000           37
```

SIGN

A math function that returns a numeric value that indicates the sign of the input expression. SIGN can have the following return values:

- -1. Value is less than zero.
- 0. Value is zero.
- 1. Value is greater than zero.

If the input value is an INTEGER, then the return value is INTEGER as well. Similarly, if the input is SMALLINT, the output is SMALLINT. Otherwise, a FLOAT value is returned.

SIGN Syntax

```
SIGN (double)
```

SIGN Examples

Input	Output
173	1
-173	-1
-173.34	-1.00000000000000e+00
173.34	1.00000000000000e+00
0	0

```
CREATE TABLE num_table (int_col INT, float_col FLOAT);
INSERT INTO num_table VALUES (173, 173.34);
1 row affected
INSERT INTO num_table VALUES (-173, -173.34);
1 row affected
INSERT INTO num_table VALUES (0, );
1 row affected
SELECT int_col, SIGN(int_col) AS result FROM num_table;
3 rows selected
int_col      result
-----------  -----------
        173            1
       -173           -1
          0            0
SELECT float_col, SIGN(float_col) AS result FROM num_table WHERE float_col IS NOT NULL;
2 rows selected
float_col               result
----------------------  ----------------------
  1.73340000000000e+02    1.00000000000000e+00
 -1.73340000000000e+02   -1.00000000000000e+00
```

SQRT

A math function that returns a FLOAT value that is the square root of the input expression.

SQRT Syntax

```
SQRT (double)
```

SQRT Examples

Input	Output
81	9.00000000000000e+00
4678	6.83959063102464e+01

```
CREATE TABLE num_table (col1 INT);
INSERT INTO num_table VALUES (81);
1 row affected
INSERT INTO num_table VALUES (4678);
1 row affected
SELECT col1, SQRT(col1) AS result FROM num_table;
2 rows selected
col1         result
-----------  ----------------------
         81    9.00000000000000e+00
       4678    6.83959063102464e+01
```

SUBSTRING

A string function that returns the specified portion of the input expression. *Value-expression* is the input character string; *start-position* is the starting position of the substring, counting from the left (the first character in the input string is position 1, the second character is position 2, and so on); and *length* is the number of characters to return, counting from left to right from the substring starting position. If the length argument is omitted, the substring from the starting position to the end of the input string is returned.

SUBSTRING Syntax

```
 SUBSTRING ( char value-expression, integer start-position [, integer length ] )
 SUBSTRING ( char value-expression FROM integer start-position [ FOR integer length ] )
 SUBSTR ( char value-expression, integer start-position [, integer length ] )
SUBSTR ( char value-expression FROM integer start-position [ FOR integer length ] )
```

SUBSTRING Examples

Input	Output
'12345', 2, 2	'23'
'PK1-H4V1Y9-QC', 5, 6	'H4V1Y9'
'514-939-3477', 5	'939-3477'

```
CREATE TABLE string_table (col1 SMALLINT, col2 VARCHAR(20));
INSERT INTO string_table VALUES (1, '12345');
1 row affected
INSERT INTO string_table VALUES (2, 'PK1-H4V1Y9-QC');
1 row affected
INSERT INTO string_table VALUES (3, '514-939-3477');
1 row affected
SELECT SUBSTR(col2, 2, 2) FROM string_table WHERE col1=1;
1 row selected
1
--
23
SELECT SUBSTRING(col2 FROM 5 FOR 6) FROM string_table WHERE col1=2;
1 row selected
```

```
1
------
H4V1Y9
SELECT SUBSTR(col2, 5) FROM string_table WHERE col1=3;
1 row selected
1
----------------
939-3477
```

SUM

An aggregate function.

SUM Syntax

- Returns the sum of the non-null values in the specified or derived column. It is only for numeric data.

  ```
  SUM (value-expression)
  ```

- Returns the sum of the non-null values in the specified or derived column, when duplicates are omitted. It is only for numeric data.

  ```
  SUM (DISTINCT value-expression)
  ```

TIME

A cast function that returns a TIME datatype representation of a TIME, TIMESTAMP, or character string input expression. An input string expression must correspond to either the standard TIME format (hh:mm:ss), or any legal format that matches the time picture (TimePic) specified in the [CLIENT] section of the nucleus.ini file.

TIME Syntax

```
TIME (char)
```

TIME Example

Input	Output
'1999-03-01-01.59.30.123456'	01:59:30
'23:12:23'	23:12:23

```
CREATE TABLE temp_time (col1 TIME);
CREATE TABLE temp_timestamp (c1 TIMESTAMP);
INSERT INTO temp_timestamp VALUES ('1999-03-01-01.59.30.123456');
1 row affected
INSERT INTO temp_time SELECT TIME(c1) FROM temp_timestamp;
1 row affected
INSERT into temp_time VALUES (TIME('23:12:23'));
1 row affected
SELECT col1 FROM temp_time;
2 rows selected
col1
--------
01:59:30
11:12:23
```

TODAY

A date and time function that returns the current date as a DATE value. (The same functionality is provided by the special constant CURRENT DATE.)

TODAY Syntax

```
TODAY ( )
```

TODAY Example

```
CREATE TABLE dt_table (col1 DATE);
INSERT INTO dt_table VALUES (TODAY());
1 row affected
SELECT * FROM dt_table;
1 row selected
COL1
----------
2004-02-17
```

TRIM

A string function that trims the specified character (*character*) from the beginning and/or end of the input string (*value-expression*) and returns the resulting string. The default trim action is BOTH, and the default *character* value is a blank space. Only a single character can be specified for *character*. If the specified leading/trailing character is not found, the input expression is returned unchanged.

TRIM Syntax

```
TRIM ( [  [ LEADING | TRAILING | BOTH ]  [ char character ]  FROM ]  char  value-
expression )
```

TRIM Examples

Input	**Output**
' midriff '	'midriff'
TRAILING, ' midriff '	' midriff'
LEADING 's' FROM 'ssssSmokin'	'Smokin'

```
CREATE TABLE string_table (col1 CHAR(20));
INSERT INTO string_table VALUES ('  midriff  ');
1 row affected
SELECT TRIM (col1) FROM string_table;
1 row selected
1
--------------------
midriff
SELECT TRIM(TRAILING FROM col1) FROM string_table;
1 row selected
1
--------------------
   midriff
SELECT TRIM(LEADING 's' FROM 'ssssSmokin') FROM string_table;
1 row selected
1
----------
Smokin
```

UPPER

A string function that returns the input expression with all lowercase characters converted to their uppercase equivalents.

UPPER Syntax

```
UPPER (char)
UCASE (char)
```

UPPER Example

Input	Output
'loud'	'LOUD'

```
CREATE TABLE string_table (col1 VARCHAR(20));
INSERT INTO string_table VALUES (UPPER('loud'));
1 row affected
SELECT col1 FROM string_table;
1 row selected
col1
--------------------
LOUD
```

YEAR

A date and time function that returns the year portion of a DATE or TIMESTAMP value as an INTEGER value.

YEAR Syntax

```
YEAR (char)
```

YEAR Examples

Input	Output
'02/17/2004'	2004
'2004-02-17-15.08.37.588000'	2004

```
CREATE TABLE dt_table (date_col DATE, timestamp_col TIMESTAMP);
INSERT INTO dt_table VALUES ('02/17/2004', '2004-02-17-15.08.37.588000');
1 row affected
SELECT date_col, YEAR(date_col) AS result FROM dt_table;
1 row selected
date_col    result
----------  -----------
2004-02-17         2004
SELECT timestamp_col, YEAR(timestamp_col) AS result FROM dt_table;
1 row selected
timestamp_col               result
--------------------------  -----------
2004-02-17-15.08.37.588000         2004
```

INDEX

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Y