teradata.

Teradata Vantage™ - SQL Functions, Expressions, and Predicates

Release 17.00

June 2020

B035-1145-170K
DOCS.TERADATA.COM

Copyright and Trademarks

Product Safety

Safety type	Description
NOTICE	Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury.
⚠ CAUTION	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
⚠ WARNING	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

Third-Party Materials

Warranty Disclaimer

Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: docs@teradata.com.

Contents

Introduction to SQL Functions, Expressions, and Predicates

Teradata Vantage™ is our flagship analytic platform offering, which evolved from our industry-leading Teradata® Database. Until references in content are updated to reflect this change, the term Teradata Database is synonymous with Teradata Vantage.

ARRAY/VARRAY functions, operators, expressions, and methods	*Teradata Vantage™ - Data Types and Literals*, B035-1143
BLOB and CLOB functions	*Teradata Vantage™ - Data Types and Literals*, B035-1143
Calendar and business calendar functions	*Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211
The CAST expression and data type conversion functions	*Teradata Vantage™ - Data Types and Literals*, B035-1143
Compression/decompression functions	*Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
DATASET functions and methods	*Teradata Vantage™ - DATASET Data Type*, B035-1198
DateTime and interval functions and expressions	*Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211
Export width procedures (manage the amount of data exported to client applications)	*Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
File system information macros and functions (gain information on specific data blocks)	*Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
Geospatial functions and methods	*Teradata Vantage™ - Geospatial Data Types*, B035-1181
JSON functions and methods	*Teradata Vantage™ - JSON Data Type*, B035-1150
Map functions, macros, and procedures (optimize the placement of tables across AMPs)	*Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
Period functions and operators	*Teradata Vantage™ - SQL Date and Time Functions and Expressions*, B035-1211
Script installation procedures	*Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
SET operators (INTERSECT, MINUS/ EXCEPT, UNION)	*Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146
Table operators	*Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210

User-defined type expressions and these UDFs supplied by Teradata: • Aggregate • Scalar • Window aggregate	*Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
XML functions and methods	*Teradata Vantage™ - XML Data Type*, B035-1140

Functions, Operators, Expressions, and Predicates

SQL Functions

SQL functions return information about some aspect of the database, depending on the arguments specified at the time the function is invoked.

Functions provide a single result by accepting input arguments and returning an output value. Some SQL functions, referred to as niladic functions, do not have arguments, but they do return values. An example of a niladic SQL function is CURRENT_DATE.

Types of SQL Functions

Function Type	Definition
Scalar	The arguments are individual scalar values of either the same or mixed type that can have different meanings. The result is a single value or null. Scalar functions can be used in any SQL statement where an expression can be used.
Aggregate	The argument is a group of rows. The result is a single value or null. Normally aggregate functions are used in the expression list of a SELECT statement and in the summary list of a WITH clause.
Table	The arguments are individual scalar values of either same or mixed type that can have different meanings. The result is a table. Table functions can be used only within the FROM clause of a SELECT statement. Table functions are a form of user-defined functions and are described in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
Ordered Analytical	The arguments are any normal SQL expression. The result is handled the same way as any other SQL expression. It can be a result column or part of a more complex arithmetic expression. Ordered analytical functions are used in operations that require an ordered set of results rows or that depend on values in a previous row.

Examples of Functions

Function	Description
SELECT CHARACTER_ LENGTH(Details)	Scalar function taking the character or CLOB value in the Details column and returning a numeric value for each row in the Orders table.

Function	Description
FROM Orders;	

SELECT AVG(Salary)
FROM Employee;
	Aggregate function returning a single numeric value for the group of numeric values specified by the Salary column in the Employee table.

Embedded Services System Functions

Vantage provides a set of system functions that support a range of functionality such as string handling, DateTime operations, byte/bit manipulation, and more.

Activating Embedded Services System Functions

Before you can use the embedded services functions, you must run the Database Initialization Program (DIP) utility and execute the DIPALL or DIPSYSFNC script. DIPALL is executed as part of system installation.

The DIP scripts create the TD_SYSFNLIB database which should be used only by the system to support the embedded services functions. Do not store any database objects in this database. Doing so may interfere with the proper operation of the embedded services functions.

If you perform a BAR operation that involves the TD_SYSFNLIB database or the DBC dictionary tables, you must re-execute the DIPALL or the DIPSYSFNC script to reactivate the embedded services functions.

Invoking Embedded Services System Functions

Embedded Services System Functions are a type of UDF. They are located in the TD_SYSFNLIB database. To invoke these system functions, you should use the fully qualified syntax, for example, `TD_SYSFNLIB.CEILING (`*`arg`*`)`.

Optionally, you can omit the database name, and use the function name by itself. For example, `CEILING (`*`arg`*`)`. However, if you do not qualify the function name, and there is also a customer-developed UDF with the same name in the current database or in the SYSLIB database, Vantage follows the normal search path precedence for UDFs, and executes the customer-developed UDF instead of the embedded services system function. For more information on UDF search path precedence, see "UDF Locations" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

To ensure you invoke the embedded services function, do one of the following:

- Use the fully qualified function name, including the containing database: prepend `TD_SYSFNLIB.` when you invoke embedded services system functions.
- Use the SET SESSION UDFSEARCHPATH statement to specify a UDF search path precendence that explicitly searches the TD_SYSFNLIB database before other locations. For more information

on SET SESSION, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- Remove any customer-developed UDFs with the same name from the normal UDF search path.

Implicit Data Type Conversion Rules

Embedded services functions follow the implicit data type conversion rules that apply to UDFs. The UDF implicit type conversion rules are more restrictive than the implicit type conversion Vantage normally uses. If a function argument cannot be converted to the required data type by following the UDF implicit conversion rules, it must be explicitly cast. For details, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Related Information

- For examples of table functions, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For example, TD_SYSFNLIB.*embedded_services_function*.

SQL Operators

SQL operators are symbols and keywords that perform operations on their arguments.

The following types of operators are available in SQL:

- Arithmetic operators such as + and - operate on numeric, DateTime, and Interval data types.
- The concatenation operator || operates on character and byte types.
- Comparison operators such as = and > test the truth of relations between their arguments. For information on comparison operators, see Comparison Operators and Functions.

  Comparison operators are also known as conditional expressions because they result in a value of TRUE, FALSE, or unknown (NULL).
- Set operators, or relational operators, such as INTERSECT and UNION combine result sets from multiple sources into a single result set.

SQL Expressions

SQL expressions specify a value, allowing you to perform arithmetic and logical operations and to generate new values or Boolean results from literals and stored values. An expression can consist of any of the following things:

- Column name
- Literal (sometimes referred to as a constant)
- Function
- USING variable

- Parameter
- Parameter marker (question mark (?) placeholder)
- Combination of column names, literals, and functions connected by operators

Types of Expressions

SQL expressions generally fall into the following categories.

Type	Description
Numeric expression	Expressions are generally classified by the type of result they produce. For example, a numeric expression consists of a column name, literal, function, or combination of column names, literals, and functions connected by arithmetic operators where the result is a numeric type.
String expression	
DateTime expression	
Interval expression	
Period expression, including a derived period	
CASE expression	A CASE expression consists of a set of WHEN/THEN clauses and an optional ELSE clause. A valued CASE expression tests for the first WHEN expression that is equal to a test expression and returns the value of the matching THEN expression. If no WHEN expression is equal to the test expression, CASE returns the ELSE expression, or, if omitted, NULL. A searched CASE expression tests for the first WHEN expression that evaluates to TRUE and returns the value of the matching THEN expression. If no WHEN expression evaluates to TRUE, CASE returns the ELSE expression, or, if omitted, NULL.

Examples of Expressions

Expression	Description
'Test Tech'	Character string literal
1024	Numeric literal
Employee.FirstName	Column name
Salary * 12 + 100	Arithmetic expression producing a numeric value
INTERVAL '10' MONTH * 4	Interval expression producing an interval value
CURRENT_DATE + INTERVAL '2' DAY	DateTime expression producing a DATE value

Expression	Description
CURRENT_TIME - INTERVAL '1' HOUR	DateTime expression producing a TIME value
'Last' || ' Order'	String expression producing a character string value

CASE x
  WHEN 1
  THEN 1001
  ELSE 1002
END
	Valued CASE conditional expression producing a numeric value

SQL Predicates

SQL predicates, also referred to as logical predicates, are types of conditional expressions. They specify a condition of a row or group that has one of three possible states:

- TRUE
- FALSE
- NULL (or unknown)

Predicates can appear in the following:

- WHERE, ON, or HAVING clause to qualify or disqualify rows in a SELECT statement.
- WHEN clause search condition of a searched CASE expression.
- CASE_N function.
- IF, WHILE, REPEAT, and CASE statements in stored procedures.

Types of Logical Predicates

SQL provides the following logical predicates:

- Comparison operators
- [NOT] BETWEEN
- LIKE
- [NOT] IN
- [NOT] EXISTS
- OVERLAPS
- IS [NOT] NULL

Logical Operators that Operate on Predicates

- NOT
- AND
- OR

Predicate Quantifiers

- SOME
- ANY
- ALL

Examples of Predicates

Predicate	Description

SELECT *
FROM Employee
WHERE Salary < 40000;
	Predicate in a WHERE clause specifying a condition for selecting rows from the Employee table.

SELECT SUM(CASE
  WHEN part BETWEEN 100 AND 199
  THEN 0
  ELSE cost
  END)
FROM Orders;
	Predicate in a CASE expression specifying a condition that determines the value passed to the SUM function for a particular row in the Orders table.

Aggregate Functions

The following sections describe SQL aggregate functions.

For information on:

- Window aggregate functions and their Teradata-specific equivalents, see Window Aggregate Functions.
- Aggregate user-defined functions (UDFs), see "Aggregate UDF" in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.
- Window aggregate UDFs, see "Window Aggregate UDF" in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

About Aggregate Functions

Aggregate functions are typically used in arithmetic expressions. Aggregate functions operate on a group of rows and return a single numeric value in the result table for each group.

In the following statement, the SUM aggregate function operates on the group of rows defined by the Sales_Table table:

```
SELECT SUM(Total_Sales)
FROM Sales_Table;
Sum(Total_Sales)
----------------
         5192.40
```

You can use GROUP BY clauses to produce more complex, finer grained results in multiple result values. In the following statement, the SUM aggregate function operates on groups of rows defined by the Product_ID column in the Sales_Table table:

```
SELECT Product_ID, SUM(Total_Sales)
FROM Sales_Table
GROUP BY Product_ID;
Product_ID  Sum(Total_Sales)
----------  ----------------
       101           2100.00
       107           1000.40
       102           2092.00
```

Aggregates in the Select List

Aggregate functions are normally used in the expression list of a SELECT statement and in the summary list of a WITH clause.

Aggregates and GROUP BY

If you use an aggregate function in the select list of an SQL statement, then either all other columns occurring in the select list must also be referenced by means of aggregate functions or their column name must appear in a GROUP BY clause. For example, the following statement uses an aggregate function and a column in the select list and references the column name in the GROUP BY clause:

```
SELECT COUNT(*), Product_ID
FROM Sales_Table
GROUP BY Product_ID;
```

The reason for this is that aggregates return only one value, while a non-GROUP BY column reference can return any number of values.

Aggregates and Date

It is valid to apply AVG, MIN, MAX, or COUNT to a date. It is not valid to specify SUM(date).

Aggregates and Literal Expressions in the Select List

Literal expressions in the select list may optionally appear in the GROUP BY clause. For example, the following statement uses an aggregate function and a literal expression in the select list, and does not use a GROUP BY clause:

```
SELECT COUNT(*),
SUBSTRING( CAST( CURRENT_TIME(0) AS CHAR(14) ) FROM 1 FOR 8 )
FROM Sales_Table;
```

The results of such statements when the table has no rows depends on the type of literal expression.

IF the literal expression …	THEN the result of the literal expression in the query result is …

does not contain a column reference
is a non-deterministic function, such as RANDOM
	the value of the literal expression.
	Functions such as RANDOM are computed in the immediate retrieve step of the request instead of in the aggregation step.
	Here is an example:
	SELECT COUNT(*),
	SUBSTRING(CAST(CURRENT_TIME(0) AS CHAR(14))
	FROM 1 FOR 8)

IF the literal expression …	THEN the result of the literal expression in the query result is …

	FROM Sales_Table;
	Count(*) Substring(Current Time(0) From 1 For 8)
	-------- ---------------------------------------
	0 09:01:43

contains a column reference is a UDF	NULL. Here is an example: SELECT COUNT(*), UDF_CALC(1,2) FROM Sales_Table; Count(*) UDF_CALC(1,2) ----------- ------------- 0 ?

Nesting Aggregates

Aggregate operations cannot be nested. The following aggregate is not valid and returns an error:

```
AVG(MAXIMUM (Salary))
```

Although direct nesting of aggregates is not supported, nested aggregates can be evaluated using a derived table that contains the aggregates to be nested. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

Also, aggregates can be nested in aggregate window functions. The following statement is valid and includes an aggregate SUM function nested in a RANK window function:

```
SELECT region
   ,product
   ,SUM(amount)
   ,RANK() OVER (PARTITION BY region ORDER by SUM (amount))
FROM table;
```

Results of Aggregation on Zero Rows

Aggregation on zero rows behaves as indicated by the following table.

This form of aggregate function …	Returns this result when there are zero rows …
COUNT(*expression*) WHERE …	0
all other forms of *aggregate_operator* (*expression*) WHERE …	Null

This form of aggregate function …	Returns this result when there are zero rows …
aggregate_operator (*expression*) … GROUP BY … *aggregate_operator* (*expression*) … HAVING …	No Record Found

Aggregates and Nulls

Aggregates (with the exception of COUNT(*)) ignore nulls in all computations.

Note:

A UDT column value is null only when you explicitly place a NULL in a column, not when a UDT instance has an attribute that is set to null.

Ignoring nulls can result in apparent nontransitive anomalies. For example, if there are nulls in either column A or column B (or both), then the following expression is virtually always true.

```
   SUM(A) + SUM(B) <> SUM(A+B)
```

The only exception to this is the case in which the values for columns A and B are both null in the same rows, because in those cases the entire row is disregarded in the aggregation. This is a trivial case that does not violate the general rule.

More formally stated, if and only if field A and field B are both null for every occurrence of a null in either field is the above inequality false.

For examples that illustrate this behavior, see "Example: Employees Returned as Nulls" and "Example: Counting Employees Not Yet Assigned to a Department" in Result Type and Attributes. Note that the aggregates are behaving exactly as they should, the results are not mathematically anomalous.

There are several ways to work around this apparent nontransitivity issue if it presents a problem. Either solution provides the same consistent results.

- Always define your numeric columns as NOT NULL DEFAULT 0.
- Use the ZEROIFNULL function within the aggregate function to convert any nulls to zeros for the computation, for example SUM(ZEROIFNULL(x) + ZEROIFNULL(y)), which produces the same result as SUM(ZEROIFNULL(x)) + SUM(ZEROIFNULL(y)).

Aggregate Operations on Floating Point Data

Operations involving floating point numbers are not always associative due to approximation and rounding errors: ((A + B) + C) is not always equal to (A + (B + C)).

Although not readily apparent, the non-associativity of floating point arithmetic can also affect aggregate operations: you can get different results each time you use an aggregate function on a given set of floating point data. When Vantage performs an aggregation, it accumulates individual terms from each AMP involved in the computation and evaluates the terms in order of arrival to produce the final result. Because the order of evaluation can produce slightly different results, and because the order in which individual AMPs finish their part of the work is unpredictable, the results of an aggregate function on the same data on the same system can vary.

Aggregates and LOBs

Aggregates do not operate on CLOB or BLOB data types.

Aggregates and Period Data Types

Aggregates (with the exception of COUNT) do not operate on Period data types.

Aggregates and SELECT AND CONSUME Statements

Aggregates cannot appear in SELECT AND CONSUME statements.

Aggregates and Recursive Queries

Aggregate functions cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify an aggregate function.

Aggregates in WHERE and HAVING Clauses

Aggregates can appear in the following types of clauses:

- The WHERE clause of an ABORT statement to specify an abort condition.

  But an aggregate function cannot appear in the WHERE clause of a SELECT statement.
- A HAVING clause to specify a group condition.

DISTINCT Option

The DISTINCT option specifies that duplicate values are not to be used when an expression is processed.

The following SELECT returns the number of unique job titles in a table.

```
SELECT COUNT(DISTINCT JobTitle) FROM Employee;
```

A query can have multiple aggregate functions that use DISTINCT with the same expression, as shown by the following example.

```
SELECT SUM(DISTINCT x), AVG(DISTINCT x) FROM XTable;
```

A query can also have multiple aggregate functions that use DISTINCT with different expressions, for example:

```
SELECT SUM(DISTINCT x), SUM(DISTINCT y) FROM XYTable;
```

Aggregates and Row Level Security Tables

When a request that includes an aggregate function, such as SUM, COUNT, MAX, MIN or AVG, references a table protected by row level security, the aggregation is based on only the rows that are accessible to the requesting user. In order to apply all rows of the table to the aggregation, the user must have one of the following:

- The required security credentials to access all rows of the table.
- The required OVERRIDE privileges on the security constraints in the table.

Time Series Aggregate Functions Overview

A set of aggregate functions is provided to support time series data (optionally stored in Primary Time Index (PTI) tables). Additionally, some traditional functions support time series as well. To operate on time series data, both time series-specific functions and traditional functions are invoked in a GROUP BY TIME clause.

Traditional Aggregate Functions that Support Time Series

You can use the following aggregate functions on time series data in PTI tables by using the GROUP BY TIME clause and in non-PTI tables by using the GROUP BY TIME clause with the USING TIMECODE option. For more information on these functions, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

- AVERAGE
- COUNT
- KURTOSIS
- MAXIMUM
- MINIMUM
- RANK (ANSI)
- SKEW
- STANDARD DEVIATION OF A POPULATION (STDDEV_POP)
- STANDARD DEVIATION OF A SAMPLE (STDDEV_SAMP)
- SUM
- VARIANCE OF A POPULATION (VAR_POP)
- VARIANCE OF A SAMPLE (VAR_SAMP)

Related Information

- Potential problems associated with floating point values in computations: *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- Window aggregate functions and their Teradata-specific equivalents, see Window Aggregate Functions.
- Aggregate user-defined functions (UDFs), see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.
- Window aggregate UDFs, see "Window Aggregate UDF" in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.
- Row level security, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.
- Time series-specific aggregate functions, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

AVG

Returns the arithmetic average of all values in *value_expression*.

AVG is valid only for numeric data.

Nulls are not included in the result computation.

This function returns the REAL data type.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

For the AVG window function that computes a group, cumulative, or moving average, see Window Aggregate Functions.

AVG Function Syntax

```
{ AVERAGE | AVG | AVE } ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

Exclude duplicates specified by *value_expression* from the computation.

ALL

All values that are not null of *value_expression*, including duplicates, are included in the computation.

value_expression

A literal or column expression for which an average is to be computed.

The value_expression cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

AVERAGE and AVE are Teradata extensions to the ANSI standard.

AVG Usage Notes

Computation of INTEGER or DECIMAL Values

An AVG of a DECIMAL or INTEGER value may overflow if the individual values are very large or if there is a large number of values.

If this occurs, change the AVG call to include a CAST function that converts the DECIMAL or INTEGER values to REAL as shown in the following example:

```
AVG(CAST(value AS REAL) )
```

Casting the values as REAL before averaging causes a slight loss in precision.

The type of the result is REAL in either case, so the only effect of the CAST is to accept a slight loss of precision where a result might not otherwise be available at all.

If x is an integer, AVG does not display a fractional value. A fractional value may be obtained by casting the value as DECIMAL, for example the following CAST to DECIMAL.

```
CAST(AVG(value) AS DECIMAL(9,2))
```

Example: Using the AVG Function

Example: Querying the Sales Table for Average Sales by Region

This example queries the sales table for average sales by region and returns the following results.

```
SELECT Region, AVG(sales)
FROM sales_tbl
GROUP BY Region
ORDER BY Region;
```

```
    Region  Average (sales)
    ------  ---------------
    North          21840.17
    East           55061.32
    Midwest        15535.73
```

For time series examples, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

Related Information

- For more information on potential problems associated with floating point values in computations, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100
- For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- To disable the AVG extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on nulls, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141 and Aggregates and Nulls.
- Aggregate user-defined functions (UDFs), see "Aggregate UDF" in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.
- Window aggregate UDFs, see "Window Aggregate UDF" in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

CORR

Returns the Sample Pearson product moment correlation coefficient of its arguments for all non-null data point pairs.

For the CORR window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Sample Pearson Product Moment Correlation Coefficient

The Sample Pearson product moment correlation coefficient is a measure of the linear association between variables. The boundary on the computed coefficient ranges from -1.00 to +1.00.

Note that high correlation does not imply a causal relationship between the variables.

The following table indicates the meaning of four extreme values for the coefficient of correlation between two variables.

IF the correlation coefficient has this value …	THEN the association between the variables …
-1.00	is perfectly linear, but inverse. As the value for y varies, the value for x varies identically in the opposite direction.
0	does not exist and they are said to be uncorrelated.
+1.00	is perfectly linear. As the value for y varies, the value for x varies identically in the same direction.
NULL	cannot be measured because there are no non-null data point pairs in the data used for the computation.

CORR Function Syntax

```
CORR (value_expression_1, value_expression_2)
```

Syntax Elements

value_expression_1

A numeric expression to be correlated with a second numeric expression.

The expression cannot contain any ordered analytical or aggregate functions.

value_expression_2

A numeric expression to be correlated with a second numeric expression.

The expression cannot contain any ordered analytical or aggregate functions.

Computation

The equation for computing CORR is defined as follows:

This variable …	Represents …
x	*value_expression_2*
y	*value_expression_1*

Division by zero results in NULL rather than an error.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for CORR(y, x) are as follows.

Data Type	Format	Title
REAL	the default format for DECIMAL(7,6)	CORR(y,x)

For an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

CORR Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including CORR, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

Combination With Other Functions

CORR can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For information on ordered analytical functions, see Ordered Analytical Functions.

CORR cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Querying Data from the HomeSales Table

This example uses the data from the HomeSales table.

```
SalesPrice   NbrSold   Area
----------   -------   ---------
    160000       126   358711030
    180000       103   358711030
    200000        82   358711030
    220000        75   358711030
    240000        82   358711030
    260000        40   358711030
    280000        20   358711030
```

Consider the following query.

```
    SELECT CAST (CORR(NbrSold,SalesPrice) AS DECIMAL (6,4))
    FROM HomeSales
    WHERE area = 358711030
    AND SalesPrice Between 160000 AND 280000;

    CORR(NbrSold,SalesPrice)
    ------------------------
                      -.9543
```

The result -.9543 suggests an inverse relationship between the variables. That is, for the area and sales price range specified in the query, the value for NbrSold increases as sales price decreases and decreases as sales price increases.

Related Information

- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

COUNT

Returns a column value that is the total number of qualified rows in *value_expression*.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

COUNT Function Syntax

```
COUNT ( { [ DISTINCT | ALL ] value_expression | * } )
```

Syntax Elements

DISTINCT

Exclude duplicates specified by *value_expression* from the computation.

The expression cannot contain any ordered analytical or aggregate functions.

ALL

All values of *value_expression* that are not null, including duplicates, are included in the computation.

value_expression

A literal or column expression for which the number of values is to be counted.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

*****

Counts all rows in the group of rows on which COUNT operates.

Result Type and Attributes

The following table lists the data type and format for the result of COUNT.

Mode	Data Type and Format
ANSI	MaxDecimal is general field 13 in the DBS Control utility. If MaxDecimal in DBSControl is… • 0 or 15, then the result type is DECIMAL(15,0) and the format is -(15)9. • 18, then the result type is DECIMAL(18,0) and the format is -(18)9. • 38, then the result type is DECIMAL(38,0) and the format is -(38)9.
Teradata	INTEGER and the format is the default format for INTEGER.
COUNT	The default value for the DBSControl General Field(80), COUNT_mode, is 0. The default is compatibility mode, which disables all extensions that impact external applications.

BIGINT and NUMBER modes impact COUNT performance:

- Type promotion may entail computing expressions using a different type if the mode is changed. This occurs when the result of the COUNT (*) based expression is materialized as a BIGINT/NUMBER type, and later used as a subexpression for computing another expression. The performance overhead is the same as that incurred when casting COUNT (*) as BIGINT/NUMBER.
- Since the data type of COUNT (*) changes if the mode is changed, queries that made assumptions on format, title, and data type must be aware of the change.

If the result of COUNT overflows and reports an error, you can cast the result to another data type, as illustrated by the following example.

```
    SELECT CAST(COUNT(*) AS BIGINT)
    FROM BIGTABLE;
```

A similar example is provided for COUNT and rank window functions:

```
SELECT CAST(COUNT(*) over([PARTITION/ORDER BY]) AS BIGINT)
FROM BIGTABLE;
SELECT CAST(rank over([PARTITION/ORDER BY]) AS BIGINT)
FROM BIGTABLE;
```

Note:

The CAST is required only for default or compatibility mode. If value of 1 or 2 is specified for NUMBER or BIGINT mode of computing COUNT, then the CAST is not required.

The following table lists the default title for the result of COUNT.

Operation	Title
COUNT(x)	Count(x)
COUNT(*)	Count(*)

COUNT Specification in Aggregate Join Index

You can specify COUNT, COUNT cast to FLOAT OR DECIMAL(38,0), BIGINT, or NUMBER for a COUNT aggregate function in a join index. The following illustrates a SHOW JOIN INDEX that accommodates data type casts to BIGINT:

```
CREATE JOIN INDEX TEST.j1 ,NO FALLBACK ,CHECKSUM = DEFAULT AS
SELECT COUNT (*)(BIGINT, NAMED a ),TEST.t1.a1
FROM TEST.t1
GROUP BY TEST.t1.a1
PRIMARY INDEX ( a1 );
```

Usage Notes

This syntax ...	Counts the total number of rows ...
COUNT(*value_expression*)	in the group for which *value_expression* is not null.
COUNT (DISTINCT *value_expression*)	in the group for which *value_expression* is unique and not null.

This syntax …	Counts the total number of rows …
COUNT(*)	in the group of rows on which COUNT operates.

COUNT is valid for any data type.

Examples: Using the COUNT Function

Example: Reporting the Number of Employees in Each Department

COUNT(*) reports the number of employees in each department because the GROUP BY clause groups results by department number.

```
SELECT DeptNo, COUNT(*) FROM Employee
GROUP BY DeptNo
ORDER BY DeptNo;
```

Without the GROUP BY clause, only the total number of employees represented in the Employee table is reported:

```
SELECT COUNT(*) FROM Employee;
```

Note that without the GROUP BY clause, the select list cannot include the DeptNo column because it returns any number of values and COUNT(*) returns only one value.

Example: Employees Returned as Nulls

If any employees have been inserted but not yet assigned to a department, the return includes them as nulls in the DeptNo column.

```
SELECT DeptNo, COUNT(*) FROM Employee
GROUP BY DeptNo
ORDER BY DeptNo;
```

Assuming that two new employees are unassigned, the results table is:

```
DeptNo   Count(*)
------   --------
?               2
   100          4
   300          3
   500          7
   600          4
   700          3
```

Example: Counting Employees Not Yet Assigned to a Department

If you ran the report in Example: Reporting the Number of Employees in Each Department using SELECT... COUNT … without grouping the results by department number, the results table would have only registered non-null occurrences of DeptNo and would not have included the two employees not yet assigned to a department(nulls). The counts differ (23 in Example: Reporting the Number of Employees in Each Department as opposed to 21 using the statement documented in this example).

Recall that in addition to the 21 employees in the Employee table who are assigned to a department, there are two new employees who are not yet assigned to a department (the row for each new employee has a null department number).

```
SELECT COUNT(deptno) FROM employee ;
```

The result of this SELECT is that COUNT returns a total of the non-null occurrences of department number.

Because aggregate functions ignore nulls, the two new employees are not reflected in the figure.

```
Count(DeptNo)
--------------
           21
```

Example: Using COUNT to Find the Number of Employees by Gender

This example uses COUNT to provide the number of male employees in the Employee table of the database.

```
SELECT COUNT(sex)
FROM Employee
WHERE sex = 'M' ;
```

The result is as follows.

```
Count(Sex)
----------
        12
```

Example: Providing a Total of the Rows with Non-Null Department Numbers

In this example COUNT provides, for each department, a total of the rows that have non-null department numbers.

```
SELECT deptno, COUNT(deptno)
FROM employee
```

```
GROUP BY deptno
ORDER BY deptno ;
```

Notice once again that the two new employees are not included in the count.

```
DeptNo   Count(DeptNo)
------   -------------
   100               4
   300               3
   500               7
   600               4
   700               3
```

Example: Returning the Number of Employees by Department

To get the number of employees by department, use COUNT(*) with GROUP BY and ORDER BY clauses.

```
SELECT deptno, COUNT(*)
FROM employee
GROUP BY deptno
ORDER BY deptno ;
```

In this case, the nulls are included, indicated by QUESTION MARK.

```
DeptNo   Count(*)
------   --------
?               2
   100          4
   300          3
   500          7
   600          4
   700          3
```

Example: Determining the Number of Departments in the Employee Table

To determine the number of departments in the Employee table, use COUNT (DISTINCT) as illustrated in the following SELECT COUNT.

```
SELECT COUNT (DISTINCT DeptNo)
FROM Employee ;
```

The system responds with the following report.

```
Count(Distinct(DeptNo))
-----------------------
                      5
```

For time series examples, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

Related Information

- For COUNT functions that return the group, cumulative, or moving count, see Window Aggregate Functions.
- With the exception of COUNT(*), the computation does not include nulls. For more information, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141 and Aggregates and Nulls.
- For information on data type default formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For information on the COUNT_mode field, see *Teradata Vantage™ - Database Utilities*, B035-1102.

COVAR_POP

Returns the population covariance of its arguments for all non-null data-point pairs.

For the COVAR_POP window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Covariance

Covariance measures whether or not two random variables vary in the same way. It is the average of the products of deviations for each non-null data point pair.

Note that high covariance does not imply a causal relationship between the variables.

Computation

When there are no non-null data point pairs in the data used for the computation, then COVAR_POP returns NULL.

Division by zero results in NULL rather than an error.

COVAR_POP Function Syntax

```
COVAR_POP (value_expression_1, value_expression_2)
```

Syntax Elements

value_expression_1

A numeric expression to be paired with a second numeric expression to determine their covariance.

The expression cannot contain any ordered analytical or aggregate functions.

value_expression_2

A numeric expression to be paired with a second numeric expression to determine their covariance.

The expression cannot contain any ordered analytical or aggregate functions.

Result Format Types

The data type, format, and title for COVAR_POP are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as x.
- If the operand is a UDT, the format is the format for the data type to which the UDT is implicitly cast.

For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for COVAR_POP are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as x.
- If the operand is a UDT, the format is the format for the data type to which the UDT is implicitly cast.

For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

COVAR_POP Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including COVAR_POP, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143 .

Combination With Other Functions

COVAR_POP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause.

COVAR_POP cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

COVAR_SAMP

Returns the sample covariance of its arguments for all non-null data point pairs.

For the COVAR_SAMP window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Covariance

Covariance measures whether or not two random variables vary in the same way. It is the sum of the products of deviations for each non-null data point pair.

Note that high covariance does not imply a causal relationship between the variables.

Computation

When there are no non-null data point pairs in the data used for the computation, then COVAR_SAMP returns NULL.

Division by zero results in NULL rather than an error.

COVAR_SAMP Function Syntax

```
COVAR_SAMP (value_expression_1, value_expression_2)
```

Syntax Elements

value_expression_1

A numeric expression to be paired with a second numeric expression to determine their covariance.

The expression cannot contain any ordered analytical or aggregate functions.

value_expression_2

A numeric expression to be paired with a second numeric expression to determine their covariance.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for COVAR_SAMP(y, x) are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as x.
- If the operand is a UDT, the format is the format for the data type to which the UDT is implicitly cast.

For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

COVAR_SAMP Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE

- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including COVAR_SAMP, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143 .

Combination with Other Functions

COVAR_SAMP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

COVAR_SAMP cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Using the SELECT statement to Return the Sample Covariance of Weight and Height

This example is based on the following regrtbl data. Nulls are indicated by the QUESTION MARK character.

c1	height	weight
1	60	84
2	62	95
3	64	140
4	66	155
5	68	119
6	70	175
7	72	145
8	74	197
9	76	150
10	76	?
11	?	150

c1	height	weight
12	?	?

The following SELECT statement returns the sample covariance of weight and height where neither weight nor height is null.

```
SELECT COVAR_SAMP(weight,height)
FROM regrtbl;

Covar_Samp(weight,height)
-------------------------
                      150
```

GROUPING

Returns a value that indicates whether a specified column in the result row was excluded from the grouping set of a GROUP BY clause.

IF the value of the specified column in the result row is …	THEN GROUPING returns …
a NULL generated when the extended grouping specification aggregated over the column and excluded it from the particular grouping	1
anything else	0

GROUPING Function Syntax

```
GROUPING (expression)
```

Syntax Elements

expression

A column in the result row that might have been excluded from a grouped query containing CUBE, ROLLUP, or GROUPING SET.

The argument must be an item of a GROUP BY clause.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for GROUPING(x) are as follows.

Data Type	Format	Title
INTEGER	Default format of the INTEGER data type	Grouping(x)

Usage Notes

A null in the result row of a grouped query containing CUBE, ROLLUP, or GROUPING SET can mean one of the following:

- The actual data for the column is null.
- The extended grouping specification aggregated over the column and excluded it from the particular grouping. A null in this case really represents all values for this column.

Use GROUPING to distinguish between rows with nulls in actual data from rows with nulls generated from grouping sets.

Example: Viewing Sales Summaries by County and by City

Suppose you have the following data in the sales_view table.

PID	Cost	Sale	Margin	State	County	City
1	38350	50150	11800	CA	Los Angeles	Long Beach
1	63375	82875	19500	CA	San Diego	San Diego
1	46800	61200	14400	CA	Los Angeles	Avalon
2	40625	53125	12500	CA	Los Angeles	Long Beach

To look at sales summaries by county and by city, use the following SELECT statement:

```
SELECT county, city, sum(margin)
FROM sale_view
GROUP BY GROUPING SETS ((county),(city));
```

The query reports the following data:

```
County       City        Sum(margin)
-----------  ----------  -----------
Los Angeles  ?                 38700
San Diego    ?                 19500
```

```
?            Long Beach        24300
?            San Diego         19500
?            Avalon            14400
```

Notice that in this example, a null represents all values for a column because the column was excluded from the grouping set represented.

To distinguish between rows with nulls in actual data from rows with nulls generated from grouping sets, use the GROUPING function:

```
SELECT county, city, sum(margin),
       GROUPING(county) AS County_Grouping,
       GROUPING(city) AS City_Grouping
FROM sale_view
GROUP BY GROUPING SETS ((county),(city));
```

The results are:

```
County      City       Sum(margin) County_Grouping City_Grouping
----------- ---------- ----------- --------------- -------------
Los Angeles ?                38700               0             1
San Diego   ?                19500               0             1
?           Long Beach       24300               1             0
?           San Diego        19500               1             0
?           Avalon           14400               1             0
```

You can also use GROUPING to replace the nulls that appear in a result row because the extended grouping specification aggregated over a column and excluded it from the particular grouping.
For example:

```
SELECT CASE
         WHEN GROUPING(county) = 1
         THEN '-All Counties-'
         ELSE county
       END AS County,
       CASE
         WHEN GROUPING(city) = 1
         THEN '-All Cities-'
         ELSE city
       END AS City,
       SUM(margin)
FROM sale_view
GROUP BY GROUPING SETS (county,city);
```

The query reports the following data:

```
County          City          Sum(margin)
--------------  ------------  -----------
Los Angeles     -All Cities-        38700
San Diego       -All Cities-        19500
-All Counties-  Long Beach          24300
-All Counties-  San Diego           19500
-All Counties-  Avalon              14400
```

Related Information

- For more information on GROUP BY, GROUPING SETS, ROLLUP, and CUBE, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- For information on the default format of data types, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

KURTOSIS

Returns the kurtosis of the distribution of *value_expression*.

This function returns the REAL data type.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

Kurtosis

Kurtosis is the fourth moment of the distribution of the standardized (z) values. It is a measure of the outlier (rare, extreme observation) character of the distribution as compared with the normal, Gaussian distribution.

The normal distribution has a kurtosis of 0.

Positive kurtosis indicates that the distribution is more outlier-prone than the normal distribution, while negative kurtosis indicates that the distribution is less outlier-prone than the normal distribution.

Computation

The equation for computing KURTOSIS is defined as follows:

$$\text{Kurtosis} = \left(\frac{(\text{COUNT}(x))(\text{COUNT}(x)+1)}{(\text{COUNT}(x)-1)(\text{COUNT}(x)-2)(\text{COUNT}(x)-3)}\right)\left(\text{SUM}\left(\frac{x-\text{AVG}(x)}{\text{STDEV\_SAMP}(x)}{**}4\right)\right)-\left(\frac{(3)((\text{COUNT}(x)-1)({**}2))}{(\text{COUNT}(x)-2)(\text{COUNT}(x)-3)}\right)$$

where:

This variable …	Represents …
x	*value_expression*

KURTOSIS Function Syntax

```
KURTOSIS ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

Exclude duplicates specified by *value_expression* from the computation.

ALL

All values of *value_expression* that are not null, including duplicates, are included in the computation.

value_expression

A literal or column expression for which the kurtosis of the distribution of its values is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

KURTOSIS Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including KURTOSIS, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the

DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143 .

Conditions That Produce a NULL Return Value

The following conditions produce a null return value:

- Fewer than four non-null data points in the data used for the computation
- STDDEV_SAMP(x) = 0
- Division by zero

MAXIMUM

Returns a column value that is the maximum value for *value_expression*.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

For the MAXIMUM window function that computes a group, cumulative, or moving maximum value, see Window Aggregate Functions.

MAXIMUM Function Syntax

```
{ MAXIMUM | MAX } ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

Exclude duplicates specified by *value_expression* from the computation.

Duplicate and values that are not null specified by *value_expression* are eliminated from the maximum value computation for the group.

ALL

All values that are not null specified by *value_expression*, including duplicates, are included in the maximum value computation for the group. This is the default.

value_expression

A literal or column expression for which the maximum value is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

MAXIMUM is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The following table lists the default attributes for the result of MAX(x).

Attribute	Value
Data Type	If operand x is not a UDT, the result data type is the data type of operand x. If operand x is a UDT, the result data type is the data type to which the UDT is implicitly cast.
Format	If operand x is not a UDT, the result data type is the data type of operand x. If operand x is a UDT, the result data type is the data type to which the UDT is implicitly cast.
Title	Maximum(x)

Usage Notes

MAX is valid for character data as well as numeric data. When used with a character expression, MAX returns the highest sort order.

Nulls are not included in the result computation. For more information, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141 and Aggregates and Nulls.

If *value_expression* is a column expression, the column must refer to at least one column in the table from which data is selected.

The *value_expression* must not specify a column reference to a view column that is derived from a function.

Support for UDTs

By default, Vantage performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and any of the following predefined types:

- Numeric
- Character
- Byte
- DATE
- TIME or TIMESTAMP
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including MAX, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143 .

Examples: Using the MAXIMUM Function

Example: CHARACTER Data

The following SELECT returns the immediately following result.

```
SELECT MAX(Name)
FROM Employee;

Maximum(Name)
-------------
Zorn J
```

Example: Column Expressions

You want to know which item in your warehouse stock has the maximum cost of sales.

```
SELECT MAX(CostOfSales) AS m, ProdID
FROM Inventory
GROUP BY ProdID
ORDER BY m DESC;

Maximum(CostOfSales)  ProdID
--------------------  ------
                1295    3815
                 975    4400
                 950    4120
```

For time series examples, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

MINIMUM

Returns a column value that is the minimum value for *value_expression*.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

For the MINIMUM window function that computes a group, cumulative, or moving minimum value, see Window Aggregate Functions.

MINIMUM Function Syntax

```
{ MINIMUM | MIN } ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

Exclude duplicates specified by *value_expression* from the computation.

Duplicate and values that are not null specified by *value_expression* are eliminated from the minimum value computation for the group.

ALL

All values that are not null specified by *value_expression*, including duplicates, are included in the minimum value computation for the group. This is the default.

value_expression

A literal or column expression for which the minimum value is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

MINIMUM is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The following table lists the default attributes for the result of MIN(x).

Attribute	Value
Data type	If operand x is not a UDT, the result data type is the data type of operand x. If operand x is a UDT, the result data type is the data type to which the UDT is implicitly cast.
Title	Minimum(x)
Format	If operand x is not a UDT, the result format is the format of operand x.

Attribute	Value
	If operand x is a UDT, the result format is the format of the data type to which the UDT is implicitly cast.

Usage Notes

MINIMUM is valid for character data as well as numeric data. MINIMUM returns the lowest sort order of a character expression.

The computation does not include nulls. For more information, see “Manipulating Nulls” in *Teradata Vantage™ - SQL Fundamentals*, B035-1141 and Aggregates and Nulls.

If *value_expression* specifies a column expression, the expression must refer to at least one column in the table from which data is selected.

If *value_expression* specifies a column reference, the column must not be a view column that is derived from a function.

Support for UDTs

By default, Vantage performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and any of the following predefined types:

- Numeric
- Character
- Byte
- DATE
- TIME or TIMESTAMP
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including MIN, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143 .

Examples: Using the MINIMUM Function

Example: MINIMUM Used with CHARACTER Data

The following SELECT returns the immediately following result.

```
SELECT MINIMUM(Name)
FROM Employee;

Minimum(Name)
-------------
Aarons A
```

Example: JIT Inventory

Your manufacturing shop has recently changed vendors and you know that you have no quantity of parts from that vendor that exceeds 20 items for the ProdID. You need to know how many of your other inventory items are low enough that you need to schedule a new shipment, where “low enough” is defined as fewer than 30 items in the QUANTITY column for the part.

```
SELECT ProdID, MINIMUM(QUANTITY)
FROM Inventory
WHERE QUANTITY BETWEEN 20 AND 30
GROUP BY ProdID
ORDER BY ProdID;
```

The report is as follows:

```
     ProdID  Minimum(Quantity)
-----------  -----------------
       1124                 24
       1355                 21
       3215                 25
       4391                 22
```

For time series examples, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

PIVOT

PIVOT is a relational operator for transforming rows into columns. The function is useful for reporting purposes, as it allows you to aggregate and rotate data to create easy-to-read tables. You can perform PIVOT aggregation on PIVOT column results by using the WITH clause.

Specify the PIVOT operator in the FROM clause of the SELECT statement. There are no restrictions on other clauses that can be specified with the SELECT query that include PIVOT operators.

PIVOT Function Syntax

```
PIVOT ( pivot_spec )
  [ WITH with_spec [,...] ]
  [AS] derived_table_name [ ( cname [,...] ) ]
```

Syntax Elements

pivot_spec

```
aggr_fn_spec [,...] FOR for_spec
```

with_spec

```
aggr_fn ( { cname [,...] | * } ) [AS] aggr_alias
```

derived_table_name

The table name specified for the resultant pivoted table.

cname

A column name.

aggr_fn_spec

```
aggr_fn ( cname ) [ [AS] pvt_aggr_alias ]
```

for_spec

```
{ cname IN ( expr_spec_1 [,...] ) |
  ( cname [,...] ) IN ( expr_spec_2 [,...] ) |
  cname IN ( subquery )
}
```

aggr_fn

An aggregate function that supports a single argument.

*****

Option to include all the Pivot columns without specifying columns explicitly.

aggr_alias

Name of the aggregate result column.

pvt_aggr_alias

An alias name specified for the Aggregate function.

expr_spec_1

```
expr [ [AS] expr_alias_name ]
```

expr_spec_2

```
( expr [,...] ) [ [AS] expr_alias_name ]
```

expr

An expression or a column value.

expr_alias_name

An alias name specified for the values/expressions specified in the IN list.

Usage Notes

Note:

For the PIVOT operation, column names within the Aggregate functions are referred to as measure columns, and column names in the FOR clause are referred to as pivot columns.

As indicated in the syntax, specify at least one Aggregate function with the PIVOT operator.

Columns with CLOB, BLOB, UDT, XML, or JSON data types are not allowed with the PIVOT operator.

Column names are not allowed within the IN-list. Only values or expressions (arithmetic expressions such as MOD or ABS, or string Manipulation expressions such as LENGTH, REVERSE) are allowed.

Measure columns and pivot columns of the PIVOT operator are not allowed in the assign list of the SELECT statement.

If *n* number of Aggregate functions are specified where *n* is greater than 1, then the alias name must be specified for at least (*n*-1) aggregate functions.

The *cname* specified in the derived_table_name takes precedence over the alias names derived from the IN-list.

If the alias names are not specified for the column values listed in the IN clause, the database processing encloses the column values into double quotes and converts these string literals to alias names using the default format. The alias names are used as column names of the pivoted table.

If the length of the alias name derived from a column value exceeds the alias name limit of 128 characters (if EON feature is enabled) or 30 characters (if EON is not enabled), the alias name is truncated.

If the IN-list contains case-specific values such as ‘abc’ & ‘ABC’, the values are treated the same and an error occurs.

PIVOT supports the UNPIVOT or TD_UNPIVOT functions as a query source for the PIVOT operator.

The PIVOT/UNPIVOT operator uses a single dimensional way of converting rows to columns, or columns to rows. You can swap both rows and columns within a single query (for example, using UNPIVOT as source to PIVOT). This provides flexibility when using the two-dimensional method of interchanging data in a table.

Using the DT column list for UNPIVOT as a query source to PIVOT is optional.

If the WITH clause is specified in the PIVOT query:

- Specifiying at least one aggregate function with the WITH operator is mandatory.
- SUM, AVG, MIN, and MAX aggregate functions are supported.
- The *cname* specified in the derived_table_name takes precedence over the alias names derived for the aggregated result columns.
- DISTINCT keyword is not supported with aggregate column.
- Column list is not allowed if an asterisk (*) is specified.
- Aggregating a column list or * may produce meaningless results if the values aggregated are not related. For example, if some pivot columns are for SUM and some are for an AVG, WITH SUM(*) is not a meaningful value.
- Column names mentioned in the aggregate function should be PIVOT columns or subset of PIVOT columns.

To avoid the overhead of issuing a separate query to generate values for input to the PIVOT IN-list clause as hard coded constants, you can issue the query as a subquery in the PIVOT IN-list. If a PIVOT query has a subquery in the IN-list:

- Alias names are not allowed in the IN-list.
- Alias names in the PIVOT derived table are not allowed.
- The SELECT list of the subquery must contain only one column reference.
- The subquery must return at least one row.
- The results returned by the subquery cannot exceed 32KB, and the row count must be less than or equal to 16.
- SET operations are not allowed on a PIVOT query that has a subquery in the IN-list.
- Columns generated by an IN-list subquery cannot be explicitly used in the SELECT.
- You cannot use a subquery in a PIVOT IN-list with DDL statements or multistatement requests.
- A PIVOT query cannot include both a WITH clause and a subquery in the IN-list.

For examples of wide tables, see Pivot Examples.

Examples

Example: Alias Names Contained in the IN List

This example uses the *star1* table, with the following definition and contents:

```
CREATE TABLE star1(country VARCHAR(20),state VARCHAR(10), yr INTEGER,qtr
VARCHAR(3),sales INTEGER,cogs INTEGER);

SELECT * FROM star1;
country  state           yr  qtr        sales         cogs
-------  -----  -----------  ---  -----------  -----------
USA      CA            2001  Q1            30           15
Canada   ON            2001  Q2            10            0
Canada   BC            2001  Q3            10            0
USA      NY            2001  Q1            45           25
USA      CA            2001  Q2            50           20
```

In this example, the IN list contains alias names. The alias names are concatenated with the alias names specified by the aggregate functions to build the column names of the output pivoted table.

```
SELECT *
FROM star1 PIVOT (
                   SUM(sales) as ss1, SUM(cogs) as sc FOR
qtr

                                       IN (‘Q1’ AS
Quarter1,

                                           ‘Q2’ AS Quarter2,
                                           ‘Q3’ AS Quarter3)
                                          )Tmp;
```

The output is re-written as an equivalent SELECT query using CASE statements:

```
SELECT *  FROM  (SELECT country ,state ,yr ,
SUM(CASE WHEN qtr =  'Q1' THEN sales ELSE NULL END )AS  Quarter1_ss1,
SUM(CASE WHEN qtr =  'Q1' THEN (cogs) ELSE NULL END )AS Quarter1_sc,
SUM(CASE WHEN qtr =  'Q2' THEN (sales) ELSE NULL END)AS Quarter2_ss1,
SUM(CASE WHEN qtr =  'Q2' THEN (cogs) ELSE NULL  END)AS Quarter2_sc,
SUM(CASE WHEN qtr =  'Q3' THEN (sales) ELSE NULL END)AS Quarter3_ss1,
SUM(CASE WHEN qtr =  'Q3' THEN (cogs) ELSE NULL END)AS Quarter3_sc
FROM star1 GROUP BY country ,state ,yr ) Tmp ;
```

Output pivoted table:

```
country state yr   Quarter1_ss1 Quarter1_sc Quarter2_ss1 Quarter2_sc
Quarter3_ssl Quarter3_sc
------- ---- ----  ------------ ----------- ------------ -----------
------------ -----------
USA     CA   2001             30          15
50          20             ?           ?
USA     NY   2001
45          25             ?           ?             ?           ?
Canada   ON  2001               ?           ?
10           0             ?           ?
Canada   BC  2001               ?           ?             ?           ?
10           0
```

Example: Naming Columns with the <column_value_list> Values

In this example, the SELECT statement does not specify the names to use for columns explicitly. The names of the columns are built internally by adding the aggregated column name to the <column_value_list> values.

```
SELECT *
FROM star1 PIVOT (SUM(sales) AS ss1, SUM(cogs) AS sc FOR (yr, qtr)
                                                  IN ((2001, ‘Q1’),
                                                      (2001, ‘Q2’),
                                                      (2001, ‘Q3’))
                                                     )Tmp;
```

This is re-written as an equivalent SELECT query that uses CASE statements:

```
SELECT *  FROM  (SELECT country ,state ,
SUM(CASE WHEN yr =  2001 AND  qtr =  'Q1' THEN sales ELSE NULL END)
AS "2001_'Q1'_ss1" ,
SUM(CASE WHEN yr =  2001 AND  qtr =  'Q1' THEN cogs ELSE NULL END)
AS "2001_'Q1'_sc",
SUM(CASE WHEN yr =  2001 AND  qtr =  'Q2' THEN sales ELSE NULL END)
AS "2001_'Q2'_ss1" ,
SUM(CASE WHEN yr =  2001 AND  qtr =  'Q2' THEN cogs ELSE NULL END)
AS "2001_'Q2'_sc",
SUM(CASE WHEN yr =  2001 AND  qtr =  'Q3' THEN sales ELSE NULL END)
AS "2001_'Q3'_ss1",
SUM(CASE WHEN yr =  2001 AND  qtr =  'Q3' THEN cogs ELSE NULL END)
AS "2001_'Q3'_sc"
FROM star1 GROUP BY country ,state ) Tmp ;
```

Output pivoted table:

```
country state 2001_'Q1'_ss1 2001'_'Q1'_sc 2001_'Q2'_ss1 2001_'Q2'_sc
2001_'Q3'_ssl 2001_'Q3'_sc
------- ----  ------------- ------------- ------------- ------------
------------- ------------
USA     CA                30            15
50           20             ?             ?
USA     NY
45            25              ?             ?             ?             ?
Canada  ON                  ?             ?
10            0             ?             ?
Canada  BC                  ?             ?             ?            ?
10             0
```

Example: Pivot Operation on View

The following example of a view as a PIVOT source.

Assume a view, *v1*, is defined on the table *s1*:

```
CREATE TABLE s1(yr INTEGER, mon VARCHAR(4), sales INTEGER);

sel * from s1;

sel * from s1;

 *** Query completed. 8 rows found. 3 columns returned.
 *** Total elapsed time was 1 second.

         yr  mon         sales
-----------  ----  -----------
       2001  jan           100
       2003  jan           300
       2002  jan           150
       2001  feb           110
       2003  feb           310
       2002  feb           200
       2001  mar           120
       2002  mar           250

CREATE VIEW V1 AS select yr,sales  from s1;

 *** View has been created.
 *** Total elapsed time was 1 second.


sel * from v1;

select * from v1;

 *** Query completed. 8 rows found. 2 columns returned.
 *** Total elapsed time was 1 second.

         yr        sales
-----------  -----------
       2002          150
       2003          300
       2002          200
       2003          310
       2002          250
       2001          100
       2001          110
       2001          120
```

The following query generates sales report with respect to each year on view V1:

```
SELECT *
FROM v1 PIVOT (SUM(sales) FOR yr IN (2001,2002,2003)) tmp;


 *** Query completed. One row found. 3 columns returned.
 *** Total elapsed time was 1 second.

       2001         2002         2003
-----------  -----------  -----------
        330          600          610
```

Example: Table Source Using the WITH Clause

The following is an example of a table using the WITH clause as a source to the pivot query.

```
SELECT *
FROM (with temp
as (select * from s1) select * from temp)dt PIVOT (SUM(sales) FOR mon IN
('Jan','Feb', 'Mar'))tmp;

 *** Query completed. 3 rows found. 4 columns returned.
 *** Total elapsed time was 1 second.
         yr          Jan          Feb          Mar
      -----       ------       ------      -------
       2001          100          110          120
       2002          150          200          250
       2003          300          310            ?
```

Example: SELECT Query with the WHERE Condition

The following is an example of using a SELECT query with the WHERE condition:

```
SELECT *
FROM s1 PIVOT (SUM(sales) FOR mon IN (‘Jan’ as Jan, ‘Feb’ as Feb, ‘Mar’ as
Mar))tmp where Jan=100;

 *** Query completed. 1 rows found. 4 columns returned.
 *** Total elapsed time was 1 second.
      Yr          Jan          Feb          Mar
  ------      --------     --------     --------
    2001          100          110          120
```

Example: CREATE TABLE AS Statement Contains Special Characters

In this example, the CREATE TABLE AS statement contains special characters in the pivot query IN list.

```
CREATE TABLE t1 AS
(SELECT *
FROM s1 PIVOT (SUM(sales) FOR mon IN (U&"#FAD7" UESCAPE '#')) tmp ) WITH DATA;

*** Failure 4306 Invalid PIVOT query: Unsupported In-List Values/Expressions.
```

```
                Statement# 1, Info =0
 *** Total elapsed time was 1 second.
```

Example: The PIVOT Query Response in Different Response Modes

Assume a table t1 is defined as:

```
CREATE TABLE t1(yr INTEGER,mon VARCHAR(3),sales INTEGER);

Assume that following insert statements
INSERT t1 VALUES(2003,'Jan',300);
INSERT t1 VALUES(2001,'Jan',100);
INSERT t1 VALUES(2003,'Feb',310);
INSERT t1 VALUES(2001,'Feb',110);
INSERT t1 VALUES(2002,'Jan',150);
INSERT t1 VALUES(2001,'Mar',120);
INSERT t1 VALUES(2002,'Feb',200);
INSERT t1 VALUES(2002,'Mar',250);
INSERT t1 VALUES(2003,'Mar',1000);
```

Assuming that the PIVOT query is submitted for execution, the output returns as different responses modes.

For a PIVOT query:

```
SELECT * FROM t1 PIVOT(SUM(sales) FOR mon IN ('Jan','Feb','Mar')) tmp;
```

For a PIVOT query re-written as a SELECT statement using CASE expressions:

```
SELECT yr,SUM(case when mon='Jan' then sales end) AS "Jan",
SUM(case when mon='Feb' then sales end) AS "Feb",
SUM(case when mon='Mar' then sales end) AS "Mar"
FROM t1 GROUP BY yr;

.field mode

 *** Query completed.  3 rows found.

         yr          Jan          Feb          Mar
-----------  -----------  -----------  -----------
       2001          100          110          120
       2003          300          310         1000
       2002          150          200          250
```

```
.multipartrecord mode

 *** Query completed.  3 rows found.

        yr         Jan         Feb         Mar
----------  ----------  ----------  ----------
      2001         100         110         120
      2003         300         310        1000
      2002         150         200         250


.record mode

 *** Query completed.  3 rows found.

         yr          Jan          Feb          Mar
-----------  -----------  -----------  -----------
       2001          100          110          120
       2003          300          310         1000
       2002          150          200          250

.indicator mode

 *** Query completed.  3 rows found.

         yr          Jan          Feb          Mar
-----------  -----------  -----------  -----------
       2001          100          110          120
       2003          300          310         1000
       2002          150          200          250
```

Example: Pivot Query Truncates the Alias Name

For the first part of this example, the EnableEON dbscontrol flag is set to false, so the column name limit defaults to 30 characters.

Assume the table *t1* is defined as:

```
CREATE TABLE t1(yr INTEGER, mon VARCHAR(41), sales INTEGER);
```

Also assume that the table *t1* contains the following row:

```
SELECT * FROM  t1;
yr          mon                                         sales
----   ----------------------------------          -----
2001      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa          200
```

The row contains 35 characters for the column ‘mon’.

The following pivot query results truncate the ‘mon’ column value from 35 characters to 30 characters:

```
SELECT * FROM t1 PIVOT(SUM(sales) FOR mon IN
('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'))tmp;

YR       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
----  -------------------------------
2001     200
```

Now, assume that the EnableEON dbscontrol flag is set to true, so the column name limit defaults to 128 characters.

Also assume that table *t2* is defined as follows:

```
CREATE TABLE t2(yr INTEGER, mon VARCHAR(131), sales INTEGER);
```

Assume that the table *t2* contains the following row:

```
SELECT mon FROM t2;
mon
--------------------------------------------------------------------------------
-----------------------------------------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

The row contains 130 characters for the column ‘mon’.

The following pivot query truncates the ‘mon’ column value from 130 characters to 128 characters:

```
SELECT * FROM t2 PIVOT(SUM(sales) FOR mon IN
('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa')) tmp;

YR
----
2001
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
------------------------------------------------------------------------------
--------------------------------------------
200
```

Example: Using TD_UNPIVOT or UNPIVOT as a Source to PIVOT

PIVOT supports UNPIVOT query or the TD_UNPIVOT function as a source for the PIVOT operator.

PIVOT/ UNPIVOT uses a single dimensional method to interchange data, such as converting rows to columns, or columns to rows, based on some aggregation on a column data.

Swap rows and columns within a single query by giving UNPIVOT query as a source to PIVOT. This provides flexibility for a two-dimensional way of interchanging data in a table based on some aggregation on a column.

Note:

To change data with a two-dimensional method, aggregate data on a column, and then interchange the rows and columns twice. In this case, swap rows and columns based on some aggregation on a column data. The table rotates twice by some aggregation, but might not return the actual table rows. It could introduce new rows where data is missing, or eliminate rows if data is aggregated in the process.

Two-dimensional uses PIVOT as source to the UNPIVOT query, or UNPIVOT as a source to a PIVOT query. Using PIVOT as source to an UNPIVOT query is complex when writing the SQL, whereas using UNPIVOT as a source to PIVOT query is easier.

First, create a table with the following data:

```
CREATE TABLE t1 (place CHAR(5), sales1 INTEGER, sales2 INTEGER,
                 sales3 INTEGER, sales4 INTEGER, sales5 INTEGER)
PRIMARY INDEX ( place );

place     sales1    sales2    sales3    sales4    sales5
-----  ---------  --------  --------  --------  --------
Hyd          110       100      1000      1100       500
Che          120       200      2000      1200       600
Kol          150       500      5000      1500       900
Mee          140       400      4000      1400       800
Pun          130       300      3000      1300       700
```

To get the SUM of sales for each place, swap the sales and place using the following query:

```
SELECT * from (SELECT * from t1
            UNPIVOT(saleval
                    for sales in (sales1, sales2, sales3,
                                  sales4, sales5))dt1)dt2
                 PIVOT(SUM(saleval)
                       for place in ('hyd','Che','pun',
                                     'mee','kol'))dt3;
```

The results for using UNPIVOT as the source:

```
sales        Hyd       Che       Pun       Mee       Kol
-----   --------  --------  --------  --------  --------
sales1       110       120       130       140       150
sales2       100       200       300       400       500
sales3      1000      2000      3000      4000      5000
sales4      1100      1200      1300      1400      1500
sales5       500       600       700       800       900
```

Example: Aggregation on Two Columns from PIVOT Results

This example shows how to sum sales in the months of Jan and Feb for each year. This is an aggregation on two columns from the PIVOT result.

Table s1 is defined as:

```
CREATE TABLE s1 (yr INTEGER, mon VARCHAR(20), sales INTEGER);
```

The table contains:

```
SELECT * FROM s1;
yr      mon      sales
-----   ---      -----
2001    Jan       100
2003    Jan       300
2002    Jan       150
2001    Feb       110
2003    Feb       310
2002    Feb       200
2001    Mar       120
2002    Mar       250
```

The PIVOT query is:

```
SELECT * FROM s1 PIVOT(SUM(SALES) FOR MON IN ('JAN', 'FEB', 'MAR')
  WITH SUM("'JAN'", "'FEB'") AS AGGR1 ) DT
order by 1;
```

AGGR1 is the name of the aggregated result column.

Output:

```
      yr        'JAN'        'FEB'        'MAR'        AGGR1
--------  -----------  -----------  -----------  -----------
    2001          100          110          120          210
    2002          150          200          250          350
    2003          300          310            ?          610
```

Example: Subquery in PIVOT IN-List

This is an example of having a subquery in PIVOT IN-list.

Table s1 is defined as:

```
CREATE TABLE s1(yr INTEGER, mon VARCHAR (5), sales INTEGER);
CREATE TABLE s2(yr INTEGER, mon VARCHAR (5), sales INTEGER);
```

The table contains:

```
SELECT * FROM s1;
yr      mon      sales
-----   ---      -----
2001    Jan       100
2003    Jan       300
2002    Jan       150
2001    Feb       110
2003    Feb       310
2002    Feb       200
2001    Mar       120
2002    Mar       250

SELECT * FROM s2;
 yr     mon      sales
-----  -----    -------
2001    Jan       100
2002    Mar       250
2003    Feb       310
```

The table as a source to a PIVOT query having a subquery in PIVOT IN-list:

```
SELECT * FROM s1 PIVOT (SUM (sales) FOR mon in (SELECT mon FROM s2)) dt;
```

The output pivoted table:

```
*** Query completed. 3 rows found. 4 columns returned.
 *** Total elapsed time was 1 second.
```

```
         yr        'Feb'        'Jan'        'Mar'
-----------  -----------  -----------  -----------
       2001          110          100          120
       2003          310          300            ?
       2002          200          150          250
```

Related Information

- For more information, see UNPIVOT.

REGR_AVGX

Returns the mean of the *independent_variable_expression* for all non-null data pairs of the dependent and independent variable arguments.

For the REGR_AVGX window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Computation

When there are fewer than two non-null data point pairs in the data used for the computation, then REGR_AVGX returns NULL.

Division by zero results in NULL rather than an error.

REGR_AVGX Function Syntax

```
REGR_AVGX (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for REGR_AVGX(y, x) are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as y.
- If the operand is a UDT, the format is the format for the data type to which the UDT is implicitly cast.

For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

REGR_AVGX Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_AVGX, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_AVGX can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_AVGX cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Mean Height for regrtbl

This example is based the following regrtbl data. Nulls are indicated by the QUESTION MARK character.

```
c1   height   weight
--   ------   ------
 1       60       84
 2       62       95
 3       64      140
 4       66      155
 5       68      119
 6       70      175
 7       72      145
 8       74      197
 9       76      150
10       76      ?
11       ?       150
12       ?       ?
```

The following SELECT statement returns the mean height for regrtbl where neither weight nor height is null.

```
   SELECT REGR_AVGX(weight,height)
   FROM regrtbl;

   Regr_Avgx(weight,height)
   ------------------------
                         68
```

REGR_AVGY

Returns the mean of the *dependent_variable_expression* for all non-null data pairs of the dependent and independent variable arguments.

For the REGR_AVGY window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Computation

When there are fewer than two non-null data point pairs in the data used for the computation, then REGR_AVGY returns NULL.

Division by zero results in NULL rather than an error.

REGR_AVGY Function Syntax

```
REGR_AVGY (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for REGR_AVGY(y, x) are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as y.
- If the operand is a UDT, the format is the format for the data type to which the UDT is implicitly cast.

REGR_AVGY Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_AVGY, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_AVGY can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_AVGY cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Mean Weight from regrtbl

This example is based the following regrtbl data. Nulls are indicated by the QUESTION MARK character.

```
c1   height   weight
--   ------   ------
 1       60       84
 2       62       95
 3       64      140
 4       66      155
 5       68      119
 6       70      175
 7       72      145
 8       74      197
 9       76      150
10       76      ?
11       ?       150
12       ?       ?
```

The following SELECT statement returns the mean weight from regrtbl where neither height nor weight is null.

```
   SELECT REGR_AVGY(weight,height)
   FROM regrtbl;

   Regr_Avgy(weight,height)
   ------------------------
                        140
```

Related Information

Teradata Vantage™ - Data Types and Literals, B035-1143:

- Information on the default format of data types and an explanation of the formatting characters in the format
- Information on implicit type conversion of UDTs

REGR_COUNT

Returns the count of all non-null data pairs of the dependent and independent variable arguments.

For the REGR_COUNT window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

REGR_COUNT Function Syntax

```
REGR_COUNT (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The following table lists the data type for the result of REGR_COUNT(y,x).

Mode	Data Type
ANSI	If MaxDecimal in DBSControl is… • 0 or 15, then the result type is DECIMAL(15,0). • 18, then the result type is DECIMAL(18,0). • 38, then the result type is DECIMAL(38,0).
Teradata	INTEGER

The result type of REGR_COUNT is consistent with the result type of COUNT for ANSI transaction mode and Teradata transaction mode.

When in Teradata mode, if the result of REGR_COUNT overflows and reports an error, you can cast the result to another data type, as illustrated by the following example.

```
SELECT CAST(REGR_COUNT(weight,height) AS BIGINT)
FROM regrtbl;
```

Here are default formats and titles for the result of REGR_COUNT.

- If operand y is numeric or character, the format is:
    - For ANSI mode, if MaxDecimal in DBSControl is:

      0 or 15, the format is -(15)9

    18, the format is -(18)9

    38, the format is -(38)9

  - For Teradata mode, the format is the default format for INTEGER
- If operand y is UDT, the format is the format for the data type to which the UDT is implicitly cast.

REGR_COUNT Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_COUNT, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_COUNT can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_COUNT cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Number of Rows in regrtbl

This example is based the following regrtbl data. Nulls are indicated by the QUESTION MARK character.

c1	height	weight
1	60	84
2	62	95
3	64	140
4	66	155
5	68	119
6	70	175
7	72	145
8	74	197
9	76	150
10	76	?
11	?	150
12	?	?

The following SELECT statement returns the number of rows in regrtbl where neither height nor weight is null.

```
SELECT REG_COUNT(weight,height)
FROM regrtbl;
```

Here is the result:

```
Regr_Count(weight,height)
-------------------------
                        9
```

Related Information

- For information on data type default formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For information on the REGR_COUNT window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

REGR_INTERCEPT

Returns the intercept of the univariate linear regression line through all non-null data pairs of the dependent and independent variable arguments.

For the REGR_INTERCEPT window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Definition

The intercept is the point at which the regression line through the non-null data pairs in the sample intersects the ordinate, or y-axis, of the graph.

The plot of the linear regression on the variables is used to predict the behavior of the dependent variable from the change in the independent variable.

Note that this computation assumes a linear relationship between the variables.

There can be a strong nonlinear relationship between independent and dependent variables, and the computation of the simple linear regression between such variable pairs does not reflect such a relationship.

Independent and Dependent Variables

An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

A dependent variable is something that is measured in response to a treatment.

For example, you might want to test the ability of various promotions to enhance sales of a particular item.

In this case, the promotion is the independent variable and the sales of the item made as a result of the individual promotion is the dependent variable.

The value of the linear regression intercept tells you the predicted value for sales when there is no promotion for the item selected for analysis.

Computation

When there are fewer than two non-null data point pairs in the data used for the computation, then REGR_INTERCEPT returns NULL.

Division by zero results in NULL rather than an error.

REGR_INTERCEPT Function Syntax

```
REGR_INTERCEPT (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for REGR_INTERCEPT(y, x) are as follows.

Data Type	Format	Title
REAL	Default format of the REAL data type	REGR_INTERCEPT(y,x)

REGR_INTERCEPT Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_INTERCEPT, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_INTERCEPT can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_INTERCEPT cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Intercept of the Regression Line for NbrSold and SalesPrice

This example uses the data from the HomeSales table.

SalesPrice	NbrSold	Area
160000	126	358711030
180000	103	358711030
200000	82	358711030
220000	75	358711030
240000	82	358711030
260000	40	358711030
280000	20	358711030

The following query returns the intercept of the regression line for NbrSold and SalesPrice in the range of 160000 to 280000 in the 358711030 area.

```
SELECT CAST (REGR_INTERCEPT(NbrSold,SalesPrice) AS DECIMAL (5,1))
FROM HomeSales
WHERE area = 358711030
AND SalesPrice BETWEEN 160000 AND 280000;
```

Here is the result:

```
REGR_INTERCEPT(NbrSold,SalesPrice)
----------------------------------
                             249.9
```

Related Information

- For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For the REGR_INTERCEPT window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

REGR_R2

Returns the coefficient of determination for all non-null data pairs of the dependent and independent variable arguments.

For the REGR_R2 window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Computation

When there are fewer than two non-null data point pairs in the data used for the computation, then REGR_R2 returns NULL.

Division by zero results in NULL rather than an error.

REGR_R2 Function Syntax

```
REGR_R2 (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for REGR_R2(y, x) are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as y.
- If the operand is UDT, the format is the format for the data type to which the UDT is implicitly cast.

For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

REGR_R2 Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_R2, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_R2 can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_R2 cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Coefficient of Determination for Height and Weight

This example is based the following regrtbl data. Nulls are indicated by the QUESTION MARK character.

```
c1   height   weight
--   ------   ------
 1       60       84
 2       62       95
 3       64      140
 4       66      155
 5       68      119
 6       70      175
 7       72      145
 8       74      197
 9       76      150
10       76      ?
11       ?       150
12       ?       ?
```

The following SELECT statement returns the coefficient of determination for height and weight where neither height nor weight is null.

```
    SELECT CAST(REGR_R2(weight,height) AS DECIMAL(4,2))
    FROM regrtbl;

    REGR_R2(weight,height)
    ---------------------
                       .58
```

Related Information

- For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For the REGR_R2 window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

REGR_SLOPE

Returns the slope of the univariate linear regression line through all non-null data pairs of the dependent and independent variable arguments.

For the REGR_SLOPE window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Slope

The slope of the best fit linear regression is a measure of the rate of change of the regression of one independent variable on the dependent variable.

The plot of the linear regression on the variables is used to predict the behavior of the dependent variable from the change in the independent variable.

Note that this computation assumes a linear relationship between the variables.

There can be a strong nonlinear relationship between independent and dependent variables, and the computation of the simple linear regression between such variable pairs does not reflect such a relationship.

Independent and Dependent Variables

An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

A dependent variable is something that is measured in response to a treatment.

For example, you might want to test the ability of various promotions to enhance sales of a particular item.

In this case, the promotion is the independent variable and the sales of the item made as a result of the individual promotion is the dependent variable.

Computation

When there are fewer than two non-null data point pairs in the data used for the computation, then REGR_SLOPE returns NULL.

Division by zero results in NULL rather than an error.

REGR_SLOPE Function Syntax

```
REGR_SLOPE (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for REGR_SLOPE(y, x) are as follows.

Data Type	Format	Title
REAL	Default format of the REAL data type	REGR_SLOPE(y,x)

REGR_SLOPE Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE

- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_SLOPE, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_SLOPE can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_SLOPE cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Slope of the Regression Line for NbrSold and SalesPrice

This example uses the data from the HomeSales table.

SalesPrice	NbrSold	Area
160000	126	358711030
180000	103	358711030
200000	82	358711030
220000	75	358711030
240000	82	358711030
260000	40	358711030
280000	20	358711030

The following query returns the slope of the regression line for NbrSold and SalesPrice in the range of 160000 to 280000 in the 358711030 area.

```
SELECT CAST (REGR_SLOPE(NbrSold,SalesPrice) AS FLOAT)
FROM HomeSales
WHERE area = 358711030
AND SalesPrice BETWEEN 160000 AND 280000;
```

Here is the result:

```
REGR_SLOPE(NbrSold,SalesPrice)
------------------------------
        -7.92857142857143E-004
```

Related Information

- For information on the default format of data types and the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For the REGR_SLOPE window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

REGR_SXX

Returns the sum of the squares of the *independent_variable_expression* for all non-null data pairs of the dependent and independent variable arguments.

For the REGR_SXX window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Computation

When there are fewer than two non-null data point pairs in the data used for the computation, then REGR_SXX returns NULL.

Division by zero results in NULL rather than an error.

REGR_SXX Function Syntax

```
REGR_SXX (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for REGR_SXX(y, x) are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as y.
- If the operand is UDT, the format is the format for the data type to which the UDT is implicitly cast.

REGR_SXX Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_SXX, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_SXX can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_SXX cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Sum of Squares for Height

This example is based the following regrtbl data. Nulls are indicated by the QUESTION MARK character.

```
c1   height   weight
--   ------   ------
 1       60       84
 2       62       95
 3       64      140
 4       66      155
 5       68      119
 6       70      175
 7       72      145
 8       74      197
 9       76      150
10       76      ?
11        ?      150
12        ?      ?
```

The following SELECT statement returns the sum of squares for height where neither height nor weight is null.

```
   SELECT REGR_SXX(weight,height)
   FROM regrtbl;
```

```
Regr_Sxx(weight,height)
-----------------------
                    240
```

Related Information

- For information on the default format of data types, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For the REGR_SXX window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

REGR_SXY

Returns the sum of the products of the *independent_variable_expression* and the *dependent_variable_expression* for all non-null data pairs of the dependent and independent variable arguments.

For the REGR_SXY window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Computation

When there are fewer than two non-null data point pairs in the data used for the computation, then REGR_SXY returns NULL.

Division by zero results in NULL rather than an error.

REGR_SXY Function Syntax

```
REGR_SXY (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for REGR_SXY(y, x) are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as y.
- If the operand is UDT, the format is the format for the data type to which the UDT is implicitly cast.

REGR_SXY Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_SXY, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_SXY can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_SXY cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Sum of Products of Height and Weight

This example is based the following regrtbl data. Nulls are indicated by the QUESTION MARK character.

```
c1   height   weight
--   ------   ------
 1       60       84
 2       62       95
 3       64      140
 4       66      155
 5       68      119
 6       70      175
 7       72      145
 8       74      197
 9       76      150
10       76      ?
11       ?       150
12       ?       ?
```

The following SELECT statement returns the sum of products of height and weight where neither height nor weight is null.

```
    SELECT REGR_SXY(weight,height)
    FROM regrtbl;

    Regr_Sxy(weight,height)
    -----------------------
                       1200
```

Related Information

Teradata Vantage™ - Data Types and Literals, B035-1143:

- Information on the default format of data types and an explanation of the formatting characters in the format
- Information on implicit type conversion of UDTs

REGR_SYY

Returns the sum of the squares of the *dependent_variable_expression* for all non-null data pairs of the dependent and independent variable arguments.

For the REGR_SYY window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Computation

When there are fewer than two non-null data point pairs in the data used for the computation, then REGR_INTERCEPT returns NULL.

Division by zero results in NULL rather than an error.

REGR_SYY Function Syntax

```
REGR_SYY (dependent_variable_expression, independent_variable_expression)
```

Syntax Elements

dependent_variable_expression

The dependent variable for the regression. A dependent variable is something that is measured in response to a treatment.

The expression cannot contain any ordered analytical or aggregate functions.

independent_variable_expression

The independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable.

The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for REGR_SYY(y, x) are as follows.

Data type: REAL

- If the operand is character, the format is the default format for FLOAT.
- If the operand is numeric, date, or interval, the format is the same format as y.
- If the operand is UDT, the format is the format for the data type to which the UDT is implicitly cast.

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

REGR_SYY Usage Notes

Support for UDTs

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts that cast between the UDTs and any of the following predefined types:

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including REGR_SYY, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Setting Up Axes for Plotting

If you export the data for plotting, define the y-axis (ordinate) as the dependent variable and the x-axis (abscissa) as the independent variable.

Combination With Other Functions

REGR_SYY can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

REGR_SYY cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

Example: Returning the Sum of Squares for Weight

This example is based the following regrtbl data. Nulls are indicated by the QUESTION MARK character.

```
c1   height   weight
--   ------   ------
 1       60       84
 2       62       95
 3       64      140
 4       66      155
 5       68      119
 6       70      175
 7       72      145
 8       74      197
 9       76      150
10       76      ?
11       ?       150
12       ?       ?
```

The following SELECT statement returns the sum of squares for weight where neither height nor weight is null.

```
   SELECT REGR_SYY(weight,height)
   FROM regrtbl;

   Regr_Syy(weight,height)
   -----------------------
                     10426
```

SKEW

Returns the skewness of the distribution of *value_expression*.

This function returns the REAL data type.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

Skewness

Skewness is the third moment of a distribution. It is a measure of the asymmetry of the distribution about its mean compared with the normal, Gaussian, distribution.

The normal distribution has a skewness of 0.

Positive skewness indicates a distribution having an asymmetric tail extending toward more positive values, while negative skewness indicates an asymmetric tail extending toward more negative values.

Computation

The equation for computing SKEW is defined as follows:

$$\text{SKEW} = \frac{\text{COUNT(x)}}{(\text{COUNT(x)} - 1)(\text{COUNT(x)} - 2)} \bullet \text{SUM}\left(\frac{\text{x} - \text{AVG(x)}}{(\text{STDDEV\_SAMP(x)**3})}\right)$$

where:

This variable ...	Represents ...
x	*value_expression*

Conditions That Produce a Null Result

The following conditions product a null result:

- Fewer than three non-null data points in the data used for the computation
- STDDEV_SAMP(x) = 0
- Division by zero

SKEW Function Syntax

```
SKEW ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

Null and duplicate values specified by *value_expression* are eliminated from the computation for the group.

ALL

All values of *value_expression* that are not null, including duplicates, are included in the computation.

value_expression

A literal or column expression for which the skewness of the distribution of its values is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Related Information

Teradata Vantage™ - Data Types and Literals, B035-1143.

STDDEV_POP

Returns the population standard deviation for the non-null data points in *value_expression*.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

For the STDDEV_POP window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Standard Deviation

The standard deviation is the second moment of a population. For a population, it is a measure of dispersion from the mean of that population.

Do not use STDDEV_POP unless the data points you are processing are the complete population.

Computation

STANDARD DEVIATION OF A SAMPLE is valid only for numeric data.

Nulls are not included in the result computation.

When there are no non-null data points in the population, then STDDEV_POP returns NULL.

Division by zero results in NULL rather than an error.

STDDEV_POP Function Syntax

```
STDDEV_POP ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

To exclude duplicates of *value_expression* from the computation.

ALL

Include all values that are not null specified by *value_expression*, including duplicates, in the computation. This is the default.

value_expression

A numeric literal or column expression whose population standard deviation is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Return Values

This function returns the REAL data type.

- If the *value_expression* is character, the format is the default format for FLOAT.
- If the *value_expression* is numeric, date, or interval, the format is the same format as x.
- If the *value_expression* is UDT, the format is the format for the data type to which the UDT is implicitly cast.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

STDDEV_POP Usage Notes

Combination With Other Functions

STDDEV_POP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

STDDEV_POP cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

How GROUP BY Affects Report Breaks

STDDEV_POP operates differently depending on whether there is a GROUP BY clause in the SELECT statement.

IF the query …	THEN STDDEV_POP is reported for …
specifies a GROUP BY clause	each individual group.
does not specify a GROUP BY clause	all the rows in the sample.

Measuring the Standard Deviation of a Population

If your data represents only a sample of the entire population for the variable, then use the STDDEV_SAMP function. For information, see STDDEV_SAMP.

As the sample size increases, the values for STDDEV_SAMP and STDDEV_POP approach the same number, but you should always use the more conservative STDDEV_SAMP calculation unless you are absolutely certain that your data constitutes the entire population for the variable.

Related Information

- *Teradata Vantage™ - Data Types and Literals*, B035-1143
- Window Aggregate Functions

STDDEV_SAMP

Returns the sample standard deviation for the non-null data points in *value_expression*.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

For the STDDEV_SAMP window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

Standard Deviation

The standard deviation is the second moment of a distribution. For a sample, it is a measure of dispersion from the mean of that sample. The computation is more conservative for the population standard deviation to minimize the effect of outliers on the computed value.

Computation

Division by zero results in NULL rather than an error.

When there are fewer than two non-null data points in the sample used for the computation, then STDDEV_SAMP returns NULL.

STDDEV_SAMP Function Syntax

```
STDDEV_SAMP ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

Exclude duplicates of *value_expression* from the computation.

ALL

All values of *value_expression* that are not null, including duplicates, are included in the computation.

value_expression

A numeric literal or column expression whose sample standard deviation is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Return Values

This function returns the REAL data type.

- If the *value_expression* is character, the format is the default format for FLOAT.
- If the *value_expression* is numeric, date, or interval, the format is the same format as x.
- If the *value_expression* is UDT, the format is the format for the data type to which the UDT is implicitly cast.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

STDDEV_SAMP Usage Notes

Combination With Other Functions

STDDEV_SAMP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Window Aggregate Functions.

STDDEV_SAMP cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

How GROUP BY Affects Report Breaks

The GROUP BY clause affects the STDDEV_SAMP operation.

IF the query …	THEN STDDEV_SAMP is reported for …
specifies a GROUP BY clause	each individual group.
does not specify a GROUP BY clause	all the rows in the sample.

Measuring the Standard Deviation of a Population

If your data represents the entire population for the variable, then use the STDDEV_POP function. For information, see STDDEV_POP.

As the sample size increases, the values for STDDEV_SAMP and STDDEV_POP approach the same number, but you should use the more conservative STDDEV_SAMP calculation unless you are absolutely certain that your data constitutes the entire population for the variable.

Related Information

- *Teradata Vantage™ - Data Types and Literals*, B035-1143
- Window Aggregate Functions

SUM

Returns a column value that is the arithmetic sum of *value_expression*.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

SUM Function Syntax

```
SUM ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

Exclude duplicates of *value_expression* from the computation.

ALL

All values of *value_expression* that are not null, including duplicates, are included in the computation.

value_expression

A literal or column expression whose sum is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

Return Values

The following table lists the default attributes for the result of SUM(x).

Data Type of Operand	Data Type of Result	Format	Title
BYTEINT or SMALLINT	INTEGER	Default format of the INTEGER data type	Sum(x)
character	FLOAT	Default format for FLOAT	
UDT	Same as the operand	Format for the data type to which the UDT is implicitly cast	
DECIMAL(*n*,*m*)	DECIMAL(*p*,*m*), where *p* is determined by the rules in the following rules: If MaxDecimal in DBSControl is 0 or 15 and • *n* ≤ 15, then *p* = 15. • 15 < *n* ≤ 18, *p* = 18. • *n* > 18, then *p* = 38. If MaxDecimal in DBSControl is 18 and • *n* ≤ 18, then *p* = 18. • *n* > 18, then *p* = 38. If MaxDecimal in DBSControl is 38 and *n* = any value, the *p* = 38.	Default format for the data type of the operand	Sum(x)
Other than UDT, SMALLINT, BYTEINT, DECIMAL, or character	Same as the operand	Default format for the data type of the operand	

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Usage Notes

SUM is valid only for numeric data.

Nulls are not included in the result computation. For details, see “Manipulating Nulls” in *Teradata Vantage™ - SQL Fundamentals*, B035-1141 and Aggregates and Nulls.

The SUM function can result in a numeric overflow or the loss of data because of the default output format. If this occurs, a data type declaration may be used to override the default.

For example, if QUANTITY comprises many rows of INTEGER values, it may be necessary to specify a data type declaration like the following for the SUM function:

```
SUM(QUANTITY(FLOAT))
```

Possible Result Overflow with SELECT Sum

Possible Result Overflow with SELECT Sum

When using this function, the result can create an overflow when the data type and format are not in sync. For a column defined as:

`Salary Decimal(15,2) Format ‘$ZZZ,ZZ9.99’`

The following query:

`SELECT SUM (Salary) FROM Employee;`

causes an overflow because the decimal operand and the format are not in sync.

To avoid possible overflows, explicitly specify the format for decimal sum to specify a format large enough to accommodate the decimal sum resultant data type.

`SELECT Sum(Salary) (format ‘$Z,ZZZ,ZZZ,ZZ9.99) FROM Employee;`

Examples

Example: Accounts Receivable

You need to know how much cash you need to pay all vendors who billed you 30 or more days ago.

```
    SELECT SUM(Invoice)
    FROM AcctsRec
    WHERE (CURRENT_DATE - InvDate) >= 30;
```

Example: Face Value of Inventory

You need to know the total face value for all items in your inventory.

```
    SELECT SUM(QUANTITY * Price)
    FROM Inventory;

    Sum((QUANTITY * Price))
```

```
   ----------------------
             38,525,151.91
```

Related Information

- For an explanation of the formatting characters in the format, and information on data type default formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For the SUM function that returns the cumulative, group, or moving sum, see Window Aggregate Functions.

UNPIVOT

UNPIVOT is the reverse of the PIVOT operation. It provides a mechanism for transforming columns into rows.

The UNPIVOT functionality was introduced previously via the TD_UNPIVOT table operator. This feature introduces grammar to support the UNPIVOT operator in the FROM clause of the SELECT statement.

Note:

UNPIVOT invokes the TD_UNPIVOT table operator internally. You can still use TD_UNPIVOT independent of UNPIVOT.

UNPIVOT Function Syntax

```
UNPIVOT [ { INCLUDE | EXCLUDE } NULLS ] (unpivot_spec)
  [AS] derived_table_name [ (cname [,...] ) ]
```

Syntax Elements

unpivot_spec

```
{ cname FOR cname IN (cname_spec_1 [,...]) |
  (cname [,...]) FOR cname IN (cname_spec_2 [,...] )
}
```

derived_table_name

The table name specified for the resultant unpivoted table.

cname

A column name.

Note:

For the UNPIVOT operation, column names within the Aggregate functions are referred to as measure columns, and column names in the FOR clause are referred to as pivot columns.

cname_spec_1

```
cname [ [AS] literal ]
```

cname_spec_2

```
(cname [,...]) [ [AS] literal ]
```

literal

Any supported Teradata numeric, character or string literal.

Usage Notes

Note:

Column names specified just before the FOR clause are referred to as *measure_columns* in the context of UNPIVOT operation. Column names specified after the FOR clause are referred to as *unpivot_columns*.

Similar to the PIVOT operator, columns with CLOB, BLOB, UDT, XML, or JSON data types are not allowed with the UNPIVOT operator.

The UNPIVOT column name and measure column names cannot be the same as the column names defined in the derived_table_name.

When multiple *measure_columns* are involved in UNPIVOT operation, the columns are compatible only if they belong to any of the following three groups:

- CHAR and VARCHAR
- BYTE and VARBYTE
- BYTEINT SMALLINT INTEGER BIGINT REAL DECIMAL NUMBER

Column names specified in the IN list cannot be specified in the assign list of the SELECT statement.

Examples

The examples in this section use the following denormalized pivoted table, *star1p*, which is defined as:

```
CREATE TABLE star1p(country VARCHAR(20),state VARCHAR(20),Q101Sales
INTEGER,Q201Sales INTEGER,Q301Sales INTEGER,Q101Cogs INTEGER,Q201Cogs
INTEGER,Q301Cogs INTEGER);

SELECT * FROM star1p;

country  state  Q101Sales  Q201Sales  Q301Sales  Q101Cogs  Q201Cogs  Q301Cogs
-------  -----  ---------  ---------  ---------  --------  --------  --------
Canada    ON        ?         10          ?         ?         0         ?
Canada    BC        ?          ?         10         ?         ?         0
USA       NY       45          ?          ?        25         ?         ?
USA       CA       30         50          ?        15        20         ?
```

Example: Unpivoted Sales and Cogs Columns

In this example, the sales and cogs columns are unpivoted.

```
SELECT *
FROM star1p UNPIVOT ((sales,cogs)  FOR  yr_qtr
                          IN ((Q101Sales, Q101Cogs) AS ‘Q101’,
                              (Q201Sales, Q201Cogs) AS ‘Q201’,
                              (Q301Sales, Q301Cogs) AS ‘Q301’)) Tmp;
```

The output for the unpivoted table:

```
country  state  yr_qtr        sales         cogs
-------  -----  ------  -----------  -----------
Canada   ON     Q201             10            0
Canada   ON     Q301             10            0
USA      NY     Q101             45           25
USA      CA     Q101             30           15
USA      CA     Q201             50           20
```

Note that a pivot combined with a matching unpivot may introduce rows with NULL values. It is possible to unpivot just the ‘yr’ column.

Example: Using UNPIVOT for a Unique Year Value

This example shows only one unique value of year, so the unpivot is straightforward.

```
SELECT *
FROM star1p UNPIVOT (Q1sales, Q2sales, Q3sales, Q1cogs, Q2cogs, Q3cogs) FOR
yr IN ((Q101Sales, Q201Sales, Q301Sales, Q101Cogs, Q201Cogs, Q301Cogs) AS
‘2001’) Tmp;

country  state  yr     Q1sales  Q2sales  Q3sales  Q1cogs  Q2cogs  Q3cogs
-------  -----  ----  --------  -------  -------  ------  ------  ------
Canada   ON     2001    ?         10        ?       ?        0       ?
Canada   BC     2001    ?          ?       10       ?        ?       0
USA      NY     2001    45         ?        ?       25       ?       ?
USA      CA     2001    30        50        ?       15       20
```

Example: Normalizing the UNPIVOT Operation

This example showcases using UNPIVOT to capture elaborate data of a base table (*star1p*, in this case). The data is spread over many columns into a compact table with an optimal number of columns and no data loss.

```
SELECT *
FROM star1p UNPIVOT (measure_value  FOR  yr_qtr_measure IN
(Q101Sales, Q201Sales, Q301Sales,Q101Cogs, Q201Cogs, Q301Cogs)) Tmp;
country  state  yr_qtr_measure  measure_value
-------  -----  --------------  -------------
Canada   BC     Q301Cogs                    0
Canada   BC     Q301Sales                  10
Canada   ON     Q201Cogs                    0
Canada   ON     Q201Sales                  10
USA      CA     Q101Cogs                   15
USA      CA     Q101Sales                  30
USA      CA     Q201Cogs                   20
USA      CA     Q201Sales                  50
USA      NY     Q101Cogs                   25
USA      NY     Q101Sales                  45
```

Example: Using UNPIVOT with the INCLUDE NULLS Clause

In this example, there are some rows with nulls in the sales and cogs columns. The rows are included in the output when using the INCLUDE NULLS clause.

```
SELECT *
FROM star1p UNPIVOT INCLUDE NULLS ((sales,cogs)  FOR  yr_qtr IN
((Q101Sales, Q101Cogs) AS 'Q101', (Q201Sales, Q201Cogs) AS 'Q201', (Q301Sales,
Q301Cogs) AS 'Q301')) Tmp;

country  state   yr_qtr       sales        cogs
-------  -----  ------    -----------  -----------
Canada     BC     Q101              ?            ?
Canada     ON     Q101              ?            ?
Canada     ON     Q201             10            0
Canada     ON     Q301             10            0
USA        NY     Q101             45           25
USA        CA     Q101             30           15
Canada     BC     Q201              ?            ?
USA        NY     Q201              ?            ?
USA        CA     Q201             50           20
Canada     BC     Q301              ?            ?
USA        NY     Q301              ?            ?
USA        CA     Q301              ?            ?
```

Example: Using UNPIVOT with the EXCLUDE NULLS Clause

In this example, there are no rows with nulls in either the sales or cogs columns, and the rows are excluded in the output when using EXCLUDE NULLS clause. This is the default option.

```
SELECT *
FROM star1p UNPIVOT EXCLUDE NULLS (sales, cogs)  FOR  yr_qtr IN
((Q101Sales, Q101Cogs) AS 'Q101', (Q201Sales, Q201Cogs) AS 'Q201', (Q301Sales,
Q301Cogs) AS 'Q301') Tmp;

country  state     yr_qtr       sales         cogs
-------  ------   --------   ------------    -------
Canada      ON     Q201           10            0
Canada      ON     Q301           10            0
USA         NY     Q101           45           25
USA         CA     Q101           30           15
USA         CA     Q201           50           20
```

Example: Using an IN List with Multiple Column Lists and Unspecified Aliases

In this example, the aliases that the IN list uses were not specified. Instead, the values of the yr_qtr column were built by adding the column names with an underscore symbol.

```
SELECT *
FROM star1p UNPIVOT ((sales, cogs)  FOR  yr_qtr IN
((Q101Sales, Q101Cogs),(Q201Sales, Q201Cogs), (Q301Sales, Q301Cogs)) Tmp;

country    state        yr_qtr                        sales         cogs
-------  -------- -----------------------            --------     --------
Canada     ON        Q201Sales_Q201Cogs                 10            0
Canada     ON        Q301Sales_Q301Cogs                 10            0
USA        NY        Q101Sales_Q101Cogs                 45           25
USA        CA        Q101Sales_Q101Cogs                 30           15
USA        CA        Q201Sales_Q201Cogs                 50           20
```

Example: Using an IN List that Contains Multiple Columns with a Compatible Data Type

In this example, the Q101Sales column contains an INTEGER data type, and Q201Sales is a BYTEINT data type. Both the INTEGER and BYTEINT data types are compatible with each other.

```
SELECT * FROM star1p UNPIVOT (measure_value  FOR  yr_qtr_measure IN
(Q101Sales, Q201Sales)) Tmp;

country     state  yr_qtr_measure  measure_value
-------     -----  --------------  -------------
Canada        ON     Q201Sales               10
USA           CA     Q101Sales               30
USA           CA     Q201Sales               50
USA           NY     Q101Sales               45
```

Example: Using an IN List that Contains Multiple Columns with an Incompatible Data Type

In this example, the *star1p* table is altered to contain a new column Q401Sales with a VARCHAR(20) data type. The Q101Sales column is an INTEGER data type, and the Q401Sales is VARCHAR.

The INTEGER and VARCHAR data types are not compatible.

```
SELECT *
FROM star1p UNPIVOT (measure_value  FOR  yr_qtr_measure IN
(Q101Sales, Q401Sales)) Tmp;

Error 9134 Failure in TD_Unpivot contract function. Error determining column
type of value columns.
```

Related Information

- PIVOT
- "TD_UNPIVOT" in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210

VAR_POP

Returns the population variance for the data points in *value_expression*.

This function returns the REAL data type.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

Variance

The variance of a population is a measure of dispersion from the mean of that population.

Do not use VAR_POP unless the data points you are processing are the complete population.

Computation

When the population has no non-null data points, VAR_POP returns NULL.

Division by zero results in NULL rather than an error.

VAR_POP Function Syntax

```
VAR_POP ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

To exclude duplicates of *value_expression* from the computation.

ALL

All values of *value_expression* that are not null, including duplicates, are included in the computation.

value_expression

A numeric literal or column expression whose population variance is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

The following restrictions apply to *value_expression*:

- If the *value_expression* is character, the format is the default format for FLOAT.
- If the *value_expression* is numeric, date, or interval, the format is the same format as x.
- If the *value_expression* is UDT, the format is the format for the data type to which the UDT is implicitly cast.

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

VAR_POP Usage Notes

Combination With Other Functions

VAR_POP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause.

VAR_POP cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

GROUP BY Affects Report Breaks

The GROUP BY clause affects the VAR_POP operation.

IF the query …	THEN VAR_POP is reported for …
specifies a GROUP BY clause	each individual group.
does not specify a GROUP BY clause	all the rows in the sample.

Measuring the Standard Deviation of a Population

If your data represents the only a sample of the entire population for the variable, then use the VAR_SAMP function. For information, see “VAR_SAMP”.

As the sample size increases, the values for VAR_SAMP and VAR_POP approach the same number, but you should always use the more conservative STDDEV_SAMP calculation unless you are absolutely certain that your data constitutes the entire population for the variable.

Related Information

- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on ordered analytical functions, see Ordered Analytical/Window Aggregate Functions.
- For the VAR_POP window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.

VAR_SAMP

Returns the sample variance for the data points in *value_expression*.

To invoke the time series version of this function, use the GROUP BY TIME clause. For more information, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

Variance

The variance of a sample is a measure of dispersion from the mean of that sample. It is the square of the sample standard deviation.

The computation is more conservative than that for the population standard deviation to minimize the effect of outliers on the computed value.

Computation

When the sample used for the computation has fewer than two non-null data points, VAR_SAMP returns NULL.

Division by zero results in NULL rather than an error.

VAR_SAMP Function Syntax

```
VAR_SAMP ( [ DISTINCT | ALL ] value_expression )
```

Syntax Elements

DISTINCT

To exclude duplicates of *value_expression* from the computation.

ALL

All values of *value_expression* that are not null, including duplicates, are included in the computation.

value_expression

A numeric literal or column expression whose sample variance is to be computed.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

- If the *value_expression* is character, the format is the default format for FLOAT.
- If the *value_expression* is numeric, date, or interval, the format is the same format as x.
- If the *value_expression* is UDT, the format is the format for the data type to which the UDT is implicitly cast.

The *value_expression* cannot be a reference to a view column derived from a function, and cannot contain any ordered analytical or aggregate functions.

VARIANCE OF A SAMPLE is valid only for numeric data.

Nulls are not included in the result computation.

Division by zero results in NULL rather than an error.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

VAR_SAMP Usage Notes

Combination With Other Functions

VAR_SAMP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause.

VAR_SAMP cannot be combined with aggregate functions within the same SELECT list, QUALIFY clause, or ORDER BY clause.

GROUP BY Affects Report Breaks

VAR_SAMP operates differently depending on whether or not there is a GROUP BY clause in the SELECT statement.

IF the query …	THEN VAR_SAMP is reported for …
specifies a GROUP BY clause	each individual group.
does not specify a GROUP BY clause	all the rows in the sample.

Measuring the Variance of a Population

If your data represents the entire population for the variable, then use the VAR_POP function.

As the sample size increases, the values for VAR_SAMP and VAR_POP approach the same number, but you should always use the more conservative VAR_SAMP calculation unless you are absolutely certain that your data constitutes the entire population for the variable.

Related Information

- For more information on ordered analytical functions, see Ordered Analytical/Window Aggregate Functions.
- For the VAR_SAMP window function that performs a group, cumulative, or moving computation, see Window Aggregate Functions.
- If your data represents the entire population for the variable, then use the VAR_POP function. For information, see VAR_POP.
- For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For details about the DisableUDTImplCastForSysFuncOp field, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

4

Arithmetic, Trigonometric, Hyperbolic Operators/Functions

The following sections describe the SQL arithmetic, trigonometric, and hyperbolic operators and functions.

Teradata SQL supports the following arithmetic operators.

Operator	Function
**	Exponentiate This is a Teradata extension to the ANSI SQL:2011 standard.
*	Multiply
/	Divide
MOD	Modulo (remainder). MOD calculates the remainder in a division operation. For example, 60 MOD 7 = 4: 60 divided by 7 equals 8, with a remainder of 4. The result takes the sign of the dividend, thus: -17 MOD 4 = -1 -17 MOD -4 = -1 17 MOD -4 = 1 17 MOD 4 = 1 This is a Teradata extension to the ANSI SQL:2011 standard.
+	Add
-	Subtract
+	Unary plus (positive value)
-	Unary minus (negative value)

ANSI Compliance

Except for MOD and **, the arithmetic operators are ANSI SQL:2011 compliant.

Arithmetic Operators and LOBs

Arithmetic operators do not support BLOB or CLOB types.

Arithmetic Operators and UDTs

By default, Vantage performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and a predefined numeric data type such as FLOAT or INTEGER.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

Implicit type conversion of UDTs for system operators and functions, including arithmetic operators, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

Related Information

- For details on the arithmetic operators permitted for DateTime and Interval data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143 .
- For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For details about the DisableUDTImplCastForSysFuncOp field of the DBS Control Record, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For more information on implicit type conversion of UDTs, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Binary Arithmetic Result Data Types

The data type of the result of an arithmetic expression depends on the data types of the two operands. Operands are converted to the result type before the operation is performed.

For example, before an INTEGER value is added to a FLOAT value, the INTEGER value is converted to FLOAT, the data type of the result.

Result Data Type

The following table shows the result data type for binary arithmetic operators.

The result data type for binary arithmetic operations involving UDT operands is the same as the result data type for the predefined data types to which the UDTs are implicitly cast.

For details on the result data type for binary arithmetic operations involving DateTime and Interval types, see Arithmetic, Trigonometric, Hyperbolic Operators/Functions.

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …
any data type	any data type	**	FLOAT
DATE	BYTEINT SMALLINT INTEGER BIGINT	+ -	DATE If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
	BYTEINT	* / MOD	INTEGER

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …
	SMALLINT INTEGER		These operations on DATE do not report an error, but results are generally not meaningful.
	BIGINT	* / MOD	BIGINT These operations on DATE do not report an error, but results are generally not meaningful.
	DECIMAL(*k*,*j*)	+ -	DATE These operations on DATE do not report an error, but results are generally not meaningful. Fractions of decimal values are truncated when added to or subtracted from date values. If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		* / MOD	DECIMAL(*p*,*j*) These operations on DATE do not report an error, but results are generally not meaningful. For details about the value of *p*, see DECIMAL Result Data Type.

	NUMBER(*k*,*j*)
	NUMBER(*k*)
	NUMBER(*,*j*)
	NUMBER
		+ -	DATE
			These operations on DATE do not report an error, but results are generally not meaningful.
			Fractions of decimal values are truncated when added to or subtracted from date values.
			If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.

		* / MOD	NUMBER
	FLOAT	* / + - MOD	FLOAT
	DATE	-	INTEGER The difference between two dates is the number of days between those dates. Note that this is not the numeric difference between the values.
		+ * / MOD	INTEGER These operations on DATE do not report an error, but results are generally not meaningful.

	CHAR(*n*)
	VARCHAR(*n*)
		* / + - MOD	FLOAT
			These operations on DATE do not report an error, but results are generally not meaningful.

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …
			If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.
BYTEINT SMALLINT INTEGER	BYTEINT SMALLINT INTEGER	* / + - MOD	INTEGER
	BIGINT	* / + - MOD	BIGINT
	DECIMAL(*k*,*j*)	* / + - MOD	DECIMAL(*p*,*j*) For details about the value of *p*, see DECIMAL Result Data Type.

	NUMBER(*k*,*j*)
	NUMBER(*k*)
	NUMBER(*,*j*)
	NUMBER
		* / + - MOD	NUMBER

	FLOAT	* / + - MOD	FLOAT

	CHAR(*n*)
	VARCHAR(*n*)
		* / + - MOD	FLOAT
			If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

	DATE	+	DATE If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		-	error
		* / MOD	INTEGER These operations on DATE do not report an error, but results are generally not meaningful.
BIGINT	BYTEINT SMALLINT INTEGER BIGINT	* / + - MOD	BIGINT
	DECIMAL(*k*,*j*)	* / + - MOD	DECIMAL(*p*,*j*) For details about the value of *p*, see DECIMAL Result Data Type.
	NUMBER(*k*,*j*)	* / + - MOD	NUMBER

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …

	NUMBER(*k*)
	NUMBER(*,*j*)
	NUMBER		

	FLOAT	* / + - MOD	FLOAT

	CHAR(*n*)
	VARCHAR(*n*)
		* / + - MOD	FLOAT
			If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

	DATE	+	DATE If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		-	error
		* / MOD	BIGINT These operations on DATE do not report an error, but results are generally not meaningful.
DECIMAL(*m*,*n*)	BYTEINT SMALLINT INTEGER BIGINT	+ - *	DECIMAL(*p*,*n*) For details about the value of *p*, see DECIMAL Result Data Type.
		/ MOD	DECIMAL(*m*,*n*)
	DECIMAL(*k*, *j*)	+ -	DECIMAL(min(*p*,(1+max(*n*,*j*)+max(*m* -*n*,*k*-*j*))), max(*n*,*j*)) For details about the value of *p*, see DECIMAL Result Data Type.
		*	DECIMAL(min(*p*,*m* +*k*),(*n*+*j*)) For details about the value of *p*, see DECIMAL Result Data Type.
		/ MOD	DECIMAL(*p*,max(*n*,*j*)) For details about the value of *p*, see DECIMAL Result Data Type.

	NUMBER(*k*,*j*)
	NUMBER(*k*)
	NUMBER(*,*j*)
	NUMBER
		* / + - MOD	NUMBER

	FLOAT	* / + - MOD	FLOAT

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …

	CHAR(*n*)
	VARCHAR(*n*)
		* / + - MOD	FLOAT
			If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

	DATE	+	DATE Fractions of decimal values are truncated when added to or subtracted from date values. If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		-	error
		*	DECIMAL(*p*,*n*) These operations on DATE do not report an error, but results are generally not meaningful. For details about the value of *p*, see DECIMAL Result Data Type.
		/ MOD	DECIMAL(*m*,*n*) These operations on DATE do not report an error, but results are generally not meaningful.

NUMBER(*m*,*n*)
NUMBER(*m*)
NUMBER(*,*n*)
NUMBER
	BYTEINT
	SMALLINT
	INTEGER
	BIGINT
		* / + - MOD	NUMBER

	DECIMAL(*k*,*j*)		
	FLOAT	* / + - MOD	FLOAT

	CHAR(*n*)
	VARCHAR(*n*)
			FLOAT
			If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

	DATE	+	DATE Fractions of decimal values are truncated when added to or subtracted from date values. If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		-	error

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …
		*	NUMBER These operations on DATE do not report an error, but results are generally not meaningful.
		/ MOD	NUMBER These operations on DATE do not report an error, but results are generally not meaningful.

	NUMBER(*k*,*j*)
	NUMBER(*k*)
	NUMBER(*,*j*)
	NUMBER
		* / + - MOD	NUMBER

FLOAT	BYTEINT SMALLINT INTEGER BIGINT DECIMAL(*k*,*j*) NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER FLOAT	* / + - MOD	FLOAT
	DATE	* / + - MOD	FLOAT These operations on DATE do not report an error, but results are generally not meaningful.

	CHAR(*n*)
	VARCHAR(*n*)
		* / + - MOD	FLOAT
			If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

CHAR(*n*)
VARCHAR(*n*)
	BYTEINT
	SMALLINT
	INTEGER
	BIGINT
	DECIMAL(*k*,*j*)
	NUMBER(*k*,*j*)
	NUMBER(*k*)
	NUMBER(*,*j*)
	NUMBER
	FLOAT
	CHAR(*n*)
	VARCHAR(*n*)
		* / + - MOD	FLOAT
			If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …
	DATE	* / + - MOD	FLOAT These operations on DATE do not report an error, but results are generally not meaningful. If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

DECIMAL Result Data Type

The result data type for binary arithmetic operations involving DECIMAL operands is as follows:

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …
DATE	DECIMAL(*k*,*j*)	* / MOD	DECIMAL(*p*,*j*) These operations on DATE do not report an error, but results are generally not meaningful.
BYTEINT SMALLINT INTEGER BIGINT		* / + - MOD	DECIMAL(*p*,*j*)
DECIMAL(*m*,*n*)	BYTEINT SMALLINT INTEGER BIGINT	+ - *	DECIMAL(*p*,*n*)
	DATE	*	DECIMAL(*p*,*n*) These operations on DATE do not report an error, but results are generally not meaningful.

In these cases the value of *p*, the number of digits in the decimal result, depends on:

- The value specified for MaxDecimal in DBSControl.

  For more information on DBSControl and MaxDecimal, see “DBS Control utility” in *Teradata Vantage™ - Database Utilities*, B035-1102.
- The number of digits in the decimal operand, where the number of digits is *k* for a DECIMAL(*k*,*j*) operand on the right side of the operator or *m* for a DECIMAL(*m*,*n*) operand on the left side of the operator.

IF MaxDecimal is …	AND the number of digits in the decimal operand is …	THEN *p* is …
0 or 15	≤ 15	15
	> 15 and ≤18	18
	> 18	38
18	≤ 18	18
	> 18	38
38	any value	38

When the operand on the left is …	And the operand on the right is …	And the operator is …	Then the result data type is …
DECIMAL(*m*,*n*)	DECIMAL(*k*,*j*)	+ -	DECIMAL(min(*p*, (1+max(*n*,*j*)+max(*m* -*n*,*k* -*j*))),max(*n*,*j*))
		*	DECIMAL(min(*p*,*m* +*k*),(*n* +*j*))
		/ MOD	DECIMAL(*p*,max(*n*,*j*))

In these cases, the value of *p* in the definition of the decimal result data type depends on the value specified for MaxDecimal in DBSControl and the number of digits in the DECIMAL(*m*,*n*) and DECIMAL(*k*,*j*) operands.

IF MaxDecimal is …	AND …	THEN *p* is …
0 or 15	*m* and *k* ≤ 15	15
	(*m* or *k* > 15) and (*m* and *k* ≤18)	18
	m or *k* > 18	38
18	*m* and *k* ≤ 18	18
	m or *k* > 18	38
38	*m* and *k* = any value	38

Numeric Results and Rounding

When computing an expression, numeric results that are not exact are rounded, not truncated.

For more information on rounding rules and how the RoundHalfwayMagUp and RoundNumberAsDec fields in DBSControl affect rounding, see “Numeric Data Types” in *Teradata Vantage™ - Data Types and Literals*, B035-1143 and “DBS Control utility” in *Teradata Vantage™ - Database Utilities*, B035-1102.

Error Conditions

An error is reported when any of the following events occurs:

- Division by zero is attempted.
- The numeric range is exceeded.
- The exponentiation operator is used with a negative left argument and a right argument that is not a whole number.

Integer Division and Truncation

Integer division yields whole results, truncated toward zero.

Structure of Arithmetic Expressions

Order of Evaluation

The following table lists the precedence of operations in arithmetic expressions.

Precedence	Operation
Highest	+ operand (unary plus) - operand (unary minus)
Intermediate	operand ** operand (exponentiation)
	operand * operand (multiplication) operand / operand (division) operand MOD operand (modulo operator)
	operand + operand (addition) operand - operand (subtraction)

In general, the order of evaluation is:

1. Operations enclosed in parentheses are performed first.
2. When no parentheses are present, operations are performed in order of precedence.
3. Operators of the same precedence are evaluated from left to right.

The Optimizer may reorder evaluations based on associative and commutative properties of the operations involved.

Format

The format of an arithmetic expression is the same as the default format of the result data type.

You can use the FORMAT phrase to change the default format of the result data type. The FORMAT phrase is relevant only in field mode, such as BTEQ applications, and in conversion to a character data type.

Example: Determining Employee Salary Increases

You want to raise the salary for each employee in department 600 by $200 for each year spent with the company (up to a maximum of $2500 per month).

To determine who is eligible, and the new salary, enter the following statement:

```
SELECT Name, (Salary+(YrsExp*200))/12 AS Projection
FROM Employee
WHERE Deptno = 600
AND Projection < 2500 ;
```

This statement returns the following response:

```
Name        Projection
--------    ----------
Newman P       2483.33
```

The statement uses parentheses to perform the operation YrsExp * 200 first. Its result is then added to Salary and the total is divided by 12.

The parentheses enclosing YrsExp * 200 are not strictly necessary, but the parentheses enclosing Salary + (YrsExp * 200) are necessary, because, if no parentheses were used in this expression, the operation YrsExp * 200 would be divided by 12 and the result added to Salary, producing an erroneous value.

The phrase AS Projection in this example associates the arithmetic expression (Salary + (YrsExp * 200)/12) with Projection. Using the AS phrase lets you use the name Projection in the WHERE clause to refer to the entire expression.

The result is formatted without a comma separating thousands from hundreds.

Arithmetic Functions

The next sections describe the following arithmetic functions:

- ABS
- CASE_N
- CEILING
- EXP
- FLOOR
- LN
- LOG

- MOD
- NULLIFZERO
- RANDOM
- RANGE_N
- SQRT
- WIDTH_BUCKET
- ZEROIFNULL

ABS

Computes the absolute value of an argument.

ABS Function Syntax

```
ABS (arg)
```

Syntax Elements

arg

A numeric column on which ABS() is requested. You must specify the input as a dynamic UDT.

Argument Types and Rules

If the argument is not numeric, it is converted to a numeric value, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If *arg* is a character string, it is converted to a numeric value of the FLOAT data type.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DateTime
  - Interval

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Implicit type conversion of UDTs for system operators and functions, including ABS, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the

DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

ABS cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

Here are the default attributes for the result of ABS.

- If the operand is numeric, the format is the default format for the resulting data type.
- If the operand is character, the format is the default format for FLOAT.
- If the operand is a UDT, the format is the default format for the predefined type to which the UDT is implicitly cast.

Note:

The NULL keyword has a data type of INTEGER.

For information on data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Examples: Using ABS Arithmetic Function Expressions

Representative ABS arithmetic function expressions and the results are as follows.

Expression	Result
ABS(-12)	12
ABS('23')	2.30000000000000E+001

CASE_N

Evaluates a list of conditions and returns the position of the first condition that evaluates to TRUE, provided that no prior condition in the list evaluates to UNKNOWN.

CASE_N Function Syntax

```
CASE_N ( conditional_expression [,...] [, case_spec ] )
```

Syntax Elements

conditional_expression

An expression or comma-separated list of condition expressions to evaluate.

A conditional expression must evaluate to TRUE, FALSE, or UNKNOWN.

CASE_N evaluates *conditional_expressions* from left to right until a condition evaluates to TRUE or UNKNOWN, or until every condition evaluates to FALSE. The position of the first *conditional_expression* is one and the positions of subsequent conditions increment by one up to *n*, where *n* is the total number of conditional expressions.

IF …	THEN …
a *conditional_expression* evaluates to TRUE, and all prior conditions evaluate to FALSE	CASE_N returns the position of the *conditional_expression*.
a *conditional_expression* evaluates to UNKNOWN, and all prior conditions evaluate to FALSE	If NO CASE OR UNKNOWN is specified, CASE_N returns *n* + 1. If UNKNOWN is specified and NO CASE is not specified, CASE_N returns *n* + 1. If NO CASE and UNKNOWN are specified, CASE_N returns *n* + 2. If neither UNKNOWN nor NO CASE OR UNKNOWN is specified, CASE_N returns NULL.
every *conditional_expression* evaluates to FALSE	If NO CASE or NO CASE OR UNKNOWN is specified, CASE_N returns *n* + 1. If neither NO CASE nor NO CASE OR UNKNOWN is specified, CASE_N returns NULL

case_spec

```
{ NO CASE [ { OR | , } UNKNOWN ] | UNKNOWN }
```

NO CASE

Evaluates to TRUE if every *conditional_expression* in the list evaluates to FALSE.

{ OR | , } UNKNOWN

The NO CASE OR UNKNOWN condition evaluates to TRUE if every *conditional_expression* in the list evaluates to FALSE, or if a conditional_expression evaluates to UNKNOWN and all prior conditions in the list evaluate to FALSE.

UNKNOWN

Evaluates to TRUE if a *conditional_expression* evaluates to UNKNOWN and all prior conditions in the list evaluate to FALSE.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for CASE_N are as follows.

Data Type	Format	Title
INTEGER	Default format for INTEGER	<CASE_N function>

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

CASE_N Usage Notes

Using CASE_N to Define Partitioned Primary Indexes

The primary index for a table or join index controls the distribution and retrieval of the data for that table or join index across the AMPs. If the primary index is a *partitioned* primary index (PPI), the data can be assigned to user-defined partitions on the AMPs.

To define a primary index for a table or join index, you specify the PRIMARY INDEX phrase in the CREATE TABLE or CREATE JOIN INDEX data definition statement. To define a partitioned primary index, you include the PARTITION BY phrase when you define the primary index.

The PARTITION BY phrase requires one or more partitioning expressions that determine the partition assignment of a row. You can use CASE_N to construct a partitioning expression such that a row with any value or NULL for the partitioning columns is assigned to some partition.

You can also use RANGE_N to construct a partitioning expression. For more information, see “RANGE_N”.

If the PARTITION BY phrase specifies a list of partitioning expressions, the PPI is a *multilevel* PPI, where each partition for a level is subpartitioned according to the next partitioning expression in the list. Unlike the partitioning expression for a single-level PPI, which can consist of any valid SQL expression (with some exceptions), each expression in the list of partitioning expressions for a multilevel PPI must be a CASE_N or RANGE_N function.

You cannot ADD or DROP partitioning expressions that are based on a CASE_N function. To modify a partitioning expression that is based on a CASE_N function, you must use the ALTER TABLE statement with the MODIFY PRIMARY INDEX option to redefine the entire PARTITION BY clause, and the table must be empty.

For more information, see “ALTER TABLE” in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Using CASE_N with CURRENT_DATE or CURRENT_TIMESTAMP in a PPI

You can define a partitioning expression that uses CASE_N with the built-in functions CURRENT_DATE or CURRENT_TIMESTAMP. Subsequently, you can use the ALTER TABLE TO CURRENT statement to re-partition the table data using a newly resolved current date or timestamp.

For more information, see “Rules and Guidelines for Optimizing the Reconciliation of CASE_N PPI Expressions Based On Updatable Current Date and Updatable Current Timestamp” in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Using CASE_N with Character Comparison

You can specify conditional expressions in the CASE_N function that compare CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data types. The following usage rules apply:

- A CASE_N partitioning expression can use character or graphic comparison except when the comparison involves KANJI1 or KANJISJIS columns or literal expressions.
- A CASE_N partitioning expression can use the UPPERCASE qualifier and the following functions: LOWER, UPPER, TRANSLATE, TRIM, VARGRAPHIC, INDEX, MINDEX, POSITION, TRANSLATE_CHK, CHAR2HEXINT.
- Any string literal referenced within a CASE_N expression must be less than 31,000 bytes.
- The order of character data used in evaluating the conditional expressions in a CASE_N function is determined by the session collation and case specificity of the expression.
    - If the expression is not part of a PPI, the current session collation is used.
    - If the expression is part of a PPI, evaluation is done using the session collation that was in effect when the table or join index was created, or when the partitioning was modified using the ALTER TABLE statement.
    - The case specificity of column references and literals is determined based on the session default, or an explicit CAST, or a specification in the CREATE TABLE statement when the table was created. The column can be explicitly assigned to be CASESPECIFIC or NOT CASESPECIFIC, and literal expressions can be CAST with these qualifiers.

      If not explicitly specified, the default of NOT CASESPECIFIC is used if Teradata session transaction semantics are in effect. If ANSI session transaction semantics are in effect, the default is CASESPECIFIC.

For example, if a conditional expression is a combination of NOT CASESPECIFIC expressions and a literal with no case specific qualifier (CASESPECIFIC, NOT CASESPECIFIC), the case specificity will be case specific in ANSI mode sessions and not case specific in Teradata mode sessions.

All character string comparisons involving graphic data are case specific.

- In character comparison operations (=, <, >, <=, >=, <>, BETWEEN, LIKE), if a string literal is shorter than the column data to which it is compared, the string literal is treated as if it is padded with a pad character specific to the character set (for example, a <space> character).

  Note that the pad character might not collate to the lowest code point in the collation. For a literal of length *n*, if the column value being compared precisely matches the literal for the first *n* characters, but contains a character that collates less than the pad character at position *n*+1, then the column value will collate less than the string literal.

Restrictions

If CASE_N is used in a PARTITION BY phrase, it:

- Can specify a maximum of 65533 conditions (unless it is part of a larger partitioning expression)
- Must not contain the system-derived columns PARTITION or PARTITION#L1 through PARTITION#L15
- Must not use Period data types, but can use the following:
  - BEGIN bound function for which input is a Period data type column and not a Period expression.
  - END bound function for which input is a Period data type column and not a Period expression.
  - IS [NOT] UNTIL_CHANGED.
  - IS [NOT] UNTIL_CLOSED.

If CASE_N is used in a partitioning expression for a multilevel PPI, it must define at least two partitions.

Note that partition elimination for queries is often limited to literal or using value equality conditions on the partitioning columns, and the Optimizer may not eliminate some partitions when it possibly could. Also, evaluating a complex CASE_N may be costly in terms of CPU cycles and the overhead of CASE_N may cause the table header to be excessively large.

Examples

Example: Defining the Partition to Which a Row is Assigned

Here is an example that uses CASE_N and the value of the totalorders column to define the partition to which a row is assigned:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER)
 PRIMARY INDEX (storeid, productid)
  PARTITION BY CASE_N(totalorders < 100, totalorders < 1000,
                      NO CASE, UNKNOWN);
```

In the example, CASE_N specifies four partitions to which a row can be assigned, based on the value of the totalorders column.

Partition Number	Condition
1	The value of the totalorders column is less than 100.
2	The value of the totalorders column is less than 1000, but greater than or equal to 100.
3	The value of the totalorders column is greater than or equal to 1000.
4	The totalorders column is NULL.

Example: Using CASE_N in a List of Partitioning Expressions that Define a Multilevel PPI

Here is an example that modifies “Example: Defining the Partition to Which a Row is Assigned” to use CASE_N in a list of partitioning expressions that define a multilevel PPI:

```
   CREATE TABLE orders
    (storeid INTEGER NOT NULL
    ,productid INTEGER NOT NULL
    ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
    ,totalorders INTEGER NOT NULL)
    PRIMARY INDEX (storeid, productid)
     PARTITION BY (CASE_N(totalorders < 100, totalorders < 1000,
                          NO CASE)
                  ,CASE_N(orderdate <= '2005-12-31', NO CASE) );
```

The example defines six partitions to which a row can be assigned. The first CASE_N expression defines three partitions based on the value of the totalorders column. The second CASE_N expression subdivides each of the three partitions into two partitions based on the value of the orderdate column.

Level 1 Partition Number	Level 2 Partition Number	Condition
1	1	The value of the totalorders column is less than 100 and the value of the orderdate column is less than or equal to '2005-12-31'.
	2	The value of the totalorders column is less than 100 and the value of the orderdate column is greater than '2005-12-31'.
2	1	The value of the totalorders column is less than 1000 but greater than or equal to 100, and the value of the orderdate column is less than or equal to '2005-12-31'.
	2	The value of the totalorders column is less than 1000 but greater than or equal to 100, and the value of the orderdate column is greater than '2005-12-31'.
3	1	The value of the totalorders column is greater than or equal to 1000 and the value of the orderdate column is less than or equal to '2005-12-31'.
	2	The value of the totalorders column is greater than or equal to 1000 and the value of the orderdate column is greater than '2005-12-31'.

Example: Showing the Count of Rows in Each Partition

The following example shows the count of rows in each partition if the orders table were to be partitioned using the CASE_N expression.

```
CREATE TABLE orders
 (orderkey INTEGER NOT NULL
 ,custkey INTEGER
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL)
 PRIMARY INDEX (orderkey);

INSERT INTO orders (1, 1, '1996-01-01');
INSERT INTO orders (2, 1, '1997-04-01');
```

The CASE_N expression in the following SELECT statement specifies three conditional expressions and the NO CASE condition.

```
SELECT COUNT(*),
       CASE_N(orderdate >= '1996-01-01' AND
              orderdate <= '1996-12-31' AND
              custkey <> 999999,
              orderdate >= '1997-01-01' AND
              orderdate <= '1997-12-31' AND
```

```
                 custkey <> 999999,
                 orderdate >= '1998-01-01' AND
                 orderdate <= '1998-12-31' AND
                 custkey <> 999999,
                 NO CASE
          ) AS Partition_Number
    FROM orders
    GROUP BY Partition_Number
    ORDER BY Partition_Number;
```

The results look like this:

```
       Count(*)  Partition_Number
    -----------  ----------------
              1                 1
              1                 2
```

Example: Creating a Table Partitioned with Orders Data

The following example creates a table partitioned with orders data for each quarter in 2008.

```
    CREATE TABLE Orders
      (O_orderkey INTEGER NOT NULL,
       O_custkey INTEGER,
       O_orderperiod PERIOD (DATE) NOT NULL,
       O_orderpriority CHAR (21),
       O_comment VARCHAR (79))
      PRIMARY INDEX (O_orderkey)
      PARTITION BY
         CASE_N (END (O_orderperiod) <= date'2008-03-31', /* First Quarter */
                 END (O_orderperiod) <= date'2008-06-30', /* Second Quarter */
                 END (O_orderperiod) <= date'2008-09-30', /* Third Quarter */
                 END (O_orderperiod) <= date'2008-12-31' /* Fourth Quarter */
                 );
```

The following SELECT statement scans two partitions and displays the details of the orders placed for the first two quarters.

```
SELECT *
FROM Orders
WHERE END (O_orderperiod) > date'2008-06-30';
```

Example: Verifying the Ending Bound of a Period Expression

The following example uses IS [NOT] UNTIL_CHANGED in the PPI partitioning expression to check whether or not the ending bound of a Period expression is UNTIL_CHANGED.

```
CREATE TABLE TESTUC
  (A  INTEGER,
   B  PERIOD (DATE),
   C INTEGER)
  PRIMARY INDEX (A)
  PARTITION BY
     CASE_N (END (b) IS UNTIL_CHANGED,
             END (b) IS NOT UNTIL_CHANGED, UNKNOWN);
```

Example: Verifying the Ending Bound of a Transaction Time Column

The following example uses IS [NOT] UNTIL_CLOSED in the PPI partitioning expression to check whether or not the ending bound of a transaction time column is UNTIL_CLOSED.

```
CREATE TABLE TESTUC
  (A  INTEGER,
   B  PERIOD (TIMESTAMP (6) WITH TIME ZONE) NOT NULL AS TRANSACTIONTIME,
   C INTEGER)
  PRIMARY INDEX (A)
  PARTITION BY
     CASE_N (END (b) IS UNTIL_CHANGED,
             END (b) IS NOT UNTIL_CHANGED, UNKNOWN);
```

Example: Viewing Results for FALSE Conditions

In this example, the session collation is ASCII.

```
CASE_N (a<'b', a>='ba' and a<'dogg' and b<>'cow', c<>'boy', NO CASE OR UNKNOWN)
```

The following table shows the result value returned by the above CASE_N function given the specified values for *a*, *b*, and *c*. *x* and *y*represent any value or NULL. The value 4 is returned when all the conditions are FALSE, or a condition is UNKNOWN with all preceding conditions evaluating to FALSE.

a	b	c	Result
'a'	*x*	*y*	1

a	b	c	Result
'boy'	'girl'	*y*	2
'boy'	NULL	*y*	4
'boy'	'cow'	'man'	3
'boy'	'cow'	'boy'	4
'dog'	'ball'	*y*	2
'dogg'	*x*	NULL	4
'dogg'	*x*	'man'	3
'egg'	*x*	'boy'	4
'egg'	*x*	NULL	4
'egg'	*x*	'girl'	3

Example: Viewing Results for UNKNOWN Conditions

In this example, the session collation is ASCII.

```
CASE_N (a<'b', a>='ba' and a<'dogg' and b<>'cow', c<>'boy', UNKNOWN)
```

The following table shows the result value returned by the above CASE_N function given the specified values for *a*, *b*, and *c*. The *x* and *y*represent any value or NULL. The value 4 is returned if a condition is UNKNOWN with all preceding conditions evaluating to FALSE. NULL is returned if all the conditions are false.

a	b	c	Result
'a'	*x*	*y*	1
'boy'	'girl'	*y*	2
'boy'	NULL	*y*	4
'boy'	'cow'	'man'	3
'boy'	'cow'	'boy'	NULL
'dog'	'ball'	*y*	2
'dogg'	*x*	NULL	4
'dogg'	*x*	'man'	3
'egg'	*NULL*	'boy'	NULL

a	b	c	Result
'egg'	*x*	'boy'	NULL
'egg'	*x*	NULL	4
'egg'	*x*	'girl'	3

Example: Defining Partitions Based on the Value of a

In this example, the session collation is ASCII when submitting the CREATE TABLE statement, and the pad character is <space>. The example defines two partitions (numbered 1 and 2) based on the value of *a*:

- The value of *a* is between 'a   ' (a followed by 9 spaces) and 'b   '.
- The value of *a* is between 'b   ' and 'c   '.

```
CREATE SET TABLE t2
   (a VARCHAR(10) CHARACTER SET UNICODE NOT CASESPECIFIC,
    b INTEGER)
PRIMARY INDEX (a)
PARTITION BY CASE_N(a BETWEEN 'a' AND 'b', a BETWEEN 'b' AND 'c');
```

The following INSERT statement inserts a character string consisting of a single <tab> character between the 'b' and '1'.

```
INSERT t2 ('b        1', 1);
```

The following INSERT statement inserts a character string consisting of a single <space> character between the 'b' and '1'.

```
INSERT t2 ('b 1', 2);
```

The following SELECT statement shows the result of the INSERT statements. Since the <tab> character has a lower code point than the <space> character, the first string inserted maps to partition 1.

```
    SELECT PARTITION, a, b FROM t2 ORDER BY 1;
    *** Query completed. 2 rows found. 3 columns returned.
    *** Total elapsed time was 1 second.
    PARTITION  a          b
 -----------  ------ -----
           1  b 1        1   (string contains single <tab> character)
           2  b 1        2   (string contains single <space> character)
```

Related Information

- For more information about PPI properties and performance considerations and PPI considerations and capacity planning, see *Teradata Vantage™ - Database Design*, B035-1094.
- For more information about the specification of a PPI for a table, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information about the specification of a PPI for a join index, see CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information about the modification of the partitioning of the primary index for a table, see ALTER TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information about the reconciliation of the partitioning based on newly resolved CURRENT_DATE and CURRENT_TIMESTAMP values, see ALTER TABLE TO CURRENT in*Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

CEILING

Returns the smallest integer value that is not less than the input argument.

CEILING is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

CEILING Function Syntax

```
[TD_SYSFNLIB.] { CEILING | CEIL } (arg)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

arg

Expression with one of the following data types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- FLOAT/REAL/DOUBLE PRECISION
- DECIMAL/NUMERIC
- NUMBER

IF *arg* is...	THEN CEILING returns...
a non-exact number	the next integer value that is greater than *arg.*
an exact number	the input argument *arg*.
NULL	NULL.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result type is the same data type as that of the numeric input argument.

If the input argument is defined as a DECIMAL/NUMERIC with a precision less than 38, the return DECIMAL/NUMERIC value will have its precision increased by 1. For example, if DECIMAL(6,4) is passed in, it will be increased and returned as a DECIMAL(7,4). If the precision is 38, the scale will be reduced by 1 unless the scale is 0. For example, a DECIMAL(38,38) results in a return data type of DECIMAL(38,37).

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Examples

Example: Querying SELECT CEILING(157E-1);

The following query returns the FLOAT value 16E0, since 16 is the smallest integer that is not less than the FLOAT value 15.7E0.

```
SELECT CEILING(157E-1);
```

Example: Querying SELECT CEILING(15.7);

The following query returns a DECIMAL value of 16.0 since 16 is the smallest integer that is not less than the DECIMAL literal 15.7.

```
SELECT CEILING(15.7);
```

Example: Querying SELECT CEILING(-12.3);

The following query returns a DECIMAL value of -12.0 since -12 is the smallest integer that is not less than the DECIMAL literal -12.3.

```
SELECT CEILING(-12.3);
```

Example: Querying SELECT CEIL(CAST(9.99 AS DECIMAL(3,2)));

The following query returns the value 10.00 with a data type of DECIMAL(4,2), since 10 is the smallest integer that is not less than 9.99. Note that the precision of the return value is increased by 1.

```
SELECT CEIL( CAST(9.99 AS DECIMAL(3,2)) );
```

DEGREES/RADIANS

DEGREES takes a value specified in radians and converts it to degrees.

RADIANS takes a value specified in degrees and converts it to radians.

DEGREES/RADIANS Function Syntax

```
{ DEGREES | RADIANS } (arg)
```

Syntax Elements

DEGREES

Converts value specified in radians to degrees.

RADIANS

Converts value specified in degrees to radians.

arg

A numeric argument.

- If the function is DEGREES, the *arg* is interpreted as an angle in radians.
- If the function is RADIANS, the *arg* is interpreted as an angle in degrees.

Result Format Types

IF *arg* is...	THEN result format is default format for ...
numeric	resulting data type
character	FLOAT
UDT	predefined type to which UDT is implicitly cast

Default Result Titles

Function	Title
DEGREES(*arg*)	(5.72957795130823E001**arg*)
RADIANS(*arg*)	(1.74532925199433E-002**arg*)

Argument Types and Rules

If the argument is not numeric, it is converted to a numeric value, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If *arg* is a character string, it is converted to a numeric value of the FLOAT data type.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DateTime
  - Interval

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Implicit type conversion of UDTs for system operators and functions, including DEGREES and RADIANS, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Neither DEGREES nor RADIANS can be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Format

Here are the result type and format of DEGREES and RADIANS.

- If the operand is numeric, the format is the default format for the resulting data type.

- If the operand is character, the format is the default format for FLOAT.
- If the operand is a UDT, the format is the default format for the predefined type to which the UDT is implicitly cast.

Note:

The NULL keyword has a data type of INTEGER.

For information on data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Usage Notes

DEGREES and RADIANS are useful when working with trigonometric functions such as SIN and COS, which expect arguments to be specified in radians, and inverse trigonometric functions such as ASIN and ACOS, which return values specified in radians.

Examples: Representative DEGREES/RADIANS Function Expressions

Representative DEGREES and RADIANS function expressions and the results are as follows.

Expression	Result
SIN(RADIANS(60.0))	8.66025403784439E-001
DEGREES(1.0)	5.72957795130823E 001

EXP

Raises *e* (the base of natural logarithms) to the power of the argument, where *e* = 2.71828182845905.

EXP Function Syntax

```
EXP (arg)
```

Syntax Elements

arg

A numeric argument.

Argument Types and Rules

If *arg* is not FLOAT, the value is converted to FLOAT, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
    - Numeric
    - Character
    - DATE

    To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.
- Implicit type conversion of UDTs for system operators and functions, including EXP, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

EXP cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The following table lists the default attributes for the result of EXP(*arg*).

Data Type	Format	Title
FLOAT	Default format for the resulting data type	EXP(*arg*)

Usage Notes

Executing EXP may sometimes result in a numeric overflow error.

Examples: Representative EXP Arithmetic Function Expressions

Representative EXP arithmetic function expressions and the results are as follows.

Expression	Result
EXP(1)	2.71828182845905E+000
EXP(0)	1.00000000000000E+000

Related Information

- For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For details on setting the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE, see *Teradata Vantage™ - Database Utilities*, B035-1102.

FLOOR

Returns the largest integer equal to or less than the input argument.

FLOOR is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

FLOOR Function Syntax

```
[TD_SYSFNLIB.] FLOOR (arg)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

arg

A numeric argument.

IF *arg* is...	THEN FLOOR returns...
a non-exact number	the next largest integer that is equal to or less than *arg.*
an exact number	the input argument *arg*.
NULL	NULL.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, FLOAT/REAL/DOUBLE PRECISION, DECIMAL/NUMERIC, or NUMBER

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result type is the same data type as that of the numeric input argument.

If the input argument is defined as a DECIMAL/NUMERIC with a precision less than 38, the return DECIMAL/NUMERIC value will have its precision increased by 1. For example, if DECIMAL(6,4) were passed in, it would be increased and returned as a DECIMAL(7,4). If the precision is 38, the scale will be reduced by 1 unless the scale is 0. For example, a DECIMAL(38,38) results in a return data type of DECIMAL(38,37).

The default title for FLOOR is FLOOR(*arg*).

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Examples: Using the FLOOR Function

Example: Querying SELECT FLOOR (136E-1);

The following query returns the FLOAT value 13E0, since 13 is the largest integer that is less than the FLOAT value 13.6E0.

```
SELECT FLOOR (136E-1);
```

Example: Querying SELECT FLOOR(-6.5);

The following query returns a DECIMAL value of -7.0 since -7 is the largest integer that is less than the DECIMAL literal -6.5.

```
SELECT FLOOR(-6.5);
```

Example: Querying SELECT FLOOR (CAST(-6.5 AS DECIMAL(2,1)));

The following query returns a value of -7.0 with a data type of DECIMAL(3,1), since -7 is the largest integer less than DECIMAL -6.5. Note that the precision of the return value is increased by 1.

```
SELECT FLOOR (CAST(-6.5 AS DECIMAL(2,1)));
```

Related Information

- Embedded Services System Functions
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

HYPERBOLIC

Performs the hyperbolic or inverse hyperbolic function of an argument.

HYPERBOLIC Function Syntax

```
{ COSH | SINH | TANH | ACOSH | ASINH | ATANH } (arg)
```

Syntax Elements

arg

Any real number.

Function	Result
COSH(*arg*)	Hyperbolic cosine of *arg*.
SINH(*arg*)	Hyperbolic sine of *arg*.
TANH(*arg*)	Hyperbolic tangent of *arg*.
ACOSH(*arg*)	Inverse hyperbolic cosine of *arg*. The inverse hyperbolic cosine is the value whose hyperbolic cosine is a number so that: acosh(cosh(arg)) = arg
ASINH(*arg*)	Inverse hyperbolic sine of *arg*. The inverse hyperbolic sine is the value whose hyperbolic sine is a number so that: asinh(sinh(arg)) = arg
ATANH(*arg*)	Inverse hyperbolic tangent of *arg*. The inverse hyperbolic tangent is the value whose hyperbolic tangent is a number so that: atanh(tanh(arg)) = arg

Argument Types and Rules

If *arg* is not FLOAT, it is converted to a FLOAT value, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
    - Numeric
    - Character
    - DATE

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Implicit type conversion of UDTs for system operators and functions, including hyperbolic and inverse hyperbolic functions, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Hyperbolic and inverse hyperbolic functions cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

  Examples: Representative Hyperbolic and Inverse Hyperbolic Function Expressions

The following are representative hyperbolic and inverse hyperbolic function expressions and results.

Expression	Result
COSH(EXP(1))	7.61012513866229E 000
SINH(1)	1.17520119364380E 000
TANH(0)	0.00000000000000E 000
ACOSH(3)	1.76274717403909E 000
ASINH(LOG(0.1))	-8.81373587019543E -001
ATANH(LN(0.5))	-8.53988047997524E -001

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

Here are the default attributes for the result of hyperbolic and inverse hyperbolic functions.

Data Type	Format	Title
FLOAT	Default format for FLOAT	Hyperbolic Cos(arg) Hyperbolic Sin(arg) Hyperbolic Tan(arg) Hyperbolic ArcCos(arg) Hyperbolic ArcSin(arg) Hyperbolic ArcTan(arg)

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Related Information

- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

LN

Computes the natural logarithm of the argument.

LN Function Syntax

```
LN (arg)
```

Syntax Elements

arg

A nonzero, positive numeric argument.

Argument Types and Rules

If *arg* is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
    - Numeric
    - Character
    - DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

- Implicit type conversion of UDTs for system operators and functions, including LN, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

LN cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for LN(*arg*) are as follows.

Data Type	Format	Title
FLOAT	Default format for FLOAT	LN(arg)

Examples: Representative LN Arithmetic Function Expressions

Representative LN arithmetic function expressions and the results are as follows.

Expression	Result
LN(2.71828182845905)	1.00000000000000E+000
LN(0)	Error

Related Information

- For more information on implicit type conversion, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For details on setting the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For information on default data type formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

LOG

Computes the base 10 logarithm of an argument.

LOG Function Syntax

```
LOG (arg)
```

Syntax Elements

arg

A nonzero, positive numeric argument.

Argument Types and Rules

If *arg* is not FLOAT, the value is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
    - Numeric
    - Character
    - DATE

    To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Implicit type conversion of UDTs for system operators and functions, including LOG, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. See *Teradata Vantage™ - Database Utilities*, B035-1102.

LOG cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for LOG(*arg*) are as follows.

Data Type	Format	Title
FLOAT	Default format for FLOAT	LOG(arg)

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*.

Example: Representative LOG Arithmetic Function Expressions

Representative LOG arithmetic function expressions and the results are as follows.

Expression	Result
LOG(50)	1.69897000433602E+000
LOG(100)	2.00000000000000E+000

Related Information

- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

MOD

Returns the remainder (modulus) of *expr1* divided by *expr2*.

MOD Function Syntax

```
MOD (expr1, expr2)
```

Syntax Elements

expr1

Numeric argument that is the dividend.

expr2

A numeric argument that is the divisor.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- FLOAT/REAL/DOUBLE PRECISION
- DECIMAL/NUMERIC
- NUMBER

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL-2011 standard.

Result Type and Attributes

The results take the sign of the dividend:

MOD (-17, 4) = -1
MOD (-17, -4) = -1
MOD (17, -4) = 1
MOD (17, 4) = 1

Example: Using MOD Arithmetic Function Expression

Example usage of the MOD function:

```
SELECT MOD(17,4);
```

```
SELECT MOD(-17,4);
```

Related Information

This function has the same functionality as arithmetic operator MOD. For more information, see Arithmetic, Trigonometric, Hyperbolic Operators/Functions.

NULLIFZERO

Converts data from zero to null to avoid problems with division by zero.

NULLIFZERO Function Syntax

```
NULLIFZERO (arg)
```

Syntax Elements

arg

A numeric argument, or an argument that can be converted to a numeric argument based on implicit type conversion rules.

IF the value of *arg* is ...	THEN NULLIFZERO returns ...
nonzero	the value of the numeric argument
null or zero	NULL

Argument Types and Rules

If *arg* is not numeric, the value is converted to a numeric value, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If *arg* is a character string, the value is converted to a numeric value of FLOAT data type.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE
  - Interval

    To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Implicit type conversion of UDTs for system operators and functions, including NULLIFZERO, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. See *Teradata Vantage™ - Database Utilities*, B035-1102.

NULLIFZERO cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB

- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

The ANSI form of this function is the CASE shorthand expression NULLIF. For more information, see NULLIF Expression.

Result Type and Attributes

Here are the default attributes for the result of NULLIFZERO.

- If *arg* is numeric, the data type is the same types as *arg* and the format is the same format as *arg*.
- If *arg* is character, the data type is FLOAT and the default format is FLOAT.
- If *arg* is a UDT, the data type is the type of which the UDT is implicitly cast and the format is the format of the data type to which the UDF is implicitly cast.

Note:

The NULL keyword has a data type of INTEGER.

For information on data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Examples

Example: Returned Expression Errors for NULLIFZERO

The following expressions return an error if the value of *x* or *expression* is zero.

```
6 / x
6 / expression
```

On the other hand, the following expressions return null, which is not an error because there is no violation of the divide by zero rule.

```
6 / NULLIFZERO(x)
6 / NULLIFZERO(expression)
```

Example: Returned Request Errors for NULLIFZERO

The following request returns a null in the second column because the HCap field value for Newman is zero. In BTEQ (field mode) this appears as a ‘?’.

```
SELECT empno, NULLIFZERO(hcap)
FROM employee
WHERE empno = 10019 ;
```

Related Information

- COALESCE Expression
- NULLIF Expression
- ZEROIFNULL
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

POWER

Returns *base_value* raised to the power of *exponent_value*.

POWER is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

POWER Function Syntax

```
[SYSLIB.] POWER (base_value, exponent_value)
```

Syntax Elements

SYSLIB.

Name of the database where the function is located.

base_value

A numeric argument.

If *base_value* is negative, exponent_value must be an integer.

If any input argument is NULL, the function returns NULL.

exponent_value

A numeric argument.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

If either of the input arguments is a FLOAT type, the result data type is FLOAT. Otherwise, the result data type is NUMBER.

Examples: Querying Returns for the Power of *exponent_value*

Example 1: Querying POWER(2, 3)

The following query returns the result 8.0.

```
SELECT POWER(2, 3);
```

Example 2: Querying POWER(2, -3)

The following query returns the result 0.125.

```
SELECT POWER(2, -3);
```

Example 3: Querying POWER(2.2, 3)

The following query returns the result 10.648.

```
SELECT POWER(2.2, 3);
```

Example 4: Querying POWER(-2.2, 3.1)

The following query returns an error because the base value is negative and the exponent value is not an integer.

```
SELECT POWER(-2.2, 3.1);
```

Related Information

- Embedded Services System Functions
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

RANDOM

Returns a random integer number for each row of the results table.

Computation

RANDOM uses the linear congruential algorithm and 48-bit integer arithmetic.

The algorithm works by generating a sequence of 48-bit integer values, Xi.

RANDOM Function Syntax

```
RANDOM ( lower_bound, upper_bound )
```

Syntax Elements

lower_bound

An integer literal to define the lower bound on the closed interval over which a random number is to be selected.

The limits for lower_bound range from -2147483648 to 2147483647.

lower_bound must be less than or equal to *upper_bound*.

upper_bound

An integer literal to define the upper bound on the closed interval over which a random number is to be selected.

The limits for upper_bound range from -2147483648 to 2147483647.

upper_bound must be greater than or equal to *lower_bound*.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for RANDOM(x,y) are as follows.

Data Type	Format	Title
INTEGER	Default format for INTEGER	Random(x,y)

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

RANDOM Usage Notes

Restrictions

The following rules and restrictions apply to the use of the RANDOM function.

- RANDOM can only be called in one of the following SELECT query clauses:
  - WHERE
  - GROUP BY
  - ORDER BY
  - HAVING/QUALIFY
- RANDOM cannot be referenced by position in a GROUP BY or ORDER BY clause.
- RANDOM cannot be nested inside aggregate or ordered analytical functions.
- RANDOM cannot be used in the expression list of an INSERT statement to create a primary index or partitioning column value.

  For example:

  ```
  INSERT t1 (RANDOM(1,10),...)
  ```

  RANDOM causes an error to be reported in this case if the first column in the table is a primary index or partitioning column.

Multiple RANDOM Calls Within a SELECT List

You can call RANDOM any number of times in the SELECT list, for example:

```
SELECT RANDOM(1,100), RANDOM(1,100);
```

Each call defines a new random value.

Using RANDOM as a Condition on an Index

Because the RANDOM function is evaluated for each selected row, a condition on an index column that includes the RANDOM function results in an all-AMP operation.

For example, consider the following table definition:

```
CREATE TABLE t1
   (c1 INTEGER
```

```
   ,c2 VARCHAR(9))
PRIMARY INDEX ( c1 );
```

The following SELECT statement results in an all-AMP operation:

```
SELECT *
FROM t1
WHERE c1 = RANDOM(1,12);
```

Example: Returning Random Integer Numbers as Results

Suppose you have a table named sales_table with the following subset of columns.

Store_ID	Product_ID	Sales
1003	C	20000
1002	C	35000
1001	C	60000
1002	D	50000
1003	D	50000
1001	D	35000
1001	A	100000
1002	A	40000
1001	E	30000

The following SELECT statement returns a random number between 1 and 3, inclusive, for each row in the results table.

```
SELECT store_id, product_id, sales, RANDOM(1,3)
FROM sales_table;
```

The results table might look like this.

Store_ID	Product_ID	Sales	RANDOM(1,3)
1003	C	20000	1
1002	C	35000	2
1001	C	60000	2
1002	D	50000	3

Store_ID	Product_ID	Sales	RANDOM(1,3)
1003	D	50000	2
1001	D	35000	3
1001	A	100000	2
1002	A	40000	1
1001	E	30000	2

Related Information

- *Teradata Vantage™ - Data Types and Literals*, B035-1143

RANGE_N

Evaluates an expression and maps the result into one of a list of specified ranges and returns the position of the range in the list.

Range

A range is defined by a starting boundary and an optional ending boundary. If an ending boundary is not specified, the range is defined by its starting boundary, inclusively, up to but not including the starting boundary of the next range.

The list of ranges must specify ranges in increasing order, where the ending boundary of a range is less than the starting boundary of the next range.

RANGE_N Function Syntax

```
RANGE_N (test_expression BETWEEN range_expression [, range_spec ])
```

Syntax Elements

test_expression

An expression that results in a BYTEINT, SMALLINT, INTEGER, DATE, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type.

RANGE_N evaluates *test_expression* and determines whether the result is within a range in the list of ranges. The position of the first range is one and the positions of subsequent ranges increment by one up to *n*, where *n* is the total number of ranges.

IF …	THEN …
the result of *test_expression* is within a range	RANGE_N returns the position of the range.
the result of *test_expression* is NULL	If RANGE_N does not specify one of the following: • BETWEEN * AND * • UNKNOWN • NO RANGE OR UNKNOWN RANGE_N returns NULL. If RANGE_N specifies the range BETWEEN * AND *, RANGE_N returns 1, regardless of whether NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN is specified. If RANGE_N does not specify the range BETWEEN * AND * and • If NO RANGE OR UNKNOWN is specified, RANGE_N returns n + 1. • If UNKNOWN is specified and NO RANGE is not specified, RANGE_N returns *n* + 1. • NO RANGE and UNKNOWN are specified, RANGE_N returns *n* + 2.
test_expression is outside all the ranges in the list	If NO RANGE or NO RANGE OR UNKNOWN is specified, RANGE_N returns *n* + 1. If neither NO RANGE nor NO RANGE OR UNKNOWN is specified, RANGE_N returns NULL.

range_expression

```
{ range_expr_1 | range_expr_2 | range_list }
```

range_spec

```
{ NO RANGE [ { OR | , } UNKNOWN ] | UNKNOWN }
```

range_expr_1

```
start_expression AND { end_expression | * } [ EACH range_size ]
```

range_expr_2

```
* AND { end_expression | * }
```

range_list

```
{ range_expr_3 | * [ AND end_expression ] }
    [, range_expr_3 [,...] ] , range_expr_1
```

NO RANGE

A range to handle a test_expression that does not map into any of the specified ranges.

OR UNKNOWN

NO RANGE OR UNKNOWN handles a test_expression that does not map into any of the specified ranges, or a test_expression that evaluates to NULL when RANGE_N does not specify the range BETWEEN * AND *.

UNKNOWN

Handles a test_expression that evaluates to NULL when RANGE_N does not specify the range BETWEEN * AND *.

start_expression

A literal or literal expression that defines the starting boundary of a range.

The data type of *start_expression* must be the same as the data type of *test_expression*, or must be implicitly cast to the same data type as *test_expression*.

If an ending boundary is not specified, the range is defined by its starting boundary (and this starting boundary is included in this range), up to but not including the starting boundary of the next range.

Use an asterisk (*) for the starting boundary of the first range in the list to indicate the lowest possible value (all values and NULL are greater than a starting boundary specified as an asterisk). An asterisk is compatible with any data type.

end_expression

A literal or literal expression that defines the ending boundary of a range.

The data type of *end_expression* must be the same as the data type of *test_expression*, or must be implicitly cast to the same data type as *test_expression*.

The last range in the list must specify an ending boundary. For all other ranges, if an ending boundary is not specified, the range is defined by its starting boundary (and this starting boundary is included in this range), up to but not including the starting boundary of the next range.

Use an asterisk (*) for the ending boundary of the last range in the list to indicate the highest possible value (all values and NULL are less than an ending boundary specified as an asterisk).

range_size

A literal or literal expression with a value greater than zero.

A range that specifies an EACH phrase is equivalent to a series of ranges, where the first range in the series starts at *start_expression*, and subsequent ranges start at *start_expression* + (*range_size* * *n*), where *n* starts at one and increments by one while *start_expression* + (*range_size* * *n*) is less than or equal to *end_expression*, or less than the next *start_expression* in the list of ranges.

For DATE types, the calculation of valid dates in subsequent ranges uses ADD_MONTHS instead of the + arithmetic operator.

The data type of *range_size* must be compatible for adding to *test_expression*.

Note:

If the data type of test_expression is a character type (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC), you cannot specify the EACH phrase.

range_expr_3

```
start_expression [ AND end_expression ] [ EACH range_size ]
```

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for RANGE_N are as follows.

Data Type	Format	Title
INTEGER	Default format of the INTEGER data type	<RANGE_N function>

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

RANGE_N Usage Notes

Restrictions

If RANGE_N appears in a PARTITION BY phrase, it:

- Can specify a maximum of 65,533 ranges (unless it is part of a larger partitioning expression)
- Must not contain the system-derived columns PARTITION or PARTITION#L1 through PARTITION#L15
- Must not use Period data types, but can use the BEGIN or END bound functions on a Period data type column when they result in a DATE data type.

If RANGE_N is used in a partitioning expression for a multilevel PPI, it must define at least two partitions.

If RANGE_N specifies CURRENT_DATE or CURRENT_TIMESTAMP in a partitioning expression, you cannot use ALTER TABLE to add or drop ranges for the table. You must use the ALTER TABLE TO CURRENT statement to achieve this function.

Using RANGE_N to Define Partitioned Primary Indexes

The primary index for a table or join index controls the distribution of the data for that table or join index across the AMPs, as well as its retrieval. If the primary index is a partitioned primary index (PPI), the data can be assigned to user-defined partitions on the AMPs.

To define a primary index for a table or join index, you specify the PRIMARY INDEX phrase in the CREATE TABLE or CREATE JOIN INDEX data definition statement. To define a partitioned primary index, you include the PARTITION BY phrase when you define the primary index.

The PARTITION BY phrase requires one or more partitioning expressions that determine the partition assignment of a row. You can use RANGE_N to construct a partitioning expression such that a row with any value or NULL for the partitioning columns is assigned to some partition.

You can also use CASE_N to construct a partitioning expression. For more information, see CASE_N.

If the PARTITION BY phrase specifies a list of partitioning expressions, the PPI is a multilevel PPI, where each partition for a level is subpartitioned according to the next partitioning expression in the list. Unlike the partitioning expression for a single-level PPI, which can consist of any valid SQL expression (with some exceptions), each expression in the list of partitioning expressions for a multilevel PPI must be a CASE_N or RANGE_N function.

Using RANGE_N with CURRENT_DATE or CURRENT_TIMESTAMP in a PPI

You can define a partitioning expression that uses RANGE_N with the built-in functions CURRENT_DATE or CURRENT_TIMESTAMP. Use of CURRENT_DATE or CURRENT_TIMESTAMP in a partitioning expression is most appropriate when the data must be partitioned as one or more current partitions and one or more history partitions where the current and history partitions are based on the resolved CURRENT_DATE or CURRENT_TIMESTAMP in the partitioning expression. This allows you to periodically reconcile the table to move older data from the current partition into one or more history partitions using the ALTER TABLE TO CURRENT statement instead of redefining the partitioning using explicit dates which must be determined each time the ALTER TABLE DROP/ADD RANGE is done.

For more information, see “Rules and Guidelines for Optimizing the Reconciliation of RANGE_N PPI Expressions Based On Updatable Current Date and Updatable Current Timestamp” in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Using RANGE_N with Character Data

You can specify character expressions (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) as the *test_expression* and/or the range boundaries in a RANGE_N function. The following usage rules apply:

- A RANGE_N partitioning expression can use the UPPERCASE qualifier and the following functions: LOWER, UPPER, TRANSLATE, TRIM, VARGRAPHIC, INDEX, MINDEX, POSITION, TRANSLATE_CHK, CHAR2HEXINT.
- If *test_expression* is a character data type, you cannot specify the EACH phrase.
- Any string literal referenced within a RANGE_N expression must be less than 31,000 bytes.
- If *test_expression* is a character data type, and the length of any of the range boundaries (minus trailing pad characters) is greater than the length of *test_expression*, an error is returned.
- For character RANGE_N partitioning, the increasing order of ranges is determined by the session collation and case specificity of the *test_expression*. If the *test_expression* is a combination of NOT CASESPECIFIC expressions and a literal with no case specific qualifier (CASESPECIFIC, NOT CASESPECIFIC), the case specificity will be case specific in ANSI mode sessions and not case specific in Teradata mode sessions.

  Note:

  All character string comparisons involving graphic data are case specific.

- An error is returned if any of the specified ranges are defined with null boundaries, are not increasing, or overlap. For character test values, increasing order is determined by the session collation and case specificity of the *test_expression*.
  - The case sensitivity of column references and literals is determined based on the session default, or an explicit CAST, or a specification in the CREATE TABLE statement when the table was created. The column can be explicitly assigned to be CASESPECIFIC or NOT CASESPECIFIC, and constant expressions can be CAST with these qualifiers.

    If not explicitly specified, the default of NOT CASESPECIFIC is used if Teradata session transaction semantics are in effect. If ANSI session transaction semantics are in effect, the default is CASESPECIFIC.

For example, if a conditional expression is a combination of NOT CASESPECIFIC expressions and a constant with no case sensitivity qualifier (CASESPECIFIC, NOT CASESPECIFIC), the case sensitivity will be case sensitive in ANSI mode sessions and case blind in Teradata mode sessions.

All character string comparisons involving graphic data are case sensitive.

- In character comparison operations (=, <, >, <=, >=, <>, BETWEEN, LIKE), if a string literal is shorter than the column data to which it is compared, the string literal is treated as if it is padded with a pad character specific to the character set (for example, a <space> character). Therefore, if a character *test_expression* is defined with a longer length than a character range boundary, comparison of the *test _expression* to that range boundary will behave as if the range boundary is padded with pad characters.

  Note that the pad character might not collate to the lowest code point in the collation. For a range boundary of length *n*, if the *test_expression* precisely matches that range boundary for the first *n* characters, but contains a character that collates less than the pad character at position *n* +1, then the *test_expression* will collate less than the range boundary. See Examples.

Using a UDT as the Test Expression

The *test_expression* should not be an expression that results in a UDT data type. An error is reported if this occurs when RANGE_N is used to define a PPI. If RANGE_N is not used to define a PPI, you should explicitly cast the expression so that it is BYTEINT, SMALLINT, INTEGER, DATE, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC instead of depending upon any implicit conversions.

Examples

Example 1: Specifying Four Partitions to which a Row Can Be Assigned

Here is an example that uses RANGE_N and the value of the totalorders column to define the partition to which a row is assigned:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER)
 PRIMARY INDEX (storeid, productid)
  PARTITION BY RANGE_N(totalorders BETWEEN *, 100, 1000 AND *,
                       UNKNOWN);
```

In the example, RANGE_N specifies four partitions to which a row can be assigned, based on the value of the totalorders column:

Partition Number	Condition
1	The value of the totalorders column is less than 100.
2	The value of the totalorders column is less than 1000, but greater than or equal to 100.
3	The value of the totalorders column is greater than or equal to 1000.
4	The totalorders column is NULL, so the range is UNKNOWN.

Example 2: Using RANGE_N in a List of Partitioning Expressions

Here is an example that modifies “Examples” to use RANGE_N in a list of partitioning expressions that define a multilevel PPI:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER NOT NULL)
 PRIMARY INDEX (storeid, productid)
  PARTITION BY (RANGE_N(totalorders BETWEEN *, 100, 1000 AND *)
               ,RANGE_N(orderdate BETWEEN *, '2005-12-31' AND *) );
```

The example defines six partitions to which a row can be assigned. The first RANGE_N expression defines three partitions based on the value of the totalorders column. The second RANGE_N expression subdivides each of the three partitions into two partitions based on the value of the orderdate column.

Level 1 Partition Number	Level 2 Partition Number	Condition
1	1	The value of the totalorders column is less than 100 and the value of the orderdate column is less than '2005-12-31'.
	2	The value of the totalorders column is less than 100 and the value of the orderdate column is greater than or equal to '2005-12-31'.
2	1	The value of the totalorders column is less than 1000 but greater than or equal to 100, and the value of the orderdate column is less than '2005-12-31'.
	2	The value of the totalorders column is less than 1000 but greater than or equal to 100, and the value of the orderdate column is greater than or equal to '2005-12-31'.
3	1	The value of the totalorders column is greater than or equal to 1000 and the value of the orderdate column is less than '2005-12-31'.

Level 1 Partition Number	Level 2 Partition Number	Condition
	2	The value of the totalorders column is greater than or equal to 1000 and the value of the orderdate column is greater than or equal to '2005-12-31'.

Example 3: Defining a Partitioned Primary Index that Specifies One Partition

Here is an example that defines a partitioned primary index that specifies one partition to which rows are assigned, for any value of the totalorders column, including NULL:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER)
 PRIMARY INDEX (storeid, productid)
  PARTITION BY RANGE_N(totalorders BETWEEN * AND *);
```

Example 4: Counting Rows in Each Partition if the Table is Partitioned Using the RANGE_N Expression

The following example shows the count of rows in each partition if the table were to be partitioned using the RANGE_N expression.

```
CREATE TABLE orders
 (orderkey INTEGER NOT NULL
 ,custkey INTEGER
 ,orderdate DATE FORMAT 'yyyy-mm-dd')
 PRIMARY INDEX (orderkey);

INSERT INTO orders (1,  100, '1998-01-01');
INSERT INTO orders (2,  100, '1998-04-01');
INSERT INTO orders (3,  109, '1998-04-01');
INSERT INTO orders (4,  101, '1998-04-10');
INSERT INTO orders (5,  100, '1998-07-01');
INSERT INTO orders (6,  109, '1998-07-10');
INSERT INTO orders (7,  101, '1998-08-01');
INSERT INTO orders (8,  101, '1998-12-01');
INSERT INTO orders (9,  111, '1999-01-01');
INSERT INTO orders (10, 111, NULL);
```

The RANGE_N expression in the following SELECT statement uses the EACH phrase to define a series of 12 ranges, where the first range starts at '1998-01-01' and the ranges that follow have starting boundaries that increment sequentially by one month intervals.

```
SELECT COUNT(*),
       RANGE_N(orderdate
               BETWEEN DATE '1998-01-01' AND DATE '1998-12-31'
               EACH INTERVAL '1' MONTH
       ) AS Partition_Number
FROM orders
GROUP BY Partition_Number
ORDER BY Partition_Number;
```

The results look like this:

```
   Count(*) Partition_Number
----------- ----------------
          2                ?
          1                1
          3                4
          2                7
          1                8
          1               12
```

Example 5: Table Partitioning Using a RANGE_N Expression

The following example creates a table with partitioning defined using a RANGE_N expression involving the END bound function. The table creates 10 partitions where each partition represents the sales history for one year.

```
CREATE TABLE SalesHistory
  (product_code CHAR (8),
   quantity_sold INTEGER,
   transaction_period PERIOD (DATE))
  PRIMARY INDEX (product_code)
  PARTITION BY
     RANGE_N (END (transaction_period) BETWEEN date'2006-01-01'
              AND date '2015-12-31' EACH INTERVAL'1' YEAR);
```

The following SELECT statement scans five partitions of the sales history before the year 2010.

```
SELECT *
FROM SalesHistory
WHERE transaction_period < period (date'2010-01-01');
```

Example 6: Start_expression with CURRENT_DATE

If CURRENT_DATE or CURRENT_TIMESTAMP is specified in the *start_expression* of the first range in RANGE_N, and if this *start_expression* when resolved with a new CURRENT_DATE or CURRENT_TIMESTAMP falls on a partition boundary, then all partitions prior to the partition matched are dropped. Otherwise, the entire table is re-partitioned with the new partitioning expression.

Consider the following CREATE TABLE statement submitted on April 1, 2006:

```
CREATE TABLE ppi (i INT, j DATE)
PRIMARY INDEX (i)
PARTITION BY
   RANGE_N (j BETWEEN CURRENT_DATE AND
            CURRENT_DATE + INTERVAL '1' YEAR - INTERVAL '1' DAY
            EACH INTERVAL '1' MONTH);
```

The last resolved date is April 1, 2006. If you submit an ALTER TABLE TO CURRENT statement on June 1, 2006, the *start_expression*, newly resolved to CURRENT_DATE ('2006-06-01'), falls on a partition boundary of the third partition. Therefore, partitions 1 and 2 are dropped, and the last reconciled date is set to the newly resolved CURRENT_DATE.

However, if you submitted the ALTER TABLE TO CURRENT statement on June 10, 2006 instead of June 1, 2006, the *start_expression*, newly resolved to CURRENT_DATE ('2006-06-10'), does not fall on a partition boundary. Therefore, all rows are scanned and the rows are repartitioned based on the new partitioning expression. The partition boundary after this statement aligns with the 10th day of the month instead of the earlier 1st day of the month.

Example 7: Table Partitioning to Record History

The following table definition is created in the year 2007 (the current year at the time). The table is partitioned to record 5 years of order history plus orders for the current year and one future year.

```
CREATE TABLE Orders
  (o_orderkey INTEGER NOT NULL,
   o_custkey INTEGER,
   o_orderstatus CHAR(1) CASESPECIFIC,
   o_totalprice DECIMAL(13,2) NOT NULL,
   o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
   o_orderpriority CHAR(21),
   o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(
   o_orderdate BETWEEN DATE '2002-01-01' AND DATE '2008-12-31'
   EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);
```

If, in 2008, you want to alter the table such that it continues to maintain 5 years of history plus the current year and one future year, you can submit the following statement in 2008:

```
   ALTER TABLE Orders MODIFY PRIMARY INDEX (o_orderkey)
      DROP RANGE WHERE PARTITION BETWEEN 1 AND 12
      ADD RANGE BETWEEN DATE '2009-01-01' AND DATE '2009-12-31'
         EACH INTERVAL '1' MONTH
   WITH DELETE;
```

In this case, you must compute the new dates and specify them explicitly in the ADD RANGE clause. This requires manual intervention every year the statement is submitted.

Alternatively, you can define the table using CURRENT_DATE as follows. This makes it easier to alter the partitioning.

```
 CREATE TABLE Orders
     (o_orderkey INTEGER NOT NULL,
     o_custkey INTEGER,
     o_orderstatus CHAR(1) CASESPECIFIC,
     o_totalprice DECIMAL(13,2) NOT NULL,
     o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
     o_orderpriority CHAR(21),
     o_comment VARCHAR(79))
   PRIMARY INDEX (o_orderkey)
   PARTITION BY RANGE_N(o_orderdate BETWEEN
      CAST(((EXTRACT(YEAR FROM CURRENT_DATE)-5-1900)*10000+0101) AS DATE)
      AND
      CAST(((EXTRACT(YEAR FROM CURRENT_DATE)+1-1900)*10000+1231) AS DATE)
      EACH INTERVAL '1' MONTH)
   UNIQUE INDEX (o_orderkey);
```

You can schedule the following ALTER TABLE statement to occur yearly. This statement rolls the partition window forward by efficiently dropping and adding partitions.

```
ALTER TABLE Orders TO CURRENT WITH DELETE;
```

With the use of CURRENT_DATE, you do not need to modify the ALTER TABLE statement each time you want to repartition the data based on the new dates.

In both cases, the partitioning starts on a year boundary. In the first example, the ALTER TABLE statement does not change this, so partitioning continues to start on a year boundary. However, you can specify an ALTER TABLE statement that changes the partitioning to start on a different boundary. For example, you can roll forward to start on a particular month in a year by specifying the desired dates in the ALTER TABLE statement.

In the second example, which uses CURRENT_DATE, you can only roll forward to start on a year boundary. However, you can modify the example as follows so that partitioning can be used to roll forward to start at the beginning of a month. This case assumes that, as of the CREATE TABLE date, the Orders table will contain the last 71 months of history plus the current month and 12 months in the future (a total of 84 months).

```
CREATE TABLE Orders
  (o_orderkey INTEGER NOT NULL,
   o_custkey INTEGER,
   o_orderstatus CHAR(1) CASESPECIFIC,
   o_totalprice DECIMAL(13,2) NOT NULL,
   o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
   o_orderpriority CHAR(21),
   o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN
   CAST(((EXTRACT(YEAR FROM CURRENT_DATE)-1900)*10000 +
          EXTRACT(MONTH FROM CURRENT_DATE)*100 + 01) AS DATE) -
       INTERVAL '71' MONTH
   AND
   CAST(((EXTRACT(YEAR FROM CURRENT_DATE)+1-1900)*10000 +
          EXTRACT(MONTH FROM CURRENT_DATE)*100 + 01) AS DATE)+
       INTERVAL '13' MONTH - INTERVAL '1' DAY
   EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);
```

You can schedule the following ALTER TABLE statement to occur monthly or less frequently (but before running out of future months). This statement rolls the partition window forward by dropping and adding partitions so that the Orders table continues to contain the last 71 months of history plus the current month and 12 months in the future.

```
ALTER TABLE Orders TO CURRENT WITH DELETE;
```

You can define the following simpler partitioning, but it might not be optimized, and the entire table might be scanned to reconcile rows when you submit an ALTER TABLE TO CURRENT statement. This case assumes that, as of the CREATE TABLE date, the Orders table will contain about 2,191 days of history plus the current day and about 365 days in the future (a total of about 7 years).

```
CREATE TABLE Orders
  (o_orderkey INTEGER NOT NULL,
   o_custkey INTEGER,
   o_orderstatus CHAR(1) CASESPECIFIC,
   o_totalprice DECIMAL(13,2) NOT NULL,
   o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
   o_orderpriority CHAR(21),
   o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN
   CURRENT_DATE - INTERVAL '6' YEAR
      AND
   CURRENT_DATE + INTERVAL '1' YEAR
   EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);
```

You can schedule the following ALTER TABLE statement to occur daily or less frequently (but before running out of future days). This statement rolls the partition window forward by dropping and adding partitions only if the CURRENT_DATE is the same day of the month as the day when the last CREATE TABLE or ALTER TABLE TO CURRENT statement was submitted. Otherwise, the entire table is scanned to reconcile the rows.

```
   ALTER TABLE Orders TO CURRENT WITH DELETE;
```

This can be very inefficient if the ALTER TABLE statement is not submitted on the same day of the month as the day when the last CREATE TABLE or ALTER TABLE TO CURRENT statement was submitted. Performance degrades as the number of days between the last resolved date and the new resolved date increases due to the increasing number of rows that must be moved.

For example, if the last resolved date was January 1, 2008, and the next ALTER TABLE TO CURRENT statement is submitted on February 2, 2008, all the rows of the table will be moved to new partitions.

Example 8: Defining Ranges

The following example defines 5 ranges. The session collation is ASCII.

```
   RANGE_N(animal BETWEEN *, 'ape', 'bird', 'bull' AND 'cow',
           'dog' AND *, NO RANGE, UNKNOWN)
```

where:

Range	Includes...
1	all values less than 'ape'.
2	strings greater than or equal to 'ape' and less than 'bird'.
3	strings greater than or equal to 'bird' and less than 'bull'.

Range	Includes...
4	strings between 'bull' and 'cow'.
5	strings greater than or equal to 'dog'.

If the value of *animal* matches one of the defined ranges, RANGE_N returns the number of the matched range.

If the value of *animal* is greater than 'cow' but less than 'dog', it does not match any of the ranges, so RANGE_N returns 6 because NO RANGE is specified.

If the value of *animal* is NULL, RANGE_N returns 7 because UNKNOWN is specified.

Example 9: Defining Five Ranges

The following example defines 5 ranges. The session collation is ASCII.

```
RANGE_N(animal BETWEEN *, 'ape', 'bird', 'bull' AND 'cow',
        'dog' AND *, UNKNOWN)
```

where:

Range	Includes...
1	all values less than 'ape'.
2	strings greater than or equal to 'ape' and less than 'bird'.
3	strings greater than or equal to 'bird' and less than 'bull'.
4	strings between 'bull' and 'cow'.
5	strings greater than or equal to 'dog'.

If the value of *animal* matches one of the defined ranges, RANGE_N returns the number of the matched range.

If the value of *animal* is greater than 'cow' but less than 'dog', it does not match any of the ranges, so RANGE_N returns NULL because NO RANGE is not specified.

If the value of *animal* is NULL, RANGE_N returns 6 because UNKNOWN is specified.

Example 10: Defining Two Ranges

In this example, the session collation is ASCII when submitting the CREATE TABLE statement, and the pad character is <space>. The example defines two ranges (numbered 1 and 2):

- Any values greater than or equal to 'a   ' (a followed by 9 spaces) or less than 'b   ' are mapped to partition 1.
- Any values greater than or equal to 'b   ' or less than 'c   ' are mapped to partition 2.

```
    CREATE SET TABLE t2
       (a VARCHAR(10) CHARACTER SET UNICODE NOT CASESPECIFIC,
        b INTEGER)
    PRIMARY INDEX (a)
    PARTITION BY RANGE_N(a BETWEEN 'a','b' AND 'c');
```

The following INSERT statement inserts a character string consisting of a single <tab> character between the 'b' and '1'.

```
    INSERT t2 ('b        1', 1);
```

The following INSERT statement inserts a character string consisting of a single <space> character between the 'b' and '1'.

```
    INSERT t2 ('b 1', 2);
```

The following SELECT statement shows the result of the INSERT statements. Since the <tab> character has a lower code point than the <space> character, the first string inserted maps to partition 1.

```
    SELECT PARTITION, a, b FROM t2 ORDER BY 1;
    *** Query completed. 2 rows found. 3 columns returned.
    *** Total elapsed time was 1 second.
    PARTITION  a          b
  -----------  ------ -----
            1  b 1        1   (string contains single <tab> character)
            2  b 1        2   (string contains single <space> character)
```

Related Information

- For more information about PPI properties and performance considerations or PPI considerations and capacity planning, see *Teradata Vantage™ - Database Design*, B035-1094.
- For more information about specifying a PPI for a table, see CREATE TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information about specifying a PPI for a join index, see CREATE JOIN INDEX in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information about modifying the partitioning of the primary index for a table, see ALTER TABLE in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information about the reconciliation of the partitioning based on newly resolved CURRENT_DATE and CURRENT_TIMESTAMP values, see ALTER TABLE TO CURRENT in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information on ADD_MONTHS, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ROUND

Returns *numeric_value* rounded *places_value* places to the right or left of the decimal point.

ROUND is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

ROUND Function Syntax

```
[TD_SYSFNLIB.] ROUND (numeric_value [, places_value])
```

Syntax Elements

TD_SYSFNLIB.
Name of the database where the function is located.

numeric_value
A numeric argument.

places_value
The number of places to round. If not specified, *numeric_value* is rounded to 0 places by default.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *places_value* = NUMBER

For the *places_value* argument, you can also pass values with data types that can be converted to INTEGER using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *numeric_value* argument.

Note:
The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is the same type as that of the *numeric_value* argument.

If the data type of *numeric_value* is DECIMAL/NUMERIC with a precision less than 38, the return DECIMAL/NUMERIC value will have its precision increased by 1. For example, a DECIMAL(6,4) argument is returned as a DECIMAL(7,4). If the precision is 38, the scale will be reduced by 1 unless the scale is 0. For example, a DECIMAL(38,38) argument is returned as DECIMAL(38,37).

For information on the default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Usage Notes

ROUND functions as follows:

- It rounds *places_value* places to the right of the decimal point if *places_value* is positive.
- It rounds *places_value* places to the left of the decimal point if *places_value* is negative.
- It rounds to 0 places if *places_value* is zero or is omitted.
- If *numeric_value* or *places_value* is NULL, the function returns NULL.
- ROUND rounds the value away from zero and it only rounds when the next digit is a value of 5 or greater.

Note:

The rounding behavior always follows the above formula regardless of the setting of the DBS Control Record RoundHalfWayMagUp field. That is, the rounding behavior always functions as if the RoundHalfWayMagUp field is set to TRUE.

Examples

Example 1: Query Returns with Result 32.4000

The following query returns the result 32.4000.

```
SELECT ROUND(32.4467, 1);
```

Example 2: Query Returns with Result 32.4600

The following query returns the result 32.4600.

```
SELECT ROUND(32.4567, 2);
```

Example 3: Query Returns with Result 100.0000

The following query returns the result 100.0000.

```
SELECT ROUND(99.9999, 3);
```

Example 4: Query Returns with Result 30.0000

The following query returns the result 30.0000.

```
SELECT ROUND(32.4567, -1);
```

Example 5: Query Returns with Result 100.0000

The following query returns the result 100.0000.

```
SELECT ROUND(55.4567, -2);
```

Example 6: Query Returns with Result 0.0000

The following query returns the result 0.0000.

```
SELECT ROUND(55.4567, -3);
```

Example 7: Query Returns with Result -5.00

The following query returns the result -5.00.

```
SELECT ROUND(-5.35, 0);
```

Example 8: Query Returns with Result -6.00

The following query returns the result -6.00.

```
SELECT ROUND(-5.55, 0);
```

Related Information

- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- Embedded Services System Functions
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

SIGN

Returns the sign of *numeric_value*.

SIGN is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

SIGN Function Syntax

```
[TD_SYSFNLIB.] SIGN (numeric_value)
```

Syntax Elements

TD_SYSFNLIB.
Name of the database where the function is located.

numeric_value
A numeric argument.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is NUMBER.

For information on the default data type format for NUMBER, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Usage Notes

For all numeric types except FLOAT/REAL/DOUBLE PRECISION, SIGN will return the following:

- If *numeric_value* is < 0, -1 is returned.
- If *numeric_value* is = 0, 0 is returned.
- If *numeric_value* is > 0, 1 is returned.

For FLOAT/REAL/DOUBLE PRECISION, SIGN will return the following:

- If *numeric_value* is < 0, -1 is returned.
- If *numeric_value* is >= 0, 1 is returned.

If the input argument is NULL, the function returns NULL.

Examples

Example 1: Query SELECT SIGN(-2);

The following query returns the result -1.

```
SELECT SIGN(-2);
```

Example 2: Query Results When the Value is an Integer and Equal to 0

The following query returns the result 0 since the value is an integer and equal to 0.

```
SELECT SIGN(CAST(0 AS INTEGER));
```

Example 3: Query Results When the data type is FLOAT and the value is >= 0

The following query returns the result 1 since the data type is FLOAT and the value is >= 0.

```
SELECT SIGN(CAST(0 as FLOAT));
```

Example 4: Query SELECT SIGN(3.74);

The following query returns the result 1.

```
SELECT SIGN(3.74);
```

Related Information

- Embedded Services System Functions
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

SQRT

Computes the square root of an argument.

SQRT Function Syntax

```
SQRT (arg)
```

Syntax Elements

arg

A positive, numeric argument.

Argument Types and Rules

If *arg* is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
    - Numeric
    - Character
    - DATE

      To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Implicit type conversion of UDTs for system operators and functions, including SQRT, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. See *Teradata Vantage™ - Database Utilities*, B035-1102.

SQRT cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for SQRT(*arg*) are as follows.

Data Type	Format	Title
FLOAT	Default format for FLOAT	SQRT(*arg*)

Examples: Representative SQRT Arithmetic Function Expressions

Representative SQRT arithmetic function expressions and the results are as follows.

Expression	Result
SQRT(2)	1.41421356237309E+000
SQRT(-2)	Error

Related Information

- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

TRIGONOMETRIC

Performs the trigonometric or inverse trigonometric function of an argument.

TRIGONOMETRIC Function Syntax

```
{ { COS | SIN | TAN | ACOS | ASIN | ATAN } (arg) |
  ATAN2 (x, y)
}
```

Syntax Elements

COS

Cosine. The cosine of an angle is the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.

COS (*arg*) is a value in radians in the range -1 to 1, inclusive.

SIN

Sine. The sine of an angle is the ratio of two sides of a right triangle. The ratio is the length of the side opposite to the angle divided by the length of the hypotenuse.

SIN (*arg*) is a value in radians in the range -1 to 1, inclusive.

TAN

Tangent. The tangent of an angle is the ratio of two sides of a right triangle. The ratio is the length of the side opposite to the angle divided by the length of the side adjacent to the angle.

TAN (*arg*) is a value in radians.

ACOS

Arccosine. The arccosine is the angle whose cosine is the argument.

ACOS (*arg*) is an angle in the range 0 to π radians, inclusive.

ASIN

Arcsine. The arcsine is the angle whose sine is the argument.

ASIN (*arg*) is an angle in the range -π /2 to π /2 radians, inclusive.

ATAN

Arctangent. The arctangent is the angle whose tangent is the argument.

ATAN (*arg*) is an angle in the range -π /2 to π /2 radians, inclusive.

ATAN2

ATAN2 (*arg*) is an angle between -π and π radians, excluding -π.

A positive result represents a counterclockwise angle from the x-axis. A negative result represents a clockwise angle.

ATAN2(*x*,*y*) equals ATAN(*y*/*x*), except that *x* can be 0 in ATAN2(*x*,*y*) and *x* cannot be 0 in ATAN(*y*/*x*) since this results in a divide by zero error.

If both *x* and y are 0, an error is returned.

arg

Any valid numeric expression that expresses an angle in radians.

x

The x-coordinate of a point to use in the arctangent2 calculation.

y

The y-coordinate of a point to use in the arctangent2 calculation.

Argument Types and Rules

Arguments that are not FLOAT are converted to FLOAT based on implicit type conversion rules. See *Teradata Vantage™ - Data Types and Literals*, B035-1143. If an argument cannot be converted, an error is reported.

If an argument is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
    - Numeric
    - Character
    - DATE

      To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- Implicit type conversion of UDTs for system operators and functions, including trigonometric and inverse trigonometric functions, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Trigonometric and inverse trigonometric functions cannot take the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

Here are the default data type, format, and title for the result of the trigonometric and inverse trigonometric functions.

Data Type	Format	Title

FLOAT	Default format for FLOAT	Cos(arg)
		Sin(arg)
		Tan(arg)
		ArcCos(arg)
		ArcSin(arg)
		ArcTan(arg)
		Atan2(x,y)

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Examples: Representative Function Expressions

The following are representative function expressions and results.

Expression	Result
COS(5-4)	5.40302305868140E -001
SIN(LOG(0.5))	-2.96504042171437E -001
SIN(RADIANS(180.0))	1.22464679914735E-016
TAN(ABS(-3))	-1.42546543074278E -001
ACOS(-0.5)	2.09439510239320E 000
ASIN(1)	1.57079632679490E 000
ATAN(1+2)	1.24904577239825E 000
ATAN2(1,1)	7.85398163397448E -001

Related Information

- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

TRUNC

Returns *numeric_value* truncated *places_value* places to the right or left of the decimal point.

TRUNC is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

TRUNC Function Syntax

```
[TD_SYSFNLIB.] TRUNC (numeric_value [, places_value ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

numeric_value

A numeric argument.

places_value

The number of places to truncate. If not specified, numeric_value is truncated to 0 places by default.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *places_value* = NUMBER

For the *places_value* argument, you can also pass values with data types that can be converted to INTEGER using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *numeric_value* argument.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is the same type as that of the *numeric_value* argument. For example, if the data type of *numeric_value* is DECIMAL/NUMERIC, the return type is DECIMAL/NUMERIC with the same precision and scale as the *numeric_value* argument.

For information on the default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Usage Notes

TRUNC functions as follows:

- It truncates *places_value* places to the right of the decimal point if *places_value* is positive.
- It truncates (makes 0) *places_value* places to the left of the decimal point if *places_value* is negative.
- It truncates to 0 places if *places_value* is zero or is omitted.
- If *numeric_value* or *places_value* is NULL, the function returns NULL.

Examples: Query Returns for SELECT TRUNC

The following query returns the result 32.4500.

```
    SELECT TRUNC(32.4567, 2);
```

The following query returns the result 30.0000.

```
    SELECT TRUNC(32.4567, -1);
```

Related Information

- *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- Embedded Services System Functions
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

WIDTH BUCKET

Returns the number of the partition to which *value_expression* is assigned.

WIDTH BUCKET Function Syntax

```
WIDTH BUCKET (value_expression, lower_bound, upper_bound, partition_count)
```

Syntax Elements

value_expression

The value for which a partition number is to be returned.

lower_bound

The lower boundary for the range of values to be partitioned equally.

upper_bound

The upper boundary for the range of values to be partitioned equally.

partition_count

Number of partitions to be created.

This value also specifies the width of the partitions by default.

The number of partitions created is *partition_count* + 2. Partition 0 and partition *partition_count* + 1 account for values of *value_expression* that are outside the lower and upper boundaries.

Argument Types and Rules

Refer to the following table for rules regarding WIDTH_BUCKET arguments.

If an argument cannot be implicitly converted to an acceptable type, an error is reported. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Data Type	Rules
Numeric	WIDTH_BUCKET accepts all numeric data types as arguments. The arguments *value_expression*, *lower_bound*, and *upper_bound* are converted to REAL before processing. The *partition_count* argument is converted to INTEGER before processing.
Character	WIDTH_BUCKET accepts character strings that represent numeric values, and converts the character strings to the appropriate numeric type.
• TIME, TIMESTAMP, or Period • INTERVAL • BYTE or VARBYTE • BLOB or CLOB • CHARACTER or VARCHAR if the server character set is GRAPHIC	WIDTH_BUCKET does not accept these types of arguments.
UDT	• The UDT must have an implicit cast to any of the following predefined types: Numeric Character DATE To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143. • Implicit type conversion of UDTs for system operators and functions, including WIDTH_BUCKET, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for WIDTH_BUCKET(*x*, *l*, *u*, *y*) are as follows.

Data Type	Format	Title
INTEGER	the default format for INTEGER	Width_bucket(x, l, u, y)

WIDTH BUCKET Usage Notes

Rules

The following rules apply to WIDTH_BUCKET:

- If any argument is null, then the result is also null.
- If *partition_count* <=0 or if *partition_count* > 2147483646, an error is returned to the requestor.
- If *lower_bound* = *upper_bound*, an error is returned to the requestor.
- If *lower_bound* < *upper_bound*, then the rules in the following table apply.

IF …	THEN the result is …
value_expression < *lower_bound*	0.
value_expression >= *upper_bound*	*partition_count* +1. If the result cannot be represented by the data type specified for the result, then an error is returned.
anything else	the greatest exact numeric value with scale 0 that is less than or equal to the following expression. $\left(\frac{(\text{partition\_count})(\text{value\_expression} - \text{lower\_bound})}{(\text{upper\_bound} - \text{lower\_bound})}\right) + 1$

- If *lower_bound* > *upper_bound*, then the rules in the following table apply.

IF …	THEN the result is …
value_expression > *lower_bound*	0.
value_expression <= *upper_bound*	*partition_count* +1.

IF …	THEN the result is …
	If the result cannot be represented by the data type specified for the result, then an error is returned.
anything else	the least exact numeric value with scale 0 that is less than or equal to the following expression. $\left(\frac{(\text{partition\_count})(\text{lower\_bound} - \text{value\_expression})}{(\text{lower\_bound} - \text{upper\_bound})}\right) + 1$

Example: Using WIDTH BUCKET to Create a Histogram for Employee Salaries within a Range

You want to create a histogram for the salaries of all employees whose salary amount ranges between $70000 and $200000. The width of each partition, or bucket, within the specified range is to be $32500.

The employee salary table contains eight employees:

```
salary   first_name   last_name
-------- ------------ -----------
50000    William      Crawford
150000   Todd         Crawford
220000   Bob          Stone
199999   Donald       Stone
70000    Betty        Crawford
70000    James        Crawford
70000    Mary         Lee
120000   Mary         Stone
```

You perform the following SELECT statement.

```
   SELECT salary, WIDTH_BUCKET(salary,70000,200000,4),COUNT(salary)
   FROM emp_salary
   GROUP BY 1Teradata Vantage
   ORDER BY 1;
```

The report produced by this statement looks like this.

```
salary   Width_bucket(salary,70000,200000,4)  Count(salary)
-------- ------------------------------------ ----------------
50000    0                                     1
70000    1                                     3
120000   2                                     1
150000   3                                     1
```

```
199999   4                                    1
220000   5                                    1
```

Related Information

- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

ZEROIFNULL

Converts data from null to 0 to avoid cases where a null result creates an error.

ZEROIFNULL Function Syntax

```
ZEROIFNULL (arg)
```

Syntax Elements

arg

A numeric argument.

Argument Types and Rules

Value of arg	ZEROIFNULL Value Returned
Not null	Value of the numeric argument.
Null or zero **Note:** A structured UDT column value is null only when you explicitly place a NULL in the column, not when a structured UDT instance has an attribute that is set to NULL.	Zero.

If the argument is not numeric, the value is converted to a numeric value according to implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

If *arg* is a character string, the string is converted to a numeric value of FLOAT data type.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
    - Numeric

  - Character
  - DATE
  - Interval

    To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- Implicit type conversion of UDTs for system operators and functions, including ZEROIFNULL, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

ZEROIFNULL cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

Here are the default attributes for the result of ZEROIFNULL.

- If the operand is numeric, the format is the same format as *arg*.
- If the operand is character, the format is the default format for FLOAT.
- If the operand is a UDT, the format is the format of the predefined type to which the UDT is implicitly cast.

Note:

The NULL keyword has a data type of INTEGER.

Example: Testing the Salary Column for Null

In this example, you can test the Salary column for null.

```
SELECT empno, ZEROIFNULL(salary)
FROM employee ;
```

A nonzero value is returned for each employee number, indicating that no nulls exist in the Salary column.

Related Information

- COALESCE Expression
- NULLIF Expression
- NULLIFZERO
- *Teradata Vantage™ - Data Types and Literals*, B035-1143
- *Teradata Vantage™ - Database Utilities*, B035-1102
- *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144

Attribute Functions

The following sections describe SQL attribute functions.

Attribute functions return descriptive information about their operand. Except for the DEFAULT function, the operand need not be a column reference; it can be a general expression that is not evaluated mathematically.

When an attribute function is used in a request, the response returns one row for every data row that meets the search condition.

Some of these functions are extensions to ANSI SQL.

For a list of data type attributes, see "Data Type Phrases" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Each attribute function is described individually in the following topics.

ANSI Equivalence of Teradata Attribute Functions

Several of the Teradata attribute functions are extensions to the ANSI SQL:2011 standard.

To maintain ANSI compatibility, use the ANSI equivalent functions instead of Teradata attribute functions, when available. As listed in the table below, change the Teradata function to the ANSI function in new applications.

Teradata Function	ANSI Function
CHARACTERS CHARS CHAR	CHARACTER_LENGTH
MCHARACTERS†	
† This function is no longer documented because its use is deprecated and it will no longer be supported after support for KANJI1 is dropped.	

The following Teradata functions have no ANSI equivalents:

- BYTES
- FORMAT
- TYPE

BIT_LENGTH

Returns the length of the string expression in bits.

BIT_LENGTH Function Syntax

```
BIT_LENGTH ( string_expression [, character_set_name ] )
```

Syntax Elements

string_expression

The character string for which the number of bits is required.

The data type of *string_expression* must be one of the following:

- CHARACTER or VARCHAR
- UDT that has an implicit cast to a predefined character type

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

  Implicit type conversion of UDTs for system operators and functions, including OCTET_LENGTH, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

If the *string_expression* argument is *null*, the result is *null*.

character_set_name

The character set in which the result is returned. If character_set_name is not provided, the session character set is assumed.

If the string _expression argument is NULL, the result is NULL.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Example

An example of BIT_LENGTH usage:

```
SELECT BIT_LENGTH('Hello');
```

Related Information

OCTET_LENGTH.

BYTE/BYTES

Returns the number of bytes contained in the specified byte string.

BYTE/BYTES Function Syntax

```
{ BYTE | BYTES } ( byte_expression )
```

Syntax Elements

byte_expression

An expression of one of the following types:

- BYTE, VARBYTE and BLOB
- UDT that has an implicit cast to a predefined byte type

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

  Implicit type conversion of UDTs for system operators and functions, including BYTES, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

BYTE/BYTES Usage Notes

Length Includes Trailing Zeros

Because trailing double zero bytes are considered bytes, the length of the value in a fixed length column is always equal to the length defined for the column.

The length of the value in a variable length column is always equal to the number of bytes, including any trailing double zero bytes, contained in that value.

If you do not want trailing blanks included in the byte count for a data value, use the TRIM function on the argument to BYTES. For example:

```
   SELECT BYTES( TRIM( TRAILING FROM byte_col ) ) FROM table1;
```

Example: Using BYTE to Obtain the Number of Bytes in a Badge Picture

The following statement applies the BYTES function to the BadgePic column, which is type VARBYTE(32000), to obtain the number of bytes in each badge picture.

```
SELECT BadgePic, BYTES(BadgePic)
FROM Employee;
```

The result is as follows:

```
BadgePic        Bytes(BadgePic)
--------------  ---------------
20003BA0                      4
9A3243F805                    5
EEFF08C3441900                7
```

Related Information

- *Teradata Vantage™ - Data Types and Literals*, B035-1143
- *Teradata Vantage™ - Database Utilities*, B035-1102
- TRIM

CHARACTER_LENGTH

Returns the length of a string either in logical characters or in bytes.

CHARACTER_LENGTH Function Syntax

```
{ CHARACTER_LENGTH | CHAR_LENGTH } ( string_expression )
```

Syntax Elements

string_expression

The string expression for which the length is to be returned.

The type of *string_expression* must be CHARACTER, VARCHAR, or CLOB. For non-character data types, the function returns an error.

By default, Vantage performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and a predefined character type.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

Implicit type conversion of UDTs for system operators and functions, including CHARACTER_LENGTH, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type

For all server character sets except KANJI1, CHARACTER_LENGTH returns the length of *string_expression* in characters.

If the *string_expression* argument is *null*, the result is *null*.

For KANJI1, the following results are obtained.

FOR this client character set …	CHARACTER_LENGTH returns …
KanjiEBCDIC	the length of *string_expression* as the number of bytes. A mix of single and multibyte characters is expected. If any Shift-Out/Shift-In characters are present, they are included in the result count.
KanjiEUC KanjiShift-JIS	the length of *string_expression* as the number of logical characters, based on the client session character set. A mix of single and multibyte characters is expected.
ASCII EBCDIC	the length of *string_expression* as the number of bytes.

Note:

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see *KANJI1 Character Set* in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Because trailing pad characters are considered characters, the length of the value in a CHARACTER column is always equal to the length defined for the column.

The length of the value in a VARCHAR or CLOB column is always equal to the number of characters, including any trailing pad characters, contained in that value.

Usage Notes

CHARACTER_LENGTH is the ANSI form of the Teradata CHARACTERS function. Use CHARACTER_LENGTH instead of CHARACTERS for ANSI SQL:2011 conformance.

Use CHARACTER_LENGTH in place of MCHARACTERS. (MCHARACTERS no longer appears in this book because its use is deprecated and it will not be supported after support for KANJI1 is dropped.)

Suppressing Trailing Pad Characters

To suppress trailing pad characters from the character count for a data value, use the TRIM function on the argument to CHARACTER_LENGTH. For example:

```
SELECT CHARACTER_LENGTH( TRIM( TRAILING FROM Name ) )
FROM Employee;
```

Examples

The following statement applies the CHARACTER_LENGTH function to the Name column, which is type VARCHAR(30) CHARACTER SET LATIN, to obtain the number of characters in each employee name:

```
SELECT Name, CHARACTER_LENGTH(Name)
FROM Employee;
```

The result is as follows (note that separator blanks are considered characters):

```
Name      Character_Length(Name)
--------  ----------------------
Smith T            7
Newman P           8
Omura H            7
      .            .
```

Example Set 1: KanjiEBCDIC

FOR this server character set …	AND example …	CHARACTER_LENGTH returns …
GRAPHIC	ABC	3
KANJI1	De<MNP >	10
	<><>	4

Set 2: KanjiShift-JIS

FOR this server character set ...	AND example ...	CHARACTER_LENGTH returns ...
KANJI1	<><>	10
	D eF	3
UNICODE	ABC	3
GRAPHIC	ABC	3

Set 3: KanjiEUC

FOR this server character set ...	AND example ...	CHARACTER_LENGTH returns ...
KANJI1	ss3 C ss3 D	2
GRAPHIC		2
UNICODE	<><>	0
	dA ss2 B ss3 E	4
LATIN	ABC	3

Related Information

- For information on data type conversions, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- *Teradata Vantage™ - Database Utilities*, B035-1102.
- “KANJI1 Character Set" in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

DEFAULT

Returns the current default value for the specified or derived column.

The DEFAULT function cannot be used as a partitioning expression for defining PPIs.

DEFAULT Function Syntax

```
DEFAULT [ ( column_name ) ]
```

Syntax Elements

column_name

The name of a column in a base table, view, queue table, or derived table.

The column name can be qualified or unqualified.

The DEFAULT function returns the default value of the specified column or derived column (if *column_name* is omitted).

If the specified or derived column is a view column or derived table column, the DEFAULT function returns the default value of the underlying table column.

If the default value of a column evaluates to a system variable, for example when the default value is CURRENT_TIME or USER, the DEFAULT function returns the value of the system variable at the time the statement is executed.

DEFAULT returns null when any of the following conditions are true:

- The specified or derived column was defined with a DEFAULT NULL phrase
- The specified or derived column has no explicit default value
- The data type of the specified or derived column is UDT
- The specified or derived column is the name of a view column that is derived from a single underlying table column that has no explicit default value

  For an example, see “Example: Specifying a View Column Name”.
- The specified or derived column is the name of a view column that is not derived from a single underlying table column, for example, the view column is derived from a literal expression

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Result Type and Attributes

The result type, format, and title for DEFAULT(*x*) appear in the following table.

Data Type	Format	Title
Data type of the specified column	Format of the specified column	Default(x)

For information on data type default formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

DEFAULT Usage Notes

Omitting the Column Name

You can use the form of DEFAULT that omits the column name under certain conditions in an INSERT, UPDATE, or MERGE statement or in a predicate clause that involves a comparison operation. The form of DEFAULT that omits the column name cannot be part of an expression.

When the DEFAULT function does not specify a column name, Vantage derives the column based on context. For example, consider the following table definition:

```
CREATE TABLE Manager
   (Emp_ID      INTEGER
   ,Dept_No     INTEGER DEFAULT 99
);
```

The following INSERT statement uses DEFAULT without a column name to insert the default value into the Dept_No column:

```
INSERT INTO Manager VALUES (103499, DEFAULT);
```

Using the DEFAULT function without specifying a column name can produce an error if Vantage cannot derive the column context.

For an example that omits the column name when using the DEFAULT function in a predicate clause that involves a comparison operation, see Example: Using DEFAULT in a Predicate.

For details on using the DEFAULT function in INSERT, UPDATE, and MERGE statements, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

Using a Qualified Column Name

If you specify a qualified column name that includes the name of the table, you can use DEFAULT in a SELECT statement that has no FROM clause. For example, you can use the following statement to get the default value of the Dept_No column in the Manager table:

```
SELECT DEFAULT(Manager.Dept_No);
```

Error Conditions

Using the DEFAULT function can result in an error when any of the following conditions are true:

- The column name is omitted and Vantage cannot derive the column context

- The DEFAULT function appears in a partitioning expression for defining PPIs
- The column name is omitted and the DEFAULT function appears in an expression that does not support the DEFAULT function without a column name
- The DEFAULT function appears in an expression for which the result type is incompatible

  For example, consider the following table definition:

  ```
  CREATE TABLE Parts_Table
     (Part_Code   INTEGER DEFAULT 9999
     ,Part_Name   CHAR(20)
  );
  ```

  The following statement results in an error because the result type of the DEFAULT function is not compatible with the column to which the result is being compared:

  ```
  SELECT * FROM Parts_Table WHERE Part_Name = DEFAULT(Part_Code);
  ```

Examples

Example: Inserting the Default Value under Certain Conditions

Consider the following Employee table definition:

```
CREATE TABLE Employee
   (Emp_ID      INTEGER
   ,Last_Name   VARCHAR(30)
   ,First_Name  VARCHAR(30)
   ,Dept_No     INTEGER DEFAULT 99
);
```

The following statement uses DEFAULT to insert the default value of the Dept_No column when the supplied value is negative.

```
USING (id INTEGER, n1 VARCHAR(30), n2 VARCHAR(30), dept INTEGER)
INSERT INTO Employee VALUES
   (:id
   ,:n1
   ,:n2
   ,CASE WHEN (:dept < 0) THEN DEFAULT(Dept_No) ELSE :dept END
);
```

Example: Using DEFAULT in a Predicate

The following statement uses DEFAULT to compare the values of the Dept_No column with the default value of the Dept_No column. Because the comparison operation involves a single column reference, Vantage can derive the column context of the DEFAULT function even though the column name is omitted.

```
SELECT * FROM Employee WHERE Dept_No < DEFAULT;
```

Note that if the DEFAULT function evaluates to null, the predicate is unknown and the WHERE condition is false.

Example: Specifying a View Column Name

Consider the DBC.HostsInfo system view, which has the following definition:

```
REPLACE VIEW DBC.HostsInfo (LogicalHostId, HostName, DefaultCharSet)
AS SELECT
    LogicalHostId
   ,HostName
   ,DefaultCharSet
FROM DBC.Hosts WITH CHECK OPTION;
```

The underlying table, DBC.Hosts, has the following definition:

```
CREATE SET TABLE DBC.Hosts, FALLBACK, NO BEFORE JOURNAL,
NO AFTER JOURNAL, CHECKSUM = DEFAULT
   (LogicalHostId SMALLINT FORMAT 'ZZZ9' NOT NULL
   ,HostName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL
   ,DefaultCharSet VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC
      NOT NULL)
UNIQUE PRIMARY INDEX (LogicalHostId)
UNIQUE INDEX (HostName);
```

The following statement uses the DEFAULT function with the DBC.HostsInfo.HostName view column name:

```
SELECT DISTINCT DEFAULT(HostName) FROM DBC.HostsInfo;
```

The result of the DEFAULT function is null because the HostName view column is derived from a table column that has no explicit default value.

Related Information

- About using predicates, see Logical Predicates.
- About comparison operations in predicates, see Comparison Operators and Functions.
- About the DEFAULT value control phrase, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- About INSERT, UPDATE, and MERGE statements, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

FORMAT

Returns the declared format for the named expression.

FORMAT Function Syntax

```
FORMAT ( expression )
```

Syntax Elements

expression

The expression for which the FORMAT is to be reported.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

FORMAT returns a CHAR(*n*) character string of up to 30 characters. The result type, character set, format, and title for FORMAT appear in the following table.

Data Type	Format	Title
CHAR(*n*) CHARACTER SET UNICODE	X(30)	Format(*named_expression*)

Example: Requesting the Format of the Salary Column

The following statement requests the format of the Salary column in the Employee table.

```
   SELECT FORMAT(Employee.Salary);
```

The result is the following.

```
Format(Salary)
------------------------------
ZZZ,ZZ9.99
```

Related Information

Teradata Vantage™ - Data Types and Literals, B035-1143.

OCTET_LENGTH

Returns the length of *string_expression* in octets when it is converted to the named character set (taking the export width value into consideration). The maximum possible value returned from the OCTET_LENGTH function is 64,000 octets.

OCTET_LENGTH Function Syntax

```
OCTET_LENGTH ( string_expression [, character_name ] )
```

Syntax Elements

string_expression

The character string for which the number of octets is required.

The data type of *string_expression* must be one of the following:

- CHARACTER or VARCHAR
- UDT that has an implicit cast to a predefined character type

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

  Implicit type conversion of UDTs for system operators and functions, including OCTET_LENGTH, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

If the *string_expression* argument is *null*, the result is *null*.

character_name

The character set in which the result is to be returned. If *character_set_name* is not provided, the session character set is assumed.

If the *string_expression* argument is *null*, the result is *null*.

See the list of Teradata-provided character sets in the table in Usage Notes.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Usage Notes

Any Shift-Out/Shift-In and trailing GRAPHIC pad characters are included in the result count.

OCTET_LENGTH operates in the same manner in both Teradata and ANSI modes.

IF *string_expression* is …	THEN …
of type KANJI1	the result is independent of *character_set_name*.
not CHARACTER data	an error is generated.

OCTET_LENGTH takes the export width value into consideration. For information about export width, see “CREATE USER” in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

The following table lists the client character sets shipped with Teradata. Although these character sets are shipped with the system, your system administrator must install them individually to become available for use.

Your site might also have site-defined character sets. Check with your system administrator for a complete list of character sets available at your site.

Character Sets found in Built-in:

- ASCII
- EBCDIC
- UTF8
- UTF16

Character Sets found in DBC.CharTranslationsV:

- EBCDIC037_0E
- EBCDIC273_0E
- EBCDIC277_0E
- HANGULEBCDIC933_1II
- HANGULKSC5601_2R4
- KANJIEBCDIC5026_0I
- KANJIEBCDIC5035_0I
- KANJIEUC_0U
- KANJISJIS_0S
- KATAKANAEBCDIC
- LATIN1252_0A

- LATIN1_0A
- LATIN9_0A
- SCHEBCDIC935_2IJ
- SCHGB2312_1T0
- TCHBIG5_1R0
- TCHEBCDIC937_3IB

Character Sets found in DBC.CharTranslationsV with Windows code page compatible session character set:

- ARABIC1256_6A0
- CYRILLIC1251_2A0
- HANGUL949_7R0
- HEBREW1255_5A0
- KANJI932_1S0
- LATIN1250_1A0
- LATIN1252_0A
- LATIN1254_7A0
- LATIN1258_8A0
- SCHINESE936_6R0
- TCHINESE950_8R0
- THAI874_4A0

Examples: Output from OCTET_LENGTH

Examples of output from OCTET_LENGTH appear in the following table.

Client Character Set	Server Character Set	string_expression	Result
EBCDIC	LATIN	abcdefgh	8
ASCII	KANJI1	abcdefgh	8
KanjiEBCDIC	KANJI1	AB<CDE >P	11
KanjiEBCDIC	GRAPHIC	MNOP	8 (record mode)
			10 (field mode)
KanjiEUC	KANJISJIS	dA ss2 B ss3 E	8
KanjiShift-JIS	KANJISJIS	D eF	5
KanjiShift-JIS	UNICODE	ABC	6

Related Information

- For information on data type conversions, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- *Teradata Vantage™ - Database Utilities*, B035-1102.
- For information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

TITLE

Returns the title of an expression as it would appear in the heading for displayed or printed results.

TITLE Function Syntax

```
TITLE ( expression )
```

Syntax Elements

expression

The expression for which the title is to be returned.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

TITLE returns a CHAR(*n*) character string of up to 60 characters. The result type, character set, format, and title for TITLE appear in the following table.

Data Type	Format	Title
CHAR(*n*) CHARACTER SET UNICODE	X(60)	Title(*named_expression*)

Usage Notes

Use the TITLE phrase to change the heading for displayed or printed results that is different from the column name, which is the default heading.

Example: Requesting the title of the Salary Column

The following statement requests the title of the Salary column in the Employee table.

```
SELECT TITLE(Employee.Salary);
```

The result is the following.

```
Title(Salary)
-------------------------------------------------------------
Salary
```

Related Information

- *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- *Teradata Vantage™ - Data Types and Literals*, B035-1143.

TYPE

Returns the data type defined for an expression.

TYPE Function Syntax

```
TYPE ( expression )
```

Syntax Elements

expression

The expression for which the data type is to be returned.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

TYPE returns a CHAR(*n*) character string that contains the name of the data type of the expression.

When the argument is a function or operation, TYPE returns a character string that contains the result type of the function or operation. For rules on the result type for an operation or function, refer to the documentation for the specific function or operation.

The result type, character set, format, and title for TYPE appear in the following table.

Data Type	Format	Title
CHAR(*n*) CHARACTER SET LATIN	X(39)	Type(*named_expression*)

For a list of the supported data types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143. For information on geospatial types, see *Teradata Vantage™ - Geospatial Data Types*, B035-1181.

TYPE Usage Notes

Character Type Arguments

If the server character set for a character type argument is different from the user default server character set, then the resulting character string also contains the CHARACTER SET phrase and the name of the server character set for the argument.

Examples

Example 1: If User Default Server Character Set is LATIN

Consider the Name column in the following table definition:

```
CREATE TABLE Employee
  (EmployeeID INTEGER
  ,Name       CHARACTER(30) CHARACTER SET LATIN
  ,Salary     DECIMAL(8,2));
```

If the user default server character set is LATIN, then the character string that TYPE returns for the Name column does not contain the CHARACTER SET phrase.

```
SELECT TYPE(Employee.Name);

Type(Name)
----------
CHAR(30)
```

Example 2: If User Default Server Character Set is LATIN but Server Character Set for Name Column is UNICODE

If the user default server character set is LATIN, but the server character set for the Name column is UNICODE, then the result string contains the CHARACTER SET phrase.

```
CREATE TABLE Employee
  (EmployeeID INTEGER
  ,Name VARCHAR(30) CHARACTER SET UNICODE
  ,Salary     DECIMAL(8,2));

SELECT TYPE(Employee.Name);
```

```
Type(Name)
---------------------------------
VARCHAR(30) CHARACTER SET UNICODE
```

Example 3: Returning the types of the Name and Salary columns

The following statement returns the types of the Name and Salary columns:

```
SELECT TYPE(Employee.Name), TYPE(Employee.Salary);
Type(Name)   Type(Salary)
-----------  ------------
VARCHAR(30)  DECIMAL(8,2)
```

Example 4: Using TYPE to Request the Data Type of Two Columns

If TYPE is used to request the data type of two columns, defined as GRAPHIC and LONG VARCHAR CHARACTER SET GRAPHIC, respectively, the result is as follows.

```
TYPE(GColName)                 TYPE(LVGColName)
-----------------------------  ------------------------------------
CHAR(4) CHARACTER SET GRAPHIC  VARCHAR(32000) CHARACTER SET GRAPHIC
```

In the case of a LONG VARCHAR CHARACTER SET GRAPHIC column, the length returned is the maximum length of 32000.

Example 5: Using the TYPE Function

Consider the following TYPE function.

```
SELECT TYPE(SUBSTR(Employee.Name,3,2));
```

The result type of SUBSTR depends on the session mode.

If the session is set to ANSI mode, the returned result is as follows:

```
Type(Substr(Name,3,2))
----------------------
VARCHAR(30)
```

If the session is set to Teradata mode, the returned result is as follows:

```
Type(Substr(Name,3,2))
----------------------
VARCHAR(2)
```

Example 6: Applying the TYPE function to the BLOB Column

Consider the following table definition:

```
   CREATE TABLE images
     (imageid INTEGER
     ,imagedesc VA
RCHAR(50)
     ,image BLOB(2K))
   UNIQUE PRIMARY INDEX (imageid);
```

The following statement applies the TYPE function to the BLOB column:

```
   SELECT TYPE(images.image) FROM images;
```

The result is:

```
   Type(image)
   -----------
   BLOB(2048)
```

Note that the result is a normal integer length, and does not use the K option that was used to define the BLOB column the CREATE TABLE statement.

Related Information

- *Teradata Vantage™ - Geospatial Data Types*, B035-1181
- *Teradata Vantage™ - Data Types and Literals*, B035-1143

6

Bit/Byte Manipulation Functions

The byte/bit manipulation functions in the following sections are embedded services system functions. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

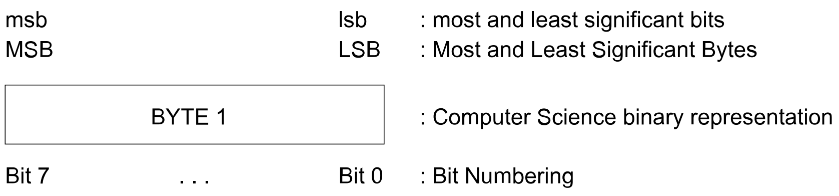
Bit and Byte Numbering Model

The following diagrams show the logical bit and byte numbering model employed by the byte/bit manipulation functions described in these sections.

The model is big endian or little endian independent. Note that the numbering system used for numeric data types is consistent with that used for byte strings. This simplifies the development of appropriate bit masks.

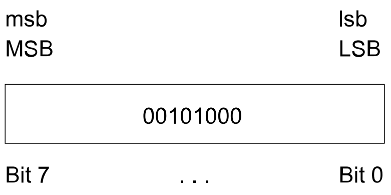
Users of the byte/bit manipulation functions should mentally visualize the numeric and byte data types as shown below when contemplating what masks (*bit_mask_arg*) need to be applied to the target data (*target_arg*).

BYTEINT


msb
lsb
: most and least significant bits
MSB
LSB
: Most and Least Significant Bytes
BYTE 1
: Computer Science binary representation
Bit 7
. . .
Bit 0
: Bit Numbering

Example: A BYTEINT Value

A BYTEINT value of 40 with a binary representation of 00101000:



SMALLINT

msb		lsb	: most and least significant bits
MSB		LSB	: Most and Least Significant Bytes
BYTE 1		BYTE 2	: Computer Science binary representation
Bit 15	. . .	Bit 0	: Bit Numbering

Example: A SMALLINT Value

A SMALLINT value of 10,280 with a binary representation of 0010100000101000:

msb		lsb
MSB		LSB
00101000		00101000
Bit 15	. . .	Bit 0

INTEGER

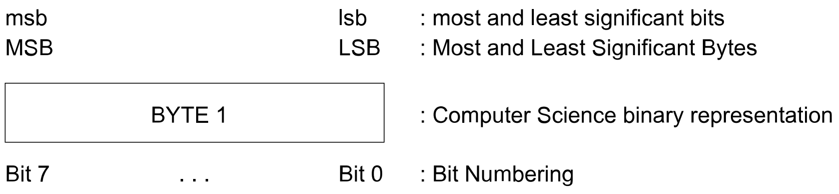
msb			lsb	
MSB			LSB	
BYTE 1	BYTE 2	BYTE 3	BYTE 4	: Computer Science binary representation
Bit 31	. . .		Bit 0	: Bit Numbering

Example: An INTEGER Value

An INTEGER value of 673,720,360 with a binary representation of 00101000 00101000 00101000 00101000:

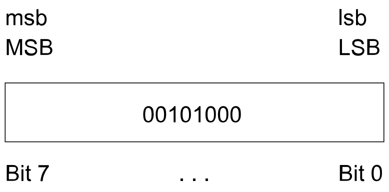
msb			lsb
MSB			LSB
00101000	00101000	00101000	00101000
Bit 32	. . .		Bit 0

BIGINT


msb
lsb
: most and least significant bits
MSB
LSB
: Most and Least Significant Bytes
BYTE 1
: Computer Science binary representation
Bit 7
. . .
Bit 0
: Bit Numbering

Example: A BIGINT Value

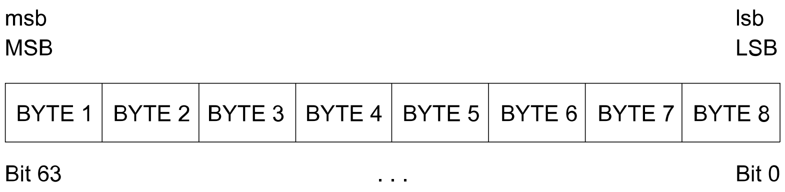
A BIGINT value of 2,893,606,913,523,066,920 with a binary representation of 00101000 00101000 00101000 00101000 00101000 00101000 00101000 00101000:



BYTE and VARBYTE

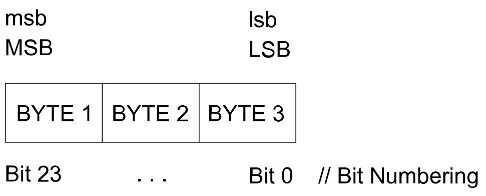
Example: A VARBYTE Value

A VARBYTE(8) with 8 bytes:



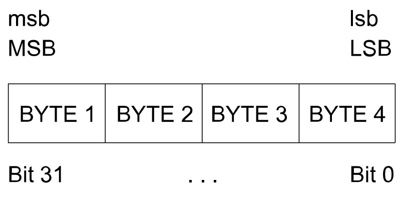
Example: A VARBYTE Value with 3 bytes

A VARBYTE(8) with 3 bytes:



Example: A VARBYTE Value

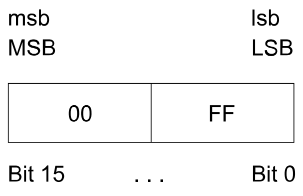
Example of BYTE(4):



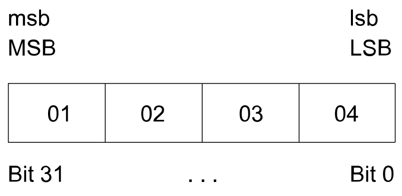
HEXADECIMAL BYTE LITERALS

With respect to byte-bit system functions, hexadecimal byte literals are interpreted as follows:

A 2-byte hexadecimal byte literal: '00FF'XB

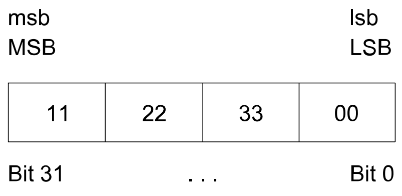


A 4-byte hexadecimal byte literal: '01020304'XB



Note that hexadecimal byte literals are represented by an even number of hexadecimal digits. Hexadecimal literals are extended on the right with zeros when required. For example:

A 3-byte hexadecimal byte literal, '112233'XB, becomes a 4-byte hexadecimal byte literal: '11223300'XB



For more information, see “Hexadecimal Byte Literals” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Performing Bit-Byte Operations against Arguments with Non-Equal Lengths

This topic applies only to the BITOR, BITXOR, and BITAND functions.

If the *target_arg* and *bit_mask_arg* arguments passed to these functions differ in length:

- The *target_arg* and *bit_mask_arg* arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

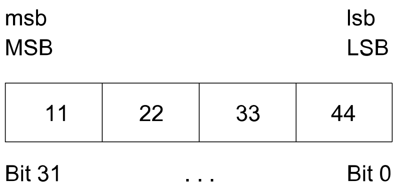
  Vantage pads to the left (instead of to the right) so that the hexadecimal byte literals, serving as bit masks, will not have to be changed every time the size of a byte string is changed.

Example: Querying the BITAND Operation On An INTEGER

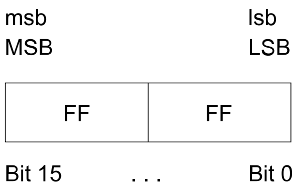
The following query performs the BITAND operation on an INTEGER and a single-byte hexadecimal byte literal.

```
SELECT BITAND(287454020, 'FFFF'XB);
```

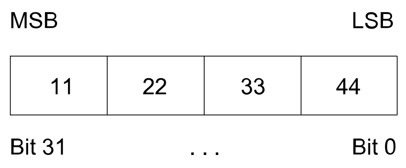
The INTEGER value 287,454,020 has a hexadecimal value of 0x11223344 and a bit numbering representation of:

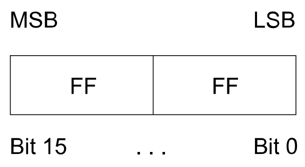


The hexadecimal byte literal 0xFFFF has a bit numbering representation of:

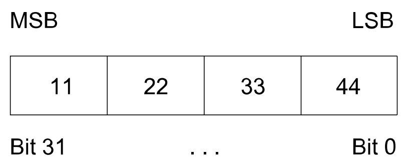


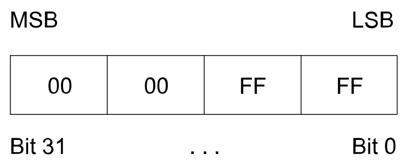
To process the BITAND operation, the two arguments are aligned on their least significant byte/bit as follows:



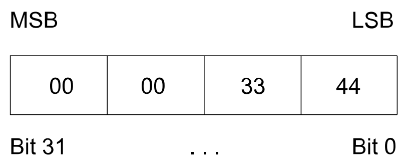


The shorter-length hexadecimal byte literal 0xFFFF is padded with zeros to the left until it is the same length as the INTEGER value 287,454,020.





When both operands are the same size, the BITAND operation is performed, producing the following result:



BITAND

Performs the logical AND operation on the corresponding bits from the two input arguments.

This function takes two bit patterns of equal length and performs the logical AND operation on each pair of corresponding bits. If the bits at the same position are both 1, then the result is 1; otherwise, the result is 0. If either input argument is NULL, the function returns NULL.

BITAND Function Syntax

```
[TD_SYSFNLIB.] BITAND ( target_arg, bit_mask_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

bit_mask_arg

A fixed byte value, a variable byte value, or a numeric expression.

If the *target_arg* and *bit_mask_arg* arguments differ in length, the arguments are processed as follows:

- The *target_arg* and *bit_mask_arg* arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

Argument Types and Rules

BITAND is an overloaded scalar function. The data type of the *target_arg* parameter can be one of the following:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- DECIMAL
- NUMBER
- VARBYTE(*n*)

Note:

DECIMAL input is implicitly converted to NUMBER(38,0). *target_arg* is not defined for DECIMAL, but it is defined for NUMBER(38,0). Due to UDF implicit type conversion rules, DECIMAL will be accepted as input.

The data type of the *bit_mask_arg* parameter varies depending upon the data type of the *target_arg* parameter. The following (*target_arg*, *bit_mask_arg*) input combinations are permitted:

***target_arg* type**	***bit_mask_arg* type**
BYTEINT	BYTE(1)

***target_arg* type**	***bit_mask_arg* type**
BYTEINT	BYTEINT
SMALLINT	BYTE(2)
SMALLINT	SMALLINT
INTEGER	BYTE(4)
INTEGER	INTEGER
BIGINT	BYTE(8)
BIGINT	BIGINT
NUMBER(38,0)	VARBYTE(16)
NUMBER(38,0)	NUMBER(38,0)
VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs. For example, BITAND(BYTEINT, INTEGER) is allowed because it can be implicitly converted to BITAND(INTEGER,INTEGER).

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For more information, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
DECIMAL	NUMBER(38,0)	NUMBER(38,0)
NUMBER	NUMBER(38,0)	NUMBER(38,0)
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

Note:

DECIMAL input is implicitly converted to NUMBER(38,0). *target_arg* is not defined for DECIMAL, but it is defined for NUMBER(38,0). Due to UDF implicit type conversion rules, DECIMAL is accepted as input.

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for BITAND is: BITAND(*target_arg, bit_mask_arg*).

Examples: Querying with the BITAND Function

Passing a BYTEINT Value to BITAND

In the following query, the input argument 23 has a data type of BYTEINT and a binary representation of 00010111. The input argument 20 has a data type of BYTEINT and a binary representation of 00010100. The bitwise AND product of the two arguments results in a BYTEINT value of 20, or binary 00010100, which is returned by the query.

```
   SELECT BITAND(23,20);
```

Passing a NUMBER Value to BITAND

```
SELECT BITAND( CAST('A593C38281B4D2E1'XI8 AS NUMBER), 'FFFFFFFFFFFFFFFF'xb);
```

Result:

```
BITAND(-6515649270585699615,'FFFFFFFFFFFFFFFF'XB)
-------------------------------------------------
                             -6515649270585699615
```

```
SELECT TO_BYTE( CAST( BITAND( CAST('5A393C28184B2D1E'XI8 AS NUMBER),
'0000FFFFFFFFFFFF'xb) AS BIGINT ));
```

Result:

```
TO_BYTE(BITAND(6501293679989959966,'0000FFFFFFFFFFFF'XB))
---------------------------------------------------------
00003C28184B2D1E
```

Related Information

- “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147
- *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about *target_arg* and *bit_mask_arg*, see Performing Bit-Byte Operations against Arguments with Non-Equal Lengths.

BITNOT

Performs a bitwise complement on the binary representation of the input argument.

The bitwise NOT, or complement, is a unary operation which performs logical negation on each bit, forming the ones' complement of the specified binary value. The digits in the argument which were 0 become 1, and vice versa.

BITNOT Function Syntax

```
[TD_SYSFNLIB.] BITNOT ( target_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

BITNOT returns NULL if *target_arg* is NULL.

Argument Types and Rules

BITNOT is an overloaded scalar function. It is defined with the following parameter data types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

If the argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for BITNOT is: BITNOT(*target_arg*).

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Example: Querying with the BITNOT Function

In the following query, the input argument 2 has a data type of BYTEINT and a binary representation of 00000010. Performing a BITNOT operation on this value results in a BYTEINT value of -3, or binary 11111101.

```
SELECT BITNOT(2);
```

BITOR

Performs the logical OR operation on the corresponding bits from the two input arguments.

This function takes two bit patterns of equal length and performs the logical OR operation on each pair of corresponding bits as follows.

IF...	THEN the result is...
either of the bits from the input arguments is 1	1
both of the bits from the input arguments are 0	0
any of the input arguments is NULL	NULL

BITOR Function Syntax

```
[TD_SYSFNLIB.] BITOR ( target_arg, bit_mask_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

bit_mask_arg

A fixed byte value, a variable byte value, or a numeric expression.

If the *target_arg* and *bit_mask_arg* arguments differ in length, the arguments are processed as follows:

- The *target_arg* and *bit_mask_arg* arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

For more information, see Performing Bit-Byte Operations against Arguments with Non-Equal Lengths.

Argument Types and Rules

BITOR is an overloaded scalar function. The data type of the *target_arg* parameter can be one of the following:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- VARBYTE(*n*)

The data type of the *bit_mask_arg* parameter varies depending upon the data type of the *target_arg* parameter. The following (*target_arg*, *bit_mask_arg*) input combinations are permitted.

***target_arg* type**	***bit_mask_arg* type**
BYTEINT	BYTE(1)
BYTEINT	BYTEINT
SMALLINT	BYTE(2)
SMALLINT	SMALLINT
INTEGER	BYTE(4)
INTEGER	INTEGER
BIGINT	BYTE(8)
BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs. For example, BITOR(BYTEINT, INTEGER) is allowed because it can be implicitly converted to BITOR(INTEGER,INTEGER).

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For more information, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table.

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for BITOR is: BITOR(*target_arg*, *bit_mask_arg*).

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Example: Querying with the BITOR Function

In the following query, the input argument 23 has a data type of BYTEINT and a binary representation of 00010111. The input argument 45 has a data type of BYTEINT and a binary representation of 00101101. The bitwise OR product of the two arguments results in a BYTEINT value of 63, or binary 00111111, which is returned by the query.

```
    SELECT BITOR(23,45);
```

BITXOR

Performs a bitwise XOR operation on the binary representation of the two input arguments.

The bitwise exclusive OR takes two bit patterns of equal length and performs the logical XOR operation on each pair of corresponding bits. The result in each position is 1 if the two bits are different, and 0 if they are the same. If either input argument is NULL, the function returns NULL.

BITXOR Function Syntax

```
[TD_SYSFNLIB.] BITXOR ( target_arg, bit_mask_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

bit_mask_arg

A fixed byte value, a variable byte value, or a numeric expression.

If the *target_arg* and *bit_mask_arg* arguments differ in length, the arguments are processed as follows:

- The *target_arg* and *bit_mask_arg* arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

Argument Types and Rules

BITXOR is an overloaded scalar function. The data type of the *target_arg* parameter can be one of the following:

- BYTEINT
- SMALLINT

- INTEGER
- BIGINT
- VARBYTE(*n*)

The data type of the *bit_mask_arg* parameter varies depending upon the data type of the *target_arg* parameter. The following (*target_arg*, *bit_mask_arg*) input combinations are permitted.

***target_arg* type**	***bit_mask_arg* type**
BYTEINT	BYTE(1)
BYTEINT	BYTEINT
SMALLINT	BYTE(2)
SMALLINT	SMALLINT
INTEGER	BYTE(4)
INTEGER	INTEGER
BIGINT	BYTE(8)
BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs. For example, BITXOR(BYTEINT, INTEGER) is allowed because it can be implicitly converted to BITXOR(INTEGER,INTEGER).

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for BITXOR is: BITXOR(*target_arg*, *bit_mask_arg*).

Example: Querying with the BITXOR Function

In the following query, the input argument 12 has a data type of BYTEINT and a binary representation of 00001100. The input argument 45 has a data type of BYTEINT and a binary representation of 00101101. The bitwise XOR product of the two arguments results in a BYTEINT value of 33, or binary 00100001, which is returned by the query.

```
SELECT BITXOR(12,45);
```

Related Information

- Performing Bit-Byte Operations against Arguments with Non-Equal Lengths.
- *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about UDF implicit type conversion rules, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

COUNTSET

Returns the count of the binary bits within the *target_arg* expression that are either set to 1 or set to 0 depending on the *target_value_arg* value.

COUNTSET Function Syntax

```
[TD_SYSFNLIB.] COUNTSET ( target_arg [, target_value_arg ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

If *target_arg* is NULL, the function returns NULL.

target_value_arg

An integer value. Only a value of 0 or 1 is allowed. If *target_value_arg* is not specified, the default is 1.

If *target_value_arg* is NULL, the function returns NULL.

Argument Types and Rules

COUNTSET is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg* [,*target_value_arg*]) input combinations:

***target_arg* type**	***target_value_arg* type (optional)**
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type is INTEGER.

The result format is the default format for INTEGER.

The default title for COUNTSET is: COUNTSET(*target_arg* [, *target_value_arg*]).

Example: Querying with the COUNTSET Function

The following query takes the input argument 23, which has a data type of BYTEINT and a binary representation of 00010111. Since *target_value_arg* is not specified, the default value of 1 is used. Therefore, the function counts the number of bit values that are set to 1. The query result is an INTEGER value of 4.

```
SELECT COUNTSET(23);
```

Related Information

- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

GETBIT

Returns the value of the bit specified by *target_bit_arg* from the *target_arg* byte expression.

GETBIT Function Syntax

```
[TD_SYSFNLIB.] GETBIT ( target_arg, target_bit_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

If *target_arg* is NULL, the function returns NULL.

target_bit_arg

An integer expression.

The range of input values for *target_bit_arg* can vary from 0 (bit 0 is the least significant bit) to the (sizeof(*target_arg*) - 1).

If *target_bit_arg* is negative or out-of-range (meaning that it exceeds the size of *target_arg*), an error is returned.

If *target_bit_arg* is NULL, the function returns NULL.

Argument Types and Rules

GETBIT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *target_bit_arg*) input combinations:

***target_arg* type**	***target_bit_arg* type**
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

GETBIT returns a BYTEINT value of either 0 or 1, reflecting the value of the bit residing at the *target_bit_arg* position of the *target_arg* byte expression.

The result format is the default format for BYTEINT.

The default title for GETBIT is: GETBIT(*target_arg*, *target_bit_arg*).

Example: Querying with the GETBIT Function

The following query gets the value of the third bit of the input argument 23, which has a data type of BYTEINT and a binary representation of 00010111. The query result is a BYTEINT value of 1 or binary 00000001.

```
   SELECT GETBIT(23,2);
```

Related Information

- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ROTATELEFT

Returns an expression rotated to the left by the number of bits you specify, with the most significant bits wrapping around to the right.

ROTATELEFT Function Syntax

```
[TD_SYSFNLIB.] ROTATELEFT ( target_arg, num_bits_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

num_bits_arg

An integer expression indicating the number of bit positions to rotate.

IF...	THEN the function...
num_bits_arg is equal to zero	returns *target_arg* unchanged.
num_bits_arg is negative	rotates the bits to the right instead of the left.
target_arg and/or *num_bits_arg* are NULL	returns NULL.
num_bits_arg is larger than the size of *target_arg*	rotates (*num_bits_arg* MOD sizeof(*target_arg*)) bits. The scope of the rotation operation is bounded by the size of the *target_arg* expression.

Note:

When operating against an integer value (BYTEINT, SMALLINT, INTEGER, or BIGINT), rotating a bit into the most significant position will result in the integer becoming negative. This is because all integers in Vantage are signed integers.

Argument Types and Rules

ROTATELEFT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *num_bits_arg*) input combinations:

target_arg **type**	***num_bits_arg*** **type**
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table.

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for ROTATELEFT is: ROTATELEFT(*target_arg*, *num_bits_arg*).

Examples

Example: Querying Input Argument 16 with the ROTATELEFT Function

In the following query, the input argument 16 has a data type of BYTEINT and a binary representation of 00010000. When this value is rotated left by two bits, the result in binary is 01000000. This value translates to a BYTEINT value of 64, which is the result returned by the query.

```
SELECT ROTATELEFT(16,2);
```

Example: Querying Input Argument 64 with the ROTATELEDT Function

In the following query, the input argument 64 has a data type of BYTEINT and a binary representation of 01000000. When this value is rotated left by three bits, the result in binary is 00000010. This value translates to a BYTEINT value of 2, which is the result returned by the query.

```
SELECT ROTATELEFT(64,3);
```

Related Information

- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For details about UDF implicit type conversion rules, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ROTATERIGHT

Returns an expression rotated to the right by the number of bits you specify, with the least significant bits wrapping around to the left.

ROTATERIGHT Function Syntax

```
[TD_SYSFNLIB.] ROTATERIGHT ( target_arg, num_bits_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

num_bits_arg

An integer expression indicating the number of bit positions to rotate.

IF...	THEN the function...
num_bits_arg is equal to zero	returns *target_arg* unchanged.
num_bits_arg is negative	rotates the bits to the left instead of the right.
target_arg and/or *num_bits_arg* are NULL	returns NULL.
num_bits_arg is larger than the size of *target_arg*	rotates (*num_bits_arg* MOD sizeof(*target_arg*)) bits. The scope of the rotation operation is bounded by the size of the *target_arg* expression.

Note:

When operating against an integer value (BYTEINT, SMALLINT, INTEGER, or BIGINT), rotating a bit into the most significant position will result in the integer becoming negative. This is because all integers in Vantage are signed integers.

Argument Types and Rules

ROTATERIGHT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *num_bits_arg*) input combinations.

***target_arg* type**	***num_bits_arg* type**
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for ROTATERIGHT is: ROTATERIGHT(*target_arg*, *num_bits_arg*).

For information on default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Examples

Example: Querying Input Argument 32 with the ROTATERIGHT Function

In the following query, the input argument 32 has a data type of BYTEINT and a binary representation of 00100000. When this value is rotated right by two bits, the result in binary is 00001000. This value translates to a BYTEINT value of 8, which is the result returned by the query.

```
SELECT ROTATERIGHT(32,2);
```

Example: Querying Input Argument 4 with the ROTATERIGHT Function

In the following query, the input argument 4 has a data type of BYTEINT and a binary representation of 00000100. When this value is rotated right by four bits, the result in binary is 01000000. This value translates to a BYTEINT value of 64, which is the result returned by the query.

```
SELECT ROTATERIGHT(4,4);
```

Related Information

- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- *Teradata Vantage™ - Data Types and Literals*, B035-1143.

- For details about UDF implicit type conversion rules, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

SETBIT

Sets the value of the bit specified by *target_bit_arg* to the value of *target_value_arg* in the *target_arg* byte expression.

SETBIT Function Syntax

```
[TD_SYSFNLIB.] SETBIT ( target_arg, target_bit_arg [, target_value_arg ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

If *target_arg* is NULL, the function returns NULL.

target_bit_arg

An integer expression.

If *target_bit_arg* is NULL, the function returns NULL.

target_value_arg

An integer value. Only a value of 0 or 1 is allowed. If *target_value_arg* is not specified, the default is 1.

The range of input values for *target_bit_arg* can vary from 0 (bit 0 is the least significant bit) to the (sizeof(*target_arg*) - 1).

If *target_bit_arg* is negative or out-of-range (meaning that it exceeds the size of *target_arg*), an error is returned.

If *target_value_arg* is NULL, the function returns NULL.

Argument Types and Rules

SETBIT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *target_bit_arg* [,*target_value_arg*]) input combinations:

***target_arg* type**	***target_bit_arg* type**	***target_value_arg* type (optional)**
BYTEINT	INTEGER	INTEGER
SMALLINT	INTEGER	INTEGER
INTEGER	INTEGER	INTEGER
BIGINT	INTEGER	INTEGER
VARBYTE(*n*)	INTEGER	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	**THEN the result type is...**	**AND the result format is the default format for...**
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for SETBIT is: SETBIT(*target_arg*, *target_bit_arg*[,*target_value_arg*]).

Examples

Example: Querying with the SETBIT Function

The following query takes the input argument 23, which has a data type of BYTEINT and a binary representation of 00010111, and sets the value of the third bit to 1. The query result is a BYTEINT value of 23 or binary 00010111.

```
SELECT SETBIT(23,2);
```

Example: Querying Input Argument 23 with the ROTATERIGHT Function

The following query takes the input argument 23, which has a data type of BYTEINT and a binary representation of 00010111, and sets the value of the third bit to 0. The query result is a BYTEINT value of 19 or binary 00010011.

```
SELECT SETBIT(23,2,0);
```

Related Information

- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details about UDF implicit type conversion rules, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

SHIFTLEFT

Returns the expression *target_arg* shifted by the specified number of bits (*num_bits_arg*) to the left. The bits in the most significant positions are lost, and the bits in the least significant positions are filled with zeros.

SHIFTLEFT Function Syntax

```
[TD_SYSFNLIB.] SHIFTLEFT ( target_arg, num_bits_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

num_bits_arg

An integer expression indicating the number of bit positions to shift.

IF...	THEN the function...
num_bits_arg is equal to zero	returns *target_arg* unchanged.
num_bits_arg is negative	shifts the bits to the right instead of the left.
target_arg and/or *num_bits_arg* are NULL	returns NULL.
num_bits_arg is larger than the size of *target_arg*	returns an error. The scope of the shift operation is bounded by the size of the *target_arg* expression. Specifying a shift that is outside the range of *target_arg* results in an SQL error.

Note:

When operating against an integer value (BYTEINT, SMALLINT, INTEGER, or BIGINT), shifting a bit into the most significant position will result in the integer becoming negative. This is because all integers in Vantage are signed integers.

Argument Types and Rules

SHIFTLEFT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *num_bits_arg*) input combinations:

***target_arg* type**	***num_bits_arg* type**
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for SHIFTLEFT is: SHIFTLEFT(*target_arg*, *num_bits_arg*).

Example: Querying with the SHIFTLEFT Function

In the following query, the input argument 3 has a data type of BYTEINT and a binary representation of 00000011. When this value is shifted left by two bits, the result in binary is 00001100. This value translates to a BYTEINT value of 12, which is the result returned by the query.

```
SELECT SHIFTLEFT(3,2);
```

Related Information

- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

- For details about UDF implicit type conversion rules, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

SHIFTRIGHT

Returns the expression *target_arg* shifted by the specified number of bits (*num_bits_arg*) to the right. The bits in the least significant positions are lost, and the bits in the most significant positions are filled with zeros.

SHIFTRIGHT Function Syntax

```
[TD_SYSFNLIB.] SHIFTRIGHT ( target_arg, num_bits_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

num_bits_arg

An integer expression indicating the number of bit positions to shift.

IF...	THEN the function...
num_bits_arg is equal to zero	returns *target_arg* unchanged.
num_bits_arg is negative	shifts the bits to the left instead of the right.
target_arg and/or *num_bits_arg* are NULL	returns NULL.
num_bits_arg is larger than the size of *target_arg*	returns an error. The scope of the shift operation is bounded by the size of the *target_arg* expression. Specifying a shift that is outside the range of *target_arg* results in an SQL error.

Note:

When operating against an integer value (BYTEINT, SMALLINT, INTEGER, or BIGINT), shifting a bit out of the most significant position will result in the integer becoming negative. This is because all integers in Vantage are signed integers.

Argument Types and Rules

SHIFTRIGHT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *num_bits_arg*) input combinations:

***target_arg* type**	***num_bits_arg* type**
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	**THEN the result type is...**	**AND the result format is the default format for...**
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for SHIFTRIGHT is: SHIFTRIGHT(*target_arg*, *num_bits_arg*).

Example: Querying with the SHIFTRIGHT Function

In the following query, the input argument 3 has a data type of BYTEINT and a binary representation of 00000011. When this value is shifted right by two bits, the result in binary is 00000000. This value translates to a BYTEINT value of 0, which is the result returned by the query.

```
SELECT SHIFTRIGHT(3,2);
```

Related Information

- For more information on overloaded functions, see "Function Name Overloading" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details about UDF implicit type conversion rules, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

SUBBITSTR

Extracts a bit substring from the *target_arg* string expression starting at the bit position specified by *position_arg*. For the range of bit positions for each data type, see Bit and Byte Numbering Model.

SUBBITSTR Function Syntax

```
[TD_SYSFNLIB.] SUBBITSTR ( target_arg, position_arg, num_bits_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

If *target_arg* is NULL, the function returns NULL.

position_arg

An integer expression indicating the starting position of the bit substring to be extracted.

If *position_arg* is negative or out-of-range (meaning that it exceeds the size of *target_arg*), an error is returned.

If *position_arg* is NULL, the function returns NULL.

num_bits_arg

An integer expression indicating the length of the bit substring to be extracted. This specifies the number of bits for the function to return. Because the return value of the function is a VARBYTE string, the number of bits returned is rounded to the byte boundary greater than the number of bits requested.

The bits returned is right-justified, and the excess bits (those exceeding the requested number of bits) are filled with zeros.

If *num_bits_arg* is negative, or is greater than the number of bits remaining after the starting *position_arg* is taken into account, an error is returned.

If *num_bits_arg* is NULL, the function returns NULL.

Argument Types and Rules

SUBBITSTR is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *position_arg*, *num_bits_arg*) input combinations:

***target_arg* type**	***position_arg* type**	***num_bits_arg* type**
BYTEINT	INTEGER	INTEGER
SMALLINT	INTEGER	INTEGER
INTEGER	INTEGER	INTEGER
BIGINT	INTEGER	INTEGER
VARBYTE(*n*)	INTEGER	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type is a VARBYTE string. The size (number of bytes) of the VARBYTE string depends on the data type of the *target_arg* input argument and the number of bits requested.

For example:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	VARBYTE(1)	VARBYTE(1)
SMALLINT	VARBYTE(2)	VARBYTE(2)
INTEGER	VARBYTE(4)	VARBYTE(4)
BIGINT	VARBYTE(8)	VARBYTE(8)
VARBYTE(*n*)	VARBYTE(*m*) where *m* is the smallest number of bytes to accommodate the requested number of bits.	VARBYTE(*m*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for SUBBITSTR is: SUBBITSTR(*target_arg*, *position_arg*, *num_bits_arg*).

Example: Querying with the SUBBITSTR Function

The following query takes the input argument 20, which has a data type of BYTEINT and a binary representation of 00010100, and requests that 3 bits be returned starting at the third bit. The 3 bits returned are 101, which are placed into a right-justified zero-filled byte. The result from the query is a value of 5, or binary 00000101, with the result data type being VARBYTE(1).

```
SELECT SUBBITSTR(20,2,3);
```

Related Information

- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For details about UDF implicit type conversion rules, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

TO_BYTE

Converts a numeric data type to Vantage byte representation (byte value) of the input value.

The number of bytes returned by the function varies according to the data type of the *target_arg* value.

TO_BYTE Function Syntax

```
[TD_SYSFNLIB.] TO_BYTE ( target_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

target_arg

A numeric or variable byte expression.

If *target_arg* is NULL, the function returns NULL.

Argument Types and Rules

TO_BYTE is an overloaded scalar function. It is defined with the following parameter data types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast.

If the argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result data type is a BYTE value (a fixed byte data type). The size of the byte string returned varies according to the data type of the *target_arg* input argument as shown in the following table.

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTE(1)	BYTE(1)
SMALLINT	BYTE(2)	BYTE(2)
INTEGER	BYTE(4)	BYTE(4)
BIGINT	BYTE(8)	BYTE(8)

The default title for TO_BYTE is: TO_BYTE(*target_arg)*.

Example: Querying with the TO_BYTE Function

In the following query, the input argument 23 has a data type of BYTEINT and a binary representation of 00010111. Performing a TO_BYTE operation on this value results in the hexadecimal value of 00010111 being returned with the data type of BYTE(1).

```
SELECT TO_BYTE(23);
```

Related Information

- For information on the server representation of integral values, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on overloaded functions, see “Function Name Overloading” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

- For details about UDF implicit type conversion rules, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Built-In Functions

Built-in functions return information about the system. Built-in functions are sometimes referred to as special registers.

The built-in functions can be used anywhere that a literal can appear.

If a SELECT statement that contains a built-in function references a table name, then the result of the query contains one row for every row of the table that satisfies the search condition.

ACCOUNT

Returns the account string for the current user.

ACCOUNT Function Syntax

```
ACCOUNT
```

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for ACCOUNT are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Account

Usage Notes

If a SET SESSION ACCOUNT statement has changed the current account string, then the ACCOUNT function returns the new account string based on the request level: whether for an entire session or for an individual request.

Example: Requesting Account Strings for a User

The following statement requests the account string for the current user:

```
    SELECT ACCOUNT;
```

The system responds with something like the following:

```
Account
------------------------------
$M_D2102
```

CURRENT_DATE/CURDATE

Returns the current date.

CURRENT_DATE provides similar functionality to the Teradata function DATE using ANSI-compliant syntax. For information on the Teradata DATE function, see DATE.

CURRENT_DATE/CURDATE Function Syntax

```
{ CURRENT_DATE | CURDATE } [()]
  [ AT { LOCAL | [ TIME ZONE ] { expression | time_zone_string } } ]
```

Syntax Elements

AT LOCAL

The value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled.

If TimeDateWZControl is disabled, the value returned is constructed from the time value local to Vantage and the session time zone.

AT [TIME ZONE] *expression*

Use the time zone displacement defined by expression.

The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE.

AT [TIME ZONE] *time_zone_string*

time_zone_string determines the time zone displacement.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Result Type and Attributes

The result data type, format, and title for CURRENT_DATE are:

Data Type	Format	Title
DATE	Default format for the DATE data type when the Dateform mode is set to IntegerDate. For more information on the default formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.	Date

To convert CURRENT_DATE, use Teradata explicit conversion syntax or ANSI CAST syntax.

Usage Notes

The optional parenthesis is syntax supported by the ODBC driver. For example:

```
SELECT CURRENT_DATE();
```

CURRENT_DATE returns the current date at the time when the request started. If CURRENT_DATE is invoked more than once during the request, the same date is returned. The date returned does not change during the duration of the request.

If you specify CURRENT_DATE without the AT clause or CURRENT_DATE AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, CURRENT_DATE returns a date constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, CURRENT_DATE returns a date constructed from the time value local to Vantage and the session time zone.

For more information, see “DBS Control (dbscontrol)” in *Teradata Vantage™ - Database Utilities*, B035-1102.

CURRENT_DATE returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- CURRENT_DATE is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- CURRENT_DATE is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

Examples

Example: Returning the Current Date for INTERVAL -'08:00' HOUR TO MINUTE

This example assumes that the default format for DATE values is 'yy/mm/dd'. Consider the following statements:

```
    SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
    SELECT CURRENT_DATE AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;
    SELECT CURRENT_DATE AT INTERVAL -'08:00' HOUR TO MINUTE;
    SELECT CURRENT_DATE AT TIME ZONE INTERVAL -'08' HOUR;
    SELECT CURRENT_DATE AT INTERVAL -'08' HOUR;
    SELECT CURRENT_DATE AT TIME ZONE '-08:00';
    SELECT CURRENT_DATE AT '-08:00';
    SELECT CURRENT_DATE AT TIME ZONE '-8';
    SELECT CURRENT_DATE AT '-8';
    SELECT CURRENT_DATE AT TIME ZONE -8;
    SELECT CURRENT_DATE AT -8;
    SELECT CURRENT_DATE AT -8.0;
```

The above SELECT statements return the current date based on the time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. If the current timestamp at UTC is TIMESTAMP '2008-06-01 06:30:00.000000+00:00', these SELECT statements would return '08/05/31' as the date.

If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, and the DBS Control flag TimeDateWZControl is enabled, the statement would return '08/06/01' as the current date based on the current session time and time zone displacement, INTERVAL '01:00' HOUR TO MINUTE. For example:

```
    SELECT CURRENT_DATE;
    SELECT CURRENT_DATE AT LOCAL;
```

```
SELECT CURRENT_DATE();
    SELECT CURRENT_DATE() AT LOCAL;
```

The date returned is not adjusted to account for the start or end of daylight saving time.

Example: Returning the Current Date for INTERVAL -'09:00' HOUR TO MINUTE

This example assumes that the default format for DATE values is 'yy/mm/dd'. Consider the following statements:

```
    SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
    SELECT CURRENT_DATE AT INTERVAL '09:00' HOUR TO MINUTE;
```

The above SELECT statement returns the current date based on the time zone displacement, INTERVAL '09:00' HOUR TO MINUTE. If the current timestamp at UTC is TIMESTAMP '2008-06-01 19:30:00.000000+00:00', the SELECT statement would return '08/06/02' as the date.

If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, and the DBS Control flag TimeDateWZControl is enabled, the statement would return
'08/06/01' as the current date based on the current session time and time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The date returned is not adjusted to account for the start or end of daylight saving time.

Example: Returning the Current Date for INTERVAL -'05:45' HOUR TO MINUTE

This example assumes that the default format for DATE values is 'yy/mm/dd'. Consider the following statements:

```
SET TIME ZONE INTERVAL '10:00' HOUR TO MINUTE;
SELECT CURRENT_DATE AT '05:45';
SELECT CURRENT_DATE AT 5.75;
```

The above SELECT statements return the current date based on the time zone displacement, INTERVAL '05:45' HOUR TO MINUTE. If the current timestamp at UTC is TIMESTAMP '2008-06-01 17:30:00.000000+00:00', the SELECT statements would return '08/06/01' as the date.

If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, and the DBS Control flag TimeDateWZControl is enabled, the statement would return
'08/06/02' as the current date based on the current session time and time zone displacement, INTERVAL '10:00' HOUR TO MINUTE.

The date returned is not adjusted to account for the start or end of daylight saving time.

Example: Returning the Current Date for the Time Zone String, 'America Pacific'

The following queries return the current date at the time zone displacement based on the time zone string, 'America Pacific'. Vantage determines the time zone displacement based on the time zone string and the CURRENT_TIMESTAMP AT '00:00' (that is, at UTC). The date returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT CURRENT_DATE AT TIME ZONE 'America Pacific';
SELECT CURRENT_DATE AT 'America Pacific';
```

Example: Changing the Default Output Format

To change the default output format of the CURRENT_DATE result, use Teradata explicit conversion syntax and specify the FORMAT phrase. For example, the following statement requests the current date and specifies a format that is different from the default:

```
SELECT CURRENT_DATE (FORMAT 'MMMBDD,BYYYY');
```

The result is similar to:

```
        Date
------------
May 31, 2007
```

Related Information

- For information on default data type formats and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For an example that uses Teradata explicit conversion syntax to change the default output format, see Example: Changing the Default Output Format.
- For information about time zone strings, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

CURRENT_ROLE

Returns the current role of the current authorized user.

CURRENT_ROLE is not supported in the FastLoad and MultiLoad utilities.

CURRENT_ROLE Function Syntax

```
CURRENT_ROLE
```

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for CURRENT_ROLE are as follows.

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Current_Role

CURRENT_ROLE Usage Notes

Result Value

If you are not accessing Vantage through a proxy connection, CURRENT_ROLE functions exactly like the ROLE built-in function and returns the session current role, which is the current role of the session user. For more information, see ROLE.

If you are accessing Vantage through a proxy connection, then CURRENT_ROLE returns the current role of the proxy user as shown in the following table.

IF the current role for the session is …	THEN the result value is …
a role set by PROXYROLE	the name of the role.
the default	If there is one proxy role in the CONNECT THROUGH privilege of the proxy user, the result value is the name of the role. If there are multiple proxy roles in the CONNECT THROUGH privilege of the proxy user, the result value is ALL.
PROXYROLE=ALL	ALL
PROXYROLE=NONE or NULL	NULL

Example: Selecting CURRENT_ROLE

You can identify the current role for the current authorized user with the following statement:

```
SELECT CURRENT_ROLE;
```

The system displays a response similar to the following:

```
Current_Role
------------------------------
Buyers_role
```

CURRENT_TIME/CURTIME

Returns the current time.

The fields in CURRENT_TIME are:

- HOUR
- MINUTE
- SECOND

  The seconds precision of the result of CURRENT_TIME is limited to hundredths of a second. CURRENT_TIME returns zeros for any digits to the right of the two most significant digits in the fractional portion of seconds.
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

CURRENT_TIME provides similar functionality to the Teradata function TIME using ANSI-compliant syntax. For information on the Teradata TIME function, see TIME.

CURRENT_TIME/CURTIME Function Syntax

```
{ CURRENT_TIME | CURTIME () } [ ( fractional_precision ) ]
  [ AT { LOCAL | [ TIME ZONE ] { expression | time_zone_string } } ]
```

Syntax Elements

fractional_precision

A precision range for the returned value. The valid range is 0 through 6. The default is 0.

AT LOCAL

The value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled.

If TimeDateWZControl is disabled, the value returned is constructed from the time value local to Vantage and the session time zone.

AT [TIME ZONE] *expression*

Use the time zone displacement defined by expression.

The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE.

AT [TIME ZONE] *time_zone_string*

time_zone_string determines the time zone displacement.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Result Type and Attributes

The result data type, format, and title for CURRENT_TIME are:

Data Type	Format	Title
TIME WITH TIME ZONE	Default format for the TIME WITH TIME ZONE data type.	Current Time (*fractional_precision*)

To convert CURRENT_TIME, use Teradata explicit conversion syntax or ANSI CAST syntax.

Usage Notes

CURRENT_TIME returns the current time when the request started. If CURRENT_TIME is invoked more than once during the request, the same time is returned. The time returned does not change during the duration of the request.

If you specify CURRENT_TIME without the AT clause or CURRENT_TIME AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, CURRENT_TIME returns a time constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, CURRENT_TIME returns a time constructed from the time value local to Vantage and the session time zone.

The following occurs when using CURRENT_TIME with daylight saving time (DST):

- When TimeDateWZControl flag is enabled and the session time zone is defined with a time zone string that follows different Daylight Saving Time and standard time zone displacements, then CURRENT_TIME AT LOCAL gives TIME value corresponding to Standard time zone rather than the Daylight Saving Time time zone.
- During the Daylight Saving Time period, CURRENT_TIME AT LOCAL follows Standard time zone, whereas CURRENT_TIME follows the Daylight Saving Time zone. During Standard time zone period, both CURRENT_TIME and CURRENT_TIME AT LOCAL follow the standard time zone.

For example, during DST period:

```
BTEQ -- Enter your SQL request or BTEQ command:
.os date
Sat Sep  5 14:00:23 PDT 2015
```

```
BTEQ -- Enter your SQL request or BTEQ command:
.os date -u
Sat Sep  5 21:00:26 UTC 2015

BTEQ -- Enter your SQL request or BTEQ command:
select current_time, current_time at local;


*** Query completed. One row found. 2 columns returned.
 *** Total elapsed time was 1 second.

Current Time(0)  Current Time(0) AT LOCAL
---------------  ------------------------
14:00:30-07:00            13:00:30-08:00
```

During Standard Time Zone Period:

```
BTEQ -- Enter your SQL request or BTEQ command:
.os date
Fri Nov  6 04:18:04 PST 2015

BTEQ -- Enter your SQL request or BTEQ command:
.os date -u
Fri Nov  6 12:18:07 UTC 2015

BTEQ -- Enter your SQL request or BTEQ command:
select current_time, current_time at local;


*** Query completed. One row found. 2 columns returned.
 *** Total elapsed time was 1 second.

Current Time(0)  Current Time(0) AT LOCAL
---------------  ------------------------
04:18:20-08:00            04:18:20-08:00
```

Note:

If CURRENT_TIME is used in a stored procedure, the procedure must be recompiled whenever the DBS Control fields System TimeZone Hour or System TimeZone Minute are changed. Recompiling stored procedures is not necessary if a time zone string is set using the tdlocaledef utility.

Examples

Example: Requesting the Current Time

If the DBS Control flag TimeDateWZControl is enabled, the following statements request the current time based on the current session time and time zone.

```
SELECT CURRENT_TIME;
SELECT CURRENT_TIME AT LOCAL;
```

The result is similar to:

```
Current Time(0)
---------------
 15:53:34+00:00
```

When TimeDateWZControl flag is enabled and the session time zone is defined with a time zone string that follows different Daylight Saving Time and standard time zone displacements, then CURRENT_TIME AT LOCAL gives TIME value corresponding to Standard time zone rather than the Daylight Saving Time time zone.

During the Daylight Saving Time period, CURRENT_TIME AT LOCAL follows Standard time zone, whereas CURRENT_TIME follows the Daylight Saving Time zone. During Standard time zone period, both CURRENT_TIME and CURRENT_TIME AT LOCAL follow the standard time zone.

Example: Requesting the Current Time with a Time Zone String

The following queries return the current time at the time zone displacement based on the time zone string, 'America Pacific'. The time returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT CURRENT_TIME AT TIME ZONE 'America Pacific';
SELECT CURRENT_TIME AT 'America Pacific';
```

Example: Changing the Default Output Format

To change the default output format of the CURRENT_TIME result, use Teradata explicit conversion syntax and specify the FORMAT phrase. For example, the following statement requests the current time and specifies a format that is different from the default:

```
SELECT CURRENT_TIME (FORMAT 'HH:MIBT');
```

The result looks like this:

```
Current Time(0)
---------------
       02:29 PM
```

Related Information

- For information about AT LOCAL and AT TIME ZONE time zone specifiers, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For information on default data type formats and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For an example that uses Teradata explicit conversion syntax to change the default output format, see Example: Changing the Default Output Format.
- For details, see “CREATE PROCEDURE (SQL Form)” in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and “DBS Control (dbscontrol)” in *Teradata Vantage™ - Database Utilities*, B035-1102.

CURRENT_TIMESTAMP

Returns the current timestamp.

The fields in CURRENT_TIMESTAMP are:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND

  The seconds precision of the result of CURRENT_TIMESTAMP is limited to hundredths of a second. CURRENT_TIMESTAMP returns zeros for any digits to the right of the two most significant digits in the fractional portion of seconds.
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

CURRENT_TIMESTAMP Function Syntax

```
CURRENT_TIMESTAMP [ ( fractional_precision ) ]
  [ AT { LOCAL | [ TIME ZONE ] { expression | time_zone_string } } ]
```

Syntax Elements

fractional_precision

A precision range for the returned value. The valid range is 0 through 6. The default is 0.

AT LOCAL

The value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled.

If TimeDateWZControl is disabled, the value returned is constructed from the time value local to Vantage and the session time zone.

AT [TIME ZONE] *expression*

Use the time zone displacement defined by expression.

The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE.

AT [TIME ZONE] *time_zone_string*

time_zone_string determines the time zone displacement.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Result Type and Attributes

The result data type, format, and title for CURRENT_TIMESTAMP are:

Data Type	Format	Title
TIMESTAMP WITH TIME ZONE	Default format for the TIMESTAMP WITH TIME ZONE data type. For more information on the default formats, see "Data Type Formats and Format Phrases" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.	Current TimeStamp (*fractional_precision*)

To convert CURRENT_TIMESTAMP, use Teradata explicit conversion syntax or ANSI CAST syntax. For an example that uses Teradata explicit conversion syntax to change the default output format, see Example: Changing the Default Output Format.

Usage Notes

CURRENT_TIMESTAMP returns the current timestamp when the request started. If CURRENT_TIMESTAMP is invoked more than once during the request, the same timestamp is returned. The timestamp returned does not change during the duration of the request.

If you specify CURRENT_TIMESTAMP without the AT clause or CURRENT_TIMESTAMP AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, CURRENT_TIMESTAMP returns a timestamp constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, CURRENT_TIMESTAMP returns a timestamp constructed from the time value local to Vantage and the session time zone.

CURRENT_TIMESTAMP returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- CURRENT_TIMESTAMP is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- CURRENT_TIMESTAMP is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

Note:

If CURRENT_TIMESTAMP is used in a stored procedure, the procedure must be recompiled whenever the DBS Control fields System TimeZone Hour or System TimeZone Minute are changed. Recompiling stored procedures is not necessary if a time zone string is set using the tdlocaledef utility.

Examples

Example: Requesting the Current Timestamp

If the DBS Control flag TimeDateWZControl is enabled, the following statements request the current timestamp based on the current session time and time zone.

```
SELECT CURRENT_TIMESTAMP;
SELECT CURRENT_TIMESTAMP AT LOCAL;
```

The result is similar to:

```
            Current TimeStamp(6)
--------------------------------
2001-11-27 15:53:34.910000+00:00
```

If the session time zone was defined with a time zone string that follows different DST and standard time zone displacements, then the timestamp returned is automatically adjusted to account for the start and end of daylight saving time. Otherwise, no adjustment for daylight saving time is done.

Example: CURRENT_TIMESTAMP and the TimeDateWZControl Flag

This example shows the effect of the DBS Control flag TimeDateWZControl on the results returned by CURRENT_TIMESTAMP when the function is specified without an AT clause or with an AT LOCAL clause.

Assume the following:

- The time local to Vantage is 11:59:00 Coordinated Universal Time (UTC), January 31, 2010.
- User TK lives in Tokyo, and has a time zone defined as +9 hours offset from UTC.
- User LA lives in Los Angeles, and has a time zone defined as -8 hours offset from UTC.
- User TK and User LA run the CURRENT_TIMESTAMP function at exactly the same time.

If the TimeDateWZControl flag is enabled:

For User TK, the CURRENT_TIMESTAMP function returns:

```
2010-02-01 10:59:00.000000+09:00
```

For User LA, the CURRENT_TIMESTAMP function returns:

```
2010-01-31 16:59:00.000000-08:00
```

If the TimeDateWZControl flag is disabled:

For User TK, the CURRENT_TIMESTAMP function returns:

```
2010-01-31 11:59:00.000000+09:00
```

For User LA, the CURRENT_TIMESTAMP function returns:

```
2010-01-31 11:59:00.000000-08:00
```

Example: Requesting the Current Timestamp with a Time Zone String

The following queries return the current timestamp at the time zone displacement based on the time zone string, 'America Pacific'. The timestamp returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT CURRENT_TIMESTAMP AT TIME ZONE 'America Pacific';
SELECT CURRENT_TIMESTAMP AT 'America Pacific';
```

Example: Changing the Default Output Format

To change the default output format of the CURRENT_TIMESTAMP result, use Teradata explicit conversion syntax and specify the FORMAT phrase. For example, the following statement requests the current timestamp and specifies a format that is different from the default:

```
SELECT CURRENT_TIMESTAMP (FORMAT 'MMMBDD,BYYYYBHH:MIBT');
```

The result looks like this:

```
 Current TimeStamp(6)
---------------------
Feb 19, 2002 07:45 am
```

Related Information

- For information about time zone strings, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- Example: Changing the Default Output Format
- “CREATE PROCEDURE (SQL Form)” in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- “DBS Control (dbscontrol)” in *Teradata Vantage™ - Database Utilities*, B035-1102
- For information on default data type formats and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

CURRENT_USER

Provides the user name of the current authorized user.

CURRENT_USER Function Syntax

```
CURRENT_USER
```

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for CURRENT_USER are as follows.

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Current_User

CURRENT_USER Usage Notes

Result Value

If you are accessing Vantage through a proxy connection, CURRENT_USER returns the proxy user name. Otherwise, it functions exactly like the USER built-in function and returns the session user name. For more information, see USER.

Examples

Example: Identifying the Current User

You can identify the current authorized user with the following statement:

```
SELECT CURRENT_USER;
```

The system responds with something like the following:

```
Current_User
------------------------------
BO-JSMITH
```

Example: Selecting the Job Title for the Current User

The following example selects the job title for the current authorized user:

```
SELECT JobTitle FROM Employee WHERE Name = CURRENT_USER;
```

DATABASE

Returns the name of the default database for the current user.

If a DATABASE request has changed the current default database, then the DATABASE function returns the new name of the default.

DATABASE Function Syntax

```
DATABASE
```

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for DATABASE are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Database

Example: Requesting the Name of the Default Database

The following statement requests the name of the default database:

```
    SELECT DATABASE;
```

The system responds with something like the following:

```
    Database
    ------------------------------
    Customer_Service
```

DATE

Returns the current date.

DATE Function Syntax

```
DATE [ AT { LOCAL | [ TIME ZONE ] { expression | time_zone_string } } ]
```

Syntax Elements

AT LOCAL

The value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled.

If TimeDateWZControl is disabled, the value returned is constructed from the time value local to Vantage and the session time zone.

AT [TIME ZONE] ***expression***

Use the time zone displacement defined by expression.

The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE.

AT [TIME ZONE] ***time_zone_string***

time_zone_string determines the time zone displacement.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Result Type and Attributes

The default format of DATE depends on the value of the Dateform mode. The data type, format, and title for DATE are as follows:

Data Type	Format		Title
	Dateform Mode	DATE Function Format	
DATE	INTEGERDATE	the default format for DATE data types as specified in the SDF.	Date
	ANSIDATE	'YYYY-MM-DD'	

Usage Notes

DATE returns the current date at the time when the request started. If DATE is invoked more than once during the request, the same date is returned. The date returned does not change during the duration of the request.

If you specify DATE without the AT clause or DATE AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, DATE returns a date constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, DATE returns a date constructed from the time value local to Vantage and the session time zone.

DATE returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- DATE is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- DATE is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

DATE cannot appear as the first argument in a user-defined method invocation.

DATE versus CURRENT_DATE

DATE is deprecated. Use the ANSI SQL:2011 compliant CURRENT_DATE function instead.

Examples

Example 1: Returning the Current Date Based on INTERVAL -'08:00' HOUR TO MINUTE

This example assumes that the default format for DATE values is 'yy/mm/dd'. Consider the following statements:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT DATE AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;
SELECT DATE AT INTERVAL -'08:00' HOUR TO MINUTE;
SELECT DATE AT TIME ZONE INTERVAL -'08' HOUR;
SELECT DATE AT INTERVAL -'08' HOUR;
SELECT DATE AT TIME ZONE '-08:00';
SELECT DATE AT '-08:00';
SELECT DATE AT TIME ZONE '-8';
SELECT DATE AT '-8';
SELECT DATE AT TIME ZONE -8;
SELECT DATE AT -8;
SELECT DATE AT -8.0;
```

The above SELECT statements return the current date based on the time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. If the current timestamp at UTC is TIMESTAMP '2008-06-01 06:30:00.000000+00:00', these SELECT statements would return '08/05/31' as the date.

If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, and the DBS Control flag TimeDateWZControl is enabled, the statement would return

'08/06/01' as the current date based on the current session time and time zone displacement, INTERVAL '01:00' HOUR TO MINUTE. For example:

```
SELECT DATE;
SELECT DATE AT LOCAL;
```

The date returned is not adjusted to account for the start or end of daylight saving time.

Example 2: Changing the Presentation by 'mm-dd-yy'

Use the FORMAT phrase to change the presentation:

```
SELECT DATE (FORMAT 'mm-dd-yy');
    Date
--------
03-30-96
```

Example 3: Returning the Current Date Based on Time Zone String 'America Pacific'

The following queries return the current date at the time zone displacement based on the time zone string, 'America Pacific'. Vantage determines the time zone displacement based on the time zone string and the CURRENT_TIMESTAMP AT '00:00' (that is, at UTC). The date returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT DATE AT TIME ZONE 'America Pacific';
SELECT DATE AT 'America Pacific';
```

Example 4: Changing the Presentation by 'mmmbdd,byyyy'

Another form gives:

```
SELECT DATE (FORMAT 'mmmbdd,byyyy');
        Date
------------
Mar 30, 1996
```

Related Information

- For information about dbscontrol, see “DBS Control (dbscontrol)” in *Teradata Vantage™ - Database Utilities*, B035-1102.
- For information about time zone strings, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

- For information on default data type formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For information about DATE versus CURRENT_DATE, see CURRENT_DATE/CURDATE.

NOW

Returns current date and current time as a TIMESTAMP value.

NOW Function Syntax

```
NOW ()
```

ANSI Compliance

This statement is ANSI SQL:2011, but includes non-ANSI Vantage extensions.

Result Type and Attributes

The result data type, format, and title are:

Data Type	Format	Title
TIMESTAMP(0)	Default format for date, followed by time without a fraction of a second.	NOW()

Example

Example of NOW function usage:

```
SELECT NOW()
```

Related Information

CURRENT_DATE/CURDATE and CURRENT_TIME/CURTIME.

PROFILE

Returns the current profile for the session or NULL if none.

PROFILE Function Syntax

```
PROFILE
```

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for PROFILE are as follows.

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Profile

Example

You can identify the current profile for the session with the following statement:

```
    SELECT PROFILE ;
```

ROLE

Returns the session current role.

ROLE is not supported in the FastLoad and MultiLoad utilities.

ROLE Function Syntax

```
ROLE
```

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for ROLE are as follows.

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Role

ROLE Usage Notes

Result Value

The session logon can be not directory-based or directory-based.

If you are accessing Vantage through a proxy connection, and you want to get the current role of the proxy user, use the CURRENT_ROLE built-in function.

Session logon is not directory-based

If the session logon is not directory-based, refer to the following table.

Current Role for the Session	Result Value
Existing role	Name of the role
ALL	'ALL'
NONE or NULL	NULL

Session logon is directory-based

If the session logon is directory-based, refer to the following table.

Session	Result Value
Assigned a set of directory-managed roles and does not change the current role	'EXTERNAL'
Uses a SET ROLE EXTERNAL statement	
• does not have an assigned set of directory-managed roles, • maps to a permanent user that has a default database-managed role, and • does not change the current role	Name of the default role of the permanent user
Uses a SET ROLE *role_name* statement, where *role_name* is either a directory-managed or database-managed role	Name of the specified role
Uses a SET ROLE ALL statement	'ALL'
• Not assigned a set of directory-managed roles, • Does not change the current role, and one of the following condition is true” Directory-based logon does not map to a permanent user Permanent user that the directory-based logon maps to does not have a default database-managed role	NULL
Uses a SET ROLE NONE statement	

Session	Result Value
Uses a SET ROLE NULL statement	

Example: Identifying the Session Current Role

You can identify the session current role with the following statement:

```
   SELECT ROLE;
```

The system responds with something like the following:

```
   Role
   ------------------------------
   EXTERNAL
```

SESSION

Returns the number of the session for the current user.

SESSION Function Syntax

```
SESSION
```

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for SESSION are as follows.

Data Type	Format	Session
INTEGER	Default format for the INTEGER data type. For more information on the default formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.	Session

Example: Identifying the Session Number for the Current User

The following statement identifies the number of the session for the current user:

```
SELECT SESSION;
```

The system responds with something like the following:

```
    Session
-----------
       1048
```

TEMPORAL_DATE

Returns the current transaction date where the evaluation is based on the session time zone.

TEMPORAL_DATE is not supported in a partitioning expression for the PARTITION BY clause that defines a partitioned primary index.

TEMPORAL_DATE Function Syntax

```
TEMPORAL_DATE
```

Result Type and Attributes

The result data type, format, and title for TEMPORAL_DATE are as follows.

Data Type	Format	Title
DATE	Default format for the DATE data type when the Dateform mode is set to IntegerDate. For details on default formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.	Date

Usage Notes

The value of TEMPORAL_DATE is the same for all requests submitted in a single transaction.

The system uses the session time zone to evaluate TEMPORAL_DATE.

When TEMPORAL_DATE appears in a CHECK constraint or DEFAULT clause, the result value is evaluated when the request applies the CHECK constraint (during an insert or update) or when the request uses the DEFAULT value for a given column.

TEMPORAL_TIMESTAMP

Returns the current transaction timestamp where the evaluation is based on the session time zone.

The seconds precision of the result of TEMPORAL_TIMESTAMP is limited to hundredths of a second. TEMPORAL_TIMESTAMP returns zeros for any digits to the right of the two most significant digits in the fractional portion of seconds.

TEMPORAL_TIMESTAMP Function Syntax

```
TEMPORAL_TIMESTAMP [ ( precision ) ]
```

Syntax Elements

precision

A precision range for the returned value. The valid range is 0 through 6. The default is 0.

Result Type and Attributes

The result data type, format, and title for TEMPORAL_TIMESTAMP are as follows.

Data Type	Format	Title
TIMESTAMP(*n*) WITH TIME ZONE, where *n* is the same as the *precision* argument or 6 if omitted	Default format for the TIMESTAMP WITH TIME ZONE type.	Timestamp

Usage Notes

The value of TEMPORAL_TIMESTAMP is the same for all requests submitted in a single transaction.

The system uses the session time zone to evaluate TEMPORAL_TIMESTAMP.

When TEMPORAL_TIMESTAMP appears in a CHECK constraint or DEFAULT clause, the result value is evaluated when the request applies the CHECK constraint (during an insert or update) or when the request uses the DEFAULT value for a given column.

Related Information

- ANSI Temporal Table Support *Teradata Vantage™ - Temporal Table Support*, B035-1182.
- For details on default formats, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

TIME

Returns the current time.

TIME is deprecated. Use the ANSI SQL:2011 compliant CURRENT_TIME function instead.

TIME Function Syntax

```
TIME [ AT { LOCAL | [ TIME ZONE ] { expression | time_zone_string } } ]
```

Syntax Elements

LOCAL

The value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled.

If TimeDateWZControl is disabled, the value returned is constructed from the time value local to Vantage and the session time zone.

TIME ZONE

Use the time zone displacement defined by *expression*.

expression

The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE.

time_zone_string

time_zone_string determines the time zone displacement.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Result Type and Attributes

The data type, format, and title for TIME are as follows:

Data Type	Format	Title
FLOAT	HHMMSS.CC (hours, minutes, seconds, hundredths of a second)	Time

Usage Notes

TIME returns the current time when the request started. If TIME is invoked more than once during the request, the same time is returned. The time returned does not change during the duration of the request.

If you specify TIME without the AT clause or TIME AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, TIME returns a time constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, TIME returns a time constructed from the time value local to Vantage and the session time zone.

TIME data is stored internally in UTC, which can affect how TIME result values sort.

TIME returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- TIME is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- TIME is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

TIME cannot appear as the first argument in a user-defined method invocation.

Examples

Example 1: Requesting the Current Time by Session Time and Time Zone

If the DBS Control flag TimeDateWZControl is enabled, the following statements request the current time based on the current session time and time zone.

```
    SELECT TIME;
    SELECT TIME AT LOCAL;
```

The result is similar to:

```
    Time
--------
16:20:20
```

If the session time zone was defined with a time zone string that follows different DST and standard time zone displacements, then the time returned is automatically adjusted to account for the start and end of daylight saving time. Otherwise, no adjustment for daylight saving time is done.

Example 2: Returning the Current Time Based on Time Zone String, 'America Pacific'

The following queries return the current time at the time zone displacement based on the time zone string, 'America Pacific'. The time returned is automatically adjusted to account for the start and end of daylight saving time.

```
    SELECT TIME AT TIME ZONE 'America Pacific';
    SELECT TIME AT 'America Pacific';
```

Example 3: Displaying Hundredths of a Second

The hundredths of a second are not displayed by the default format, but you can use the FORMAT phrase to display it:

```
    SELECT TIME (FORMAT '99:99:99.99');
```

The system responds with something like the following:

```
       Time
-----------
16:26:30.19
```

Example 4: Inserting a Row in a Table

The following example inserts a row in a hypothetical table in which the column InsertTime has data type FLOAT and records the time that the row was inserted:

```
    INSERT INTO HypoTable (ColumnA, ColumnB, InsertTime)
    VALUES ('Abcde', 12345, TIME);
```

Related Information

- For more information, see “DBS Control (dbscontrol)” in *Teradata Vantage™ - Database Utilities*, B035-1102.
- For information about how TIME data is stored internally in UTC, see “ORDER BY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- For information about time zone strings, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For information about TIME versus CURRENT_TIME, see CURRENT_TIME/CURTIME.

USER

Provides the session user name.

IF the session logon is …	THEN …
not directory-based	the result value is the session user name.
directory-based	If the session maps to a permanent table, the result value is the name of the permanent user. If the session does not map to a permanent user, the result value is the authcid of the external user.

USER Function Syntax

```
USER
```

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for USER are as follows.

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	User

Examples

Example: Identifying the User Name

You can identify the session user name with the following statement:

```
    SELECT USER;
```

The system responds with something like the following.

```
    User
    ------------------------------
    JJ43901
```

Example: Selecting the User Job Title

The following example selects the job title for the session user.

```
    SELECT JobTitle FROM Employee WHERE Name = USER;
```

Related Information

CURRENT_USER.

Comparison Operators and Functions

The following sections describe SQL comparison operators.

Comparison Operators

Comparison operators, which are types of logical predicates (also called conditional expressions), test the truth of relations between expressions.

- IF, WHILE, REPEAT, and CASE statements in stored procedures
- WHEN clauses in searched CASE expressions
- WHERE, ON, and HAVING clauses to qualify or disqualify rows in a SELECT statement
- CASE_N functions

Comparison Operator Syntax

```
{ expression_1 operator expression_2 |
  expression_1 operator quantifier ( literal [,...] ) |
  expression_1 operator [ quantifier ] ( subquery ) |
  ( expression_1 [,...] ) operator [ quantifier ] ( subquery )
}
```

Syntax Elements

expression_1

An SQL scalar expression, including derived period expressions.

Comparison operators do not support BLOB or CLOB type expressions. You can explicitly cast BLOBs to BYTE or VARBYTE and cast CLOBs to CHARACTER or VARCHAR, and use the result with comparison operators.

An expression that results in a UDT data type can only be compared with another expression that results in the same UDT data type.

operator

A comparison operator—see Supported Comparison Operators.

expression_2

An SQL scalar expression, including derived period expressions.

quantifier

One of the following quantifier keywords:

- ANY
- SOME
- ALL

literal

Any of the following:

- Defined value
- Macro parameter
- Built-in value such as TIME, DATE, or USER

The comparison operation may compare an expression against a list of explicit literals.

The data types of expression and literal must be compatible. If the data types of the operands differ, Vantage performs an implicit conversion from one type to another in some cases.

subquery

An SQL SELECT statement.

Using a subquery in a condition is restricted in certain cases.

ANSI Compliance

The following comparison operators are ANSI SQL:2011 compliant.

• = • >	• < • <>	• <= • >=

The following comparison operators are Teradata extensions to the ANSI SQL:2011 standard.Their use is deprecated.

• EQ • ^= • NE	• NOT= • LT • LE	• GT • GE

Results

A logical expression that uses a comparison operator evaluates to TRUE, FALSE, or UNKNOWN.

Comparison Operators Usage Notes

Supported Comparison Operators

Vantage supports the following comparison operators.

ANSI Operator	Teradata Extensions	Function
=	EQ	Tests for equality.
<>	^= NE NOT=	Tests for inequality.
<	LT	Tests for less than.
<=	LE	Tests for less than or equal.
>	GT	Tests for greater than.
>=	GE	Tests for greater than or equal.

Comparison Operators Using Subqueries

A subquery is a SELECT statement that returns values used to satisfy the comparison operation. The subquery must be enclosed in parentheses, and it does not end with a semicolon.

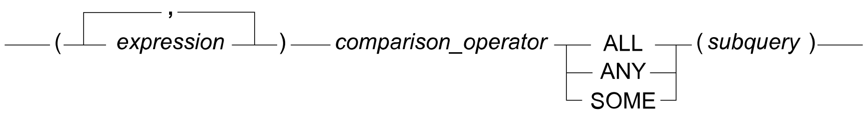
The subquery must refer to at least one table. A table that is in the WHERE clause, but that is not referred to in any other parts of the subquery, is not applicable.

A comparison operation may be used with a subquery whether or not a quantifier is used. If a quantifier is not used, however, then an error condition results if the subquery returns more than one value.

If a subquery returns no values, and if a quantifier is not used, then the result of the comparison is false. Therefore, if the following form is used, the subquery must return either no values (in which case the comparison evaluates to false), or it returns one value.

```
expression > (subquery)
```

With the following form, subquery must select the same number of expressions as are specified in the expression list.



The two expression lists are equal if each of the respective expressions are equal.

If the respective expressions are not equal, then the result of the comparison is determined by comparing the first pair of expressions (from the left) for which the comparison is not true.

A subquery in a comparison operation cannot specify a SELECT AND CONSUME statement.

Example: Using the ALL Quantifier to Compare Two Expressions

The following statement uses the ALL quantifier to compare two expressions with the values returned from a subquery to find the employee(s) with the most years of experience in the group of employees having the highest salary:

```
SELECT EmpNo, Name, DeptNo, JobTitle, Salary, YrsExp
FROM Employee
WHERE (Salary,YrsExp) >= ALL
 (SELECT Salary,YrsExp FROM Employee) ;
```

Related Information

- Supported Comparison Operators
- Implicit Type Conversion of Comparison Operands
- ANY/ALL/SOME

Comparisons That Produce TRUE Results

Conditions

The following table provides the conditions when comparisons produce TRUE results.

For simplicity, assume the syntax:

```
expression_1  operator  expression_2
```

expression_1 and *expression_2* must contain the same number of scalar values and range from 1 through *n* rows, represented by *r*, so that the *r*th components of *expression_1* and *expression_2* are *expression_1*r and *expression_2*r.

The δth item in the range is notated as row δ such that the δth component of *expression_1* is notated as *expression_1*δ and the δth component of *expression_2* is notated as *expression_2*δ.

The data types of *expression_1* and *expression_2* must be compatible. If the data types of the expressions differ, Vantage performs an implicit conversion from one type to another in some cases.

This comparison …	Is TRUE if …
expression_1 = *expression_2*	∀ *r*, *expression_1*r = *expression_2*r is TRUE.
expression_1 <> *expression_2*	∃ δ such that *expression_1*δ <> *expression_2*δ is TRUE.
expression_1 < *expression_2*	∃ δ such that *expression_1*δ < *expression_2*δ is TRUE and for all *r* < δ, *expression_1*r = *expression_2*r is TRUE.
expression_1 > *expression_2*	∃ δ such that *expression_1*δ >*expression_2*δ is TRUE and for all *r* > δ, *expression_1*r = *expression_2*r is TRUE.
expression_1 <= *expression_2*	*expression_1* < *expression_2* is TRUE or *expression_1* = *expression_2* is TRUE.
expression_1 => *expression_2*	*expression_1* > *expression_2* is TRUE or *expression_1* = *expression_2* is TRUE.

Null Expressions

If any expression in a comparison is null, the result of the comparison is unknown.

For a comparison to provide a TRUE result when comparing fields that might result in nulls, the statement must include the IS [NOT] NULL operator.

Floating Point Expressions

Calculations involving floating point values often produce results that are not what you expect. If you perform a floating point calculation and then compare the results against some expected value, it is unlikely that you get the intended result.

Instead of comparing the results of a floating point calculation, make sure that the result is greater or less than what is needed, with a given error. Here is an example:

```
SELECT i, SUM(a) as sum_a, SUM(b) as sum_b
FROM t1
GROUP BY i
HAVING ABS(sum_a - sum_b) > 1E-10;
```

Related Information

- For more information on potential problems associated with floating point values in comparison operations, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details about *expression_1* and *expression_2*, see Implicit Type Conversion of Comparison Operands.
- For more information about using comparison operators in conditional expressions in searched CASE expressions, see CASE Expressions.

- For more information about using comparison operators in conditional expressions in WHERE, ON, or HAVING clauses in SELECT statements, see “The SELECT Statement” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- For more information about using comparison operators in conditional expressions in IF, WHILE, or REPEAT statements in stored procedures, see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.
- For more information about other comparison operators, including the following, see Logical Predicates:
    - [NOT] EXISTS
    - [NOT] IN
    - LIKE
    - IS [NOT] NULL
    - [NOT] BETWEEN … AND …
- For more information about the following predicate quantifiers, see Logical Predicates:
    - ALL
    - ANY
    - SOME

Data Type Evaluation

Different data types define equality and inequality differently. The following table explains the foundations for how the various data types are compared:

This data type …	Is evaluated in this way …
Numeric	Algebraically, with negatives considered to be smaller irrespective of their absolute value.
Byte	Bit-by-bit from left to right. A 0 bit is less than a 1 bit. • If every pair comparison is equal, then the two byte string are equal. • If any pairwise comparison is not equal, then that comparison determines the result. • If two byte strings of different lengths are compared, then the shorter string is padded to the right with binary zeros to make the lengths equal prior to making the comparison.
Character	Character-by-character from left to right. Exact comparisons depend on the collation sequence assigned and whether the comparison is case specific or case blind. The available collations are: • ASCII • EBCDIC • MULTINATIONAL • CHARSET_COLL • JIS_COLL If every pairwise comparison is equal, then the two character strings are equal. If any pairwise comparison is not equal, then that comparison determines the result.

This data type …	Is evaluated in this way …
	For more information on character comparison, see Comparison of Character Strings of Unequal Length.
DateTime	Chronologically. For information on how Time Zone affects Time comparison, see "Time Zone Sort Order" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
Interval	According to sign and magnitude.

Period	Assuming p1 and p2 are Period expressions or derived periods, the evaluation of a Period comparison predicate uses the following logic:
	IF BEGIN(p1) = BEGIN(p2) is TRUE, return END(p1) *operator* END(p2)
	ELSE return (BEGIN(p1) *operator* BEGIN(p2))
	For details on BEGIN and END, see "Period Functions and Operators" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

UDT	According to the ordering definition of the UDT. Vantage generates ordering functionality for distinct UDTs where the source types are not LOBs. To create an ordering definition for structured UDTs or distinct UDTs where the source types are LOBs, or to replace system-generated ordering functionality, use CREATE ORDERING. For more information on CREATE ORDERING, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit Type Conversion of Comparison Operands

Expression operands must be of the same data type before a comparison operation can occur.

Data Types on Which Implicit Conversion is Performed

If operand data types differ, Vantage performs an implicit conversion according to the following table. Implicit conversions are Teradata extensions to the ANSI SQL:2011 standard.

Comparisons between character and numeric data types require that the character field be convertible to a numeric value.

IF one expression operand is …	AND the other expression operand is …	THEN Vantage compares the data as …
Character	Character	Character. For more details, see Character String Comparisons.
	Date	Date. Vantage returns an error for character data with GRAPHIC server character set.

IF one expression operand is …	AND the other expression operand is …	THEN Vantage compares the data as …
	BYTEINT SMALLINT INTEGER FLOAT	FLOAT. Vantage returns an error for character data with GRAPHIC server character set.
	Period	Period.

CHAR(*k*)
VARCHAR(*k*))
WHERE *k* <= 16
	BIGINT
	DECIMAL(*m*,*n*)
		FLOAT.
		Vantage returns an error for character data with GRAPHIC server character set.

CHAR(*k*)
VARCHAR(*k*)
where *k* > 16
	DECIMAL(*m*,*n*)
	where *m* <= 16
		FLOAT.
		Vantage returns an error for character data with GRAPHIC server character set.

	BIGINT DECIMAL(*m*,*n*) where *m*> 16	Vantage returns an error.

CHAR(*k*)
VARCHAR(*k*)
	NUMBER and *k*<=16	FLOAT.
		Vantage returns an error for character data with GRAPHIC server character set.

	NUMBER and *k*> 16	Vantage returns an error.
BYTEINT	SMALLINT	SMALLINT.
BYTEINT SMALLINT	INTEGER	INTEGER.
BYTEINT SMALLINT INTEGER BIGINT	BIGINT	BIGINT.
BYTEINT	DECIMAL(*m*,*n*) where *m* <= 18 and *m-n* >= 3	DECIMAL(18,*n*).
SMALLINT	DECIMAL(*m*,*n*) where *m* <= 18 and *m-n* >= 5	
INTEGER	DECIMAL(*m*,*n*) where *m* <= 18 and *m-n* >= 10	
DATE		
BYTEINT	DECIMAL(*m*,*n*) where *m* > 18 or *m-n* < 3	DECIMAL(38,*n*).

IF one expression operand is …	AND the other expression operand is …	THEN Vantage compares the data as …
SMALLINT	DECIMAL(*m*,*n*) where *m* > 18 or *m*-*n* < 5	
INTEGER	DECIMAL(*m*,*n*) where *m* > 18 or *m*-*n* < 10	
DATE		
BIGINT	DECIMAL(*m*,*n*)	
DECIMAL(*m*,*n*)	DECIMAL(*k*,*j*) where max(*m*-*n*,*k*-*j*) + max(*j*,*n*) <= 18	DECIMAL(18,max(*j*,*n*)).
	DECIMAL(*k*,*j*) where max(*m*-*n*,*k*-*j*) + max(*j*,*n*) > 18	DECIMAL(38,max(*j*,*n*)).

BYTEINT
SMALLINT
INTEGER
BIGINT
DECIMAL(*m*,*n*)
NUMBER(*m*,*n*)
NUMBER(*m*)
NUMBER(*,*n*)
NUMBER
	NUMBER(*k*,*j*)
	NUMBER(*k*)
	NUMBER(*,*j*)
	NUMBER
		NUMBER

DATE	BYTEINT SMALLINT INTEGER	INTEGER.
	BIGINT	BIGINT.
	FLOAT	FLOAT.
FLOAT	BYTEINT SMALLINT INTEGER BIGINT DECIMAL(*m*,*n*) NUMBER(*m*,*n*) NUMBER(*m*) NUMBER(*,*n*) NUMBER	FLOAT.
Period	Character	Period.

Implicit Conversion of DateTime Types

In comparisons involving DateTime operands that differ, Vantage performs an implicit conversion according to the following table.

IF one expression operand is …	AND the other expression operand is …	THEN Vantage compares the data as …
TIMESTAMP	DATE ANSIDate dateform mode or IntegerDate dateform mode	DATE. See "Implicit TIMESTAMP-to-DATE Conversion" in *Teradata Vantage™ - Data Types and Literals*, B035-1143 .
TIMESTAMP WITH TIME ZONE		
Interval The INTERVAL type must have only one field, e.g. INTERVAL YEAR.	Exact Numeric	Numeric. See "Implicit INTERVAL-to-NUMERIC Conversion" in *Teradata Vantage™ - Data Types and Literals*, B035-1143 .

Data Types on Which Implicit Conversion is Not Performed

The following table identifies data types on which Vantage does not perform implicit type conversion.

Type	Rules
Byte	Byte data types can only be compared with byte data types. Attempts to compare a byte type with another type produces an error.
DateTime	Vantage does not perform implicit type conversion on the operands of a comparison operation involving a combination of DateTime and Interval data types. For details, see "Data Type Compatibility" in *Teradata Vantage™ - Data Types and Literals*, B035-1143 and Comparison of ANSI DateTime and Interval in USING Clause.
Interval	
TIME	Vantage does not perform implicit type conversion from TIME to TIMESTAMP and from TIMESTAMP to TIME in comparison operations.
TIMESTAMP	
UDT	Vantage does not perform implicit type conversion on UDTs for comparison operations. A UDT value can only be compared with another value of the same UDT type. To compare UDTs with other data types, you must use explicit data type conversion. For information, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143 .

Comparison of ANSI DateTime and Interval in USING Clause

External values for ANSI DateTime and Interval data are expressed as fixed length character strings in the designated client character set for the session.

When you import ANSI DateTime and Interval values with a USING phrase, you must explicitly cast them from the external character format to the proper ANSI DateTime and Interval types for comparison.

For example, consider the following statement, where the data type of the TimeField column is TIME(2):

```
USING (TimeVal CHARACTER(11), NumVal INTEGER)
UPDATE TABLE_1
SET TimeField=:TimeVal, NumField=:NumVal
WHERE CAST(:TimeVal AS TIME(2)) > TimeField;
```

Although you can use TimeVal CHAR(11) directly for assignment in this USING phrase, you must CAST the column data definition explicitly as TIME(2) in order to compare the field value TimeField in the table because TimeField is an ANSI TIME defined as TIME(2).

Proper Forms of DATE Types in Comparisons

A DATE operand must be submitted in the proper form in order to achieve a correct comparison.

Arithmetic on DATE operands causes an error if a created value is not a valid date. Therefore, although a date value can be submitted in integer form for comparison purposes, a column that contains date data should be defined as data type DATE, not INTEGER.

If an integer is used for input to DATE (this is not recommended), the way to enter the first date of the year 2000 is 1000101.

For more information, see "Teradata Date and Time Expressions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Proper forms for submitting a DATE operand are:

- An integer in the form (year-1900)*10000 + month*100 + day. The form YYMMDD is only valid for the years 1900 - 1999. For the years 2000 - 2099, the form is 1YYMMDD.
- As a character string in the same form as the date against which the compare is being done or as the date field the assignment is being done.
- A character string that is qualified with a data type phrase defining the appropriate data conversion, and a FORMAT phrase defining the format.
- As an ANSI date literal, which is always valid for a date comparison with any date format.

Examples

The following examples use a comparison operator on a value in the Employee.DOB column (defined as DATE FORMAT ' MMMbDDbYYYY ') to illustrate correct forms for a DATE operand.

Example: Entering the Operand as an Integer

In the first example, the operand is entered as an integer.

```
SELECT *
FROM Employee
WHERE DOB = 420327 ;
```

Example: Entering the Character String to Agree with the DOB column Format

In the second example, the character string is entered in a form that agrees with the format of the DOB column.

```
SELECT *
FROM Employee
WHERE DOB = 'Mar 27 1942';
```

Example: Entering the Character String with Data Type and FORMAT Phrases

In the third example, the value is entered as a character string, and so is cast with both a data type phrase (DATE) and a FORMAT phrase.

```
SELECT *
FROM Employee
WHERE DOB = CAST ('03/27/42' AS DATE FORMAT 'MM/DD/YY');
```

Example: Entering the Value as an ANSI Date Literal

In the fourth example, the value is entered as an ANSI date literal, which works regardless of the date format of the column.

```
SELECT *
FROM Employee
WHERE DOB = DATE '1942-03-27';
```

Character String Comparisons

Comparison of Character Strings of Unequal Length

If character strings of unequal length are being compared, the shorter of the two is padded on the right with pad characters before the comparison occurs.

Character Strings and Server Character Sets

When comparing character strings, data characters must have the same server character set. If they do not, then the system translates them using the implicit translation rules described in "Implicit Character-to-Character Conversion" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Effect of Collation on Character String Comparisons

Collations control character ordering. The results of character comparisons depends on the collation sequence of the character set in use.

You can set the default collation to a sequence that is compatible with the character set for your session. Use the HELP SESSION SQL statement to determine the collation setting for your current session.

The availability of diacritical or Japanese character sets, and your default collation sequence are under the control of your database administrator.

To ensure that sorting and comparison of character data are identical with the same operations performed by the client, users on a Japanese language site should set collation to CHARSET_COLL.

Case Sensitivity

All character data, except for CLOBs, accessed in the execution of a Teradata SQL statement has an attribute of CASESPECIFIC or NOT CASESPECIFIC, either by default or by explicit designation. Character string comparisons use this attribute to determine whether the comparison is case blind or case specific. Case specificity does not apply to CLOBs.

This is not an ANSI SQL:2011 compatible attribute—ANSI does all character comparisons as the equivalent of CASESPECIFIC.

The CASESPECIFIC attribute has higher precedence over the NOT CASESPECIFIC attribute:

IF …	THEN the comparison is …
either argument is CASESPECIFIC	case specific.
both arguments are NOT CASESPECIFIC	case blind.

The exception is comparisons on GRAPHIC character data, which are always CASESPECIFIC.

To apply a case specification attribute to a character string, you can:

- Use the default case specification for the session.

IF the session mode is …	THEN the default case specification is …
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. The exception is character data of type GRAPHIC, which is always CASESPECIFIC.

  Default case specification applies to all character data, including literals.
- Use the CASESPECIFIC or NOT CASESPECIFIC phrase with a character column in a CREATE TABLE or ALTER TABLE statement.

For example:

```
CREATE TABLE Students
  (StudentID INTEGER
  ,Firstname CHAR(10) CASESPECIFIC
  ,Lastname CHAR(20) NOT CASESPECIFIC);
```

Table columns carry the attribute assigned at the time the columns were defined or altered unless a CASESPECIFIC or NOT CASESPECIFIC phrase is used in their access.

- Apply the CASESPECIFIC or NOT CASESPECIFIC phrase to a character expression in the comparison.

  For example, the following statement applies the CASESPECIFIC phrase to a character literal:

  ```
  SELECT *
  FROM Students
  WHERE Firstname = 'Ike' (CASESPECIFIC);
  ```

  Use this to override the default case specification for character data, or to override the case specification attribute assigned at the time a character column was defined or altered.

For case blind comparisons, any lowercase single byte Latin letters are converted to uppercase before comparison begins. The prepared strings are compared and any trailing pad characters are ignored.

A case blind comparison always considers lowercase and uppercase Cyrillic, Greek and full-width ASCII letters to be equivalent. To distinguish lowercase and uppercase Cyrillic, Greek, and fullwidth ASCII letters you must explicitly declare CASESPECIFIC comparison.

These options work for the KANJISJIS character set as if the data were first converted to the Unicode type and then the options applied.

Using UPPER for Case Blind Comparisons

Case blind comparisons can be accomplished using the UPPER function, to make sure a character string value contains no lowercase Latin letters.

The UPPER function is not the same as declaring a value UPPERCASE.

Example: Querying for Case-Specific Names

Consider the following query:

```
SELECT *
FROM STUDENTS
WHERE Firstname = 'George';
```

The behavior of the comparison Firstname = 'George' under different case specification attributes and session modes is described in the table that follows.

IF column Firstname is ...	THEN ...
CASESPECIFIC	• If the session mode is ANSI, then ‘George’ is CASESPECIFIC and the match succeeds for rows with Firstname containing ‘George’. • If the session mode is Teradata, then “George’ is NOT CASESPECIFIC and the match succeeds for rows with Firstname containing ‘George’. When either character sting is CASESPECIFIC, the comparison is case specific.
NOT CASESPECIFIC	• If the session mode is ANSI, then ‘George’ is CASESPECIFIC and the match succeeds for rows with Firstname containing ‘George’. When either character string is CASESPECIFIC, the comparison is case specific. • If the session mode is Teradata, then ‘George’ is NOT CASESPECIFIC and the match succeeds for rows with Firstname containing any combination of cases that spell the name George, such as ‘george’ or ‘GEORGE’ or ‘George’. When both character strings are NOT CASESPECIFIC, the comparison is case blind.

Related Information

- “SET SESSION COLLATION” in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144
- *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125
- “ORDER BY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146

Comparison of KANJI1 Characters

The following sections describe how Vantage compares KANJI1 characters.

NOTICE

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see *KANJI1 Character Set* in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Equality Comparison

Comparison of character strings, which can contain mixed single byte and multibyte character data, is handled as follows:

- If *expression_1* and *expression_2* have different server character sets, then they are converted to the same type. For details, see "Implicit Character-to-Character Translation" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- If *expression_1* and *expression_2* are of different lengths, the shorter string is padded with enough pad characters to make both the same length.
- Session mode is identified:

In this mode …	The default case specification for a character string is …
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. Unless the CASESPECIFIC phrase is applied to one or both of the expressions, any simple Latin letters in both *expression_1* and *expression_2* are converted to uppercase before comparison begins.

  To override the default case specification of a character expression, apply the CASESPECIFIC or NOT CASESPECIFIC phrase.
- Case specification is determined:

IF …	THEN the comparison is …
either argument is CASESPECIFIC	case specific.
both arguments are NOT CASESPECIFIC	case blind.

- Trailing pad characters are ignored.

Nonequality Comparison

Nonequality comparisons are handled as follows:

1. If *expression_1* and *expression_2*are of different lengths, the shorter string is padded with enough pad characters to make both the same length.
2. Session mode is identified.

In this mode …	The default case specification for a character string is …
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. Unless the CASESPECIFIC qualifier is applied to one or both of the expressions, any simple Latin letters in both *expression_1* and *expression_2* are converted to uppercase before comparison begins.

   To override the default case specification of a character expression, apply the CASESPECIFIC or NOT CASESPECIFIC phrase.

3. Characters identified as single byte characters under the current character set are converted according to the collation sequence in effect for the session.
4. For the KanjiEUC character set, the ss3 0x8F character is converted to 0xFF. This means that a user-defined KanjiEUC codeset 3 is not properly ordered with respect to other KanjiEUC code sets.

   The ordering of other KanjiEUC codesets is proper; that is, ordering is the same as the binary ordering on the client system.
5. The prepared strings are compared and trailing pad characters are ignored.

Nonequality comparisons involve the collation in effect for the session. Five collations are available:

- EBCDIC
- ASCII
- MULTINATIONAL
- CHARSET_COLL
- JIS_COLL

Collation can be set at the user level with the COLLATION option of the CREATE USER or MODIFY USER statements, and at the session level with the [[.]SET] SESSION COLLATION statement or the CLIv2 CHARSET call.

If the MULTINATIONAL collation sequence is in effect, the collation sequence of a Japanese language site is determined by the collation setting installed during start-up.

Related Information

For further details on collation sequences, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Comparison Operators and the DEFAULT Function in Predicates

The DEFAULT function returns the default value of a column. It has two forms: one that specifies a column name and one that omits the column name.

Predicates using comparison operators support both forms of the DEFAULT function, but when the DEFAULT function omits the column name, the following conditions must be true:

- The comparison can only involve a single column reference.
- The DEFAULT function cannot be part of an expression.

For example, the following statement uses DEFAULT to compare the values of the Dept_No column with the default value of the Dept_No column. Because the comparison operation involves a single column reference, Vantage can derive the column context of the DEFAULT function even though the column name is omitted.

```
SELECT * FROM Employee WHERE Dept_No < DEFAULT;
```

Note that if the DEFAULT function evaluates to null, the predicate is unknown and the WHERE condition is false.

DECODE

Compares *expr* to each *search* parameter with its corresponding *result* parameter.

DECODE is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

DECODE Function Syntax

```
[TD_SYSFNLIB.] DECODE (expr, search_result [,...] default )
```

Syntax Elements

search_result

```
search, result,
```

TD_SYSFNLIB.

Name of the database where the function is located.

expr

A numeric or character expresssion.

- If *expr* is equal to one of the *search* arguments, the function returns the corresponding *result* value.
- If *expr* is not equal to any of the *search* arguments, the functions returns *default*, if the default value is specified.
- If *expr* is NULL, the function returns the result of the first search parameter that is NULL.

search

A numeric or character expresssion.

DECODE supports 1-10 *search* arguments tied to an equal number of *result* arguments.

result

A numeric or character expresssion.

DECODE supports 1-10 *result* arguments tied to an equal number of *search* arguments.

default

A numeric or character expresssion.

If *default* is not specified and there are no matches, the function returns NULL.

Argument Types and Rules

Expressions passed to this function must be one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, NUMBER, CHAR, VARCHAR

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

This function is CASESPECIFIC.

Result Type

DECODE is a scalar function whose return value data type depends on the data type associated with the *result* parameter passed into the function.

- If *result* or *default* is a numeric type, DECODE determines which argument has the highest precedence and converts the other *result* or *default* arguments to that data type and returns that data type.

  If that data type is DECIMAL/NUMERIC and the precision and scale of the *result* or *default* argument is different, the precision and scale of the return type is set to achieve the maximum precision possible.

  For example, if the *result* / *default* argument are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return type would need 3 digits to the left of the decimal point and 7 digits to the right of the decimal point to avoid any reduction in precision. In this case, the return data type is set to DECIMAL(10,7).

In cases where it is not possible to maintain the maximum precision, the data is rounded according to the DBS Control RoundHalfWayMagUp field. For example, if the *result* and *default* argument are DECIMAL(32, 8) and DECIMAL(30, 28), the return type is DECIMAL(38,14). This allows for 24 digits to the left of the decimal point required for DECIMAL(32, 8) and 14 digits to the right of the decimal point. If the DECIMAL(30,28) *result* or *default* argument is the greatest value, it is rounded to 14 places to the right of the decimal point.

If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).

- If *result* or *default* is a character data type, the function returns a VARCHAR in the character set of the first result argument.

Examples

Example: Decoding IDs

The following query:

```
SELECT DECODE(country_id, 1, 'United States',
                          2, 'England',
                          3, 'France',
                          'United States')
    FROM customers;
```

returns:

- 'United States' if the country_id is 1
- 'England' if the country_id is 2
- 'France' if the country_id is 3
- 'United States' if the country_id is not equal to 1, 2, or 3

Example: Decoding IDs Using NULL

The following query

```
SELECT DECODE(country_id, 1, 'United States',
                          2, 'England')
    FROM customers;
```

returns:

- 'United States' if the country_id is 1
- 'England' if country_id is 2
- NULL if country_id isn't in the range 1 to 2

Example: Decoding IDs When ID is Not Equal to 1, 2 or NULL

The following query:

```
SELECT DECODE(country_id, 1, 'United States',
                          2, 'England',
                          NULL, 'France')
    FROM customers;
```

returns:

- 'United States' if the country_id is 1
- 'England' if the country_id is 2
- NULL if the country_id is NULL
- NULL if the country_id is not equal to 1, 2, or NULL

Related Information

- For details about the order of precedence, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For information on the default data type format for DOUBLE PRECISION, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

GREATEST

Returns the greatest value in the list of input arguments.

GREATEST is an embedded services system function.

GREATEST Function Syntax

```
[TD_SYSFNLIB.] GREATEST
  { ( numeric_value [,...] ) |
    ( string_value [,...] ) |
    ( date_value [,...] ) |
    ( timestamp_value [,...] )
  }
```

Syntax Elements

TD_SYSFNLIB.
Name of the database where the function is located.

numeric_value
A numeric argument.

string_value
A string value.

date_value
A DATE value.

timestamp_value
A TIMESTAMP or TIMESTAMP WITH TIME ZONE value.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- CHAR or VARCHAR
- DATE
- TIMESTAMP or TIMESTAMP WITH TIME ZONE

All of the input arguments must be the same data type or else the types must be compatible.

The function accepts a maximum of 10 input arguments.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

If the arguments are character types, the string comparison uses non-padded comparison semantics. Character comparison is binary and based on the numerical codes of the characters. The string is treated as a sequence of bytes for the comparison rather than character by character.

Timestamp comparisons are made in GMT. GMT conversion occurs if a time zone is associated with the timestamp input.

This function is CASESPECIFIC.

Result Type

If the input arguments are numeric types, the function determines which argument has the highest precedence, converts the other arguments to that data type, and returns that data type.

If the data type is DECIMAL/NUMERIC and the precision and scale of the input arguments are different, the precision and scale of the return type is set to achieve the maximum precision possible. For example, if the input arguments are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return type would need 3 digits to the left of the decimal point and 7 digits to the right of the decimal point to avoid any reduction in precision. In this case, the return data type is set to DECIMAL(10,7).

In cases where it is not possible to maintain the maximum precision, the data is rounded according to the DBS Control Record RoundHalfWayMagUp field. For example, if the input arguments are DECIMAL(32, 8) and DECIMAL(30, 28), the return type is DECIMAL(38,14). This allows for 24 digits to the left of the decimal point (required for the DECIMAL(32,8) argument), and 14 digits to the right of the decimal point. If the DECIMAL(30,28) input argument is the greatest value, it is rounded to 14 places to the right of the decimal point.

If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).

If the input arguments are character types, the function converts the 2nd through 10th arguments to the data type of the 1st argument and returns the type as VARCHAR in the character set of the 1st argument.

If the input arguments are DATE types, the function returns a DATE type.

If the input arguments are TIMESTAMP types, the function returns a TIMESTAMP type. If the first parameter includes an explicit time zone, the result will also include a time zone.

If any of the input arguments is NULL, the function returns NULL.

Examples

Example: Querying for the Largest Integer Value

The following query returns the result 13.

```
SELECT GREATEST(13, 6);
```

Example: GREATEST with DECIMAL Input

In the following query, if the input arguments have data types of DECIMAL(4,2) and DECIMAL(5,4), the return data type is DECIMAL(6,4) and the result value is 13.1200.

```
    SELECT GREATEST(13.12, 6.1234);
```

Example: GREATEST with Character Input

The following query returns the result 'apples'.

```
    SELECT GREATEST('apples', 'alpha');
```

Example: Comparing Milliseconds in a TIMESTAMP

```
SELECT GREATEST('2003-08-17 12:15:24.756' (TIMESTAMP(6)), '2003-08-17
12:15:24.456' (TIMESTAMP(6)));
```

Result:

```
greatest('2003-08-17 12:15:24.756','2003-08-17 12:15:24.456')
-------------------------------------------------------------
                                   2003-08-17 12:15:24.756000
```

Related Information

- For information on activating and invoking embedded services functions, see Embedded Services System Functions.
- For information on the default data type formats, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details about the order of precedence, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

LEAST

Returns the least value in the list of input arguments.

LEAST is an embedded services system function.

LEAST Function Syntax

```
[TD_SYSFNLIB.] LEAST
  { ( numeric_value [,...] ) |
    ( string_value [,...] ) |
    ( date_value [,...] ) |
    ( timestamp_value [,...] )
  }
```

Syntax Elements

TD_SYSFNLIB.
: Name of the database where the function is located.

numeric_value
: A numeric argument.

string_value
: A string value.

date_value
: A DATE value.

timestamp_value
: A TIMESTAMP or TIMESTAMP WITH TIME ZONE value.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- CHAR or VARCHAR
- DATE
- TIMESTAMP or TIMESTAMP WITH TIME ZONE

All of the input arguments must be the same data type or else the types must be compatible.

The function accepts a maximum of 10 input arguments.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

If the arguments are character types, the string comparison uses non-padded comparison semantics. Character comparison is binary and based on the numerical codes of the characters. The string is treated as a sequence of bytes for the comparison rather than character by character.

Timestamp comparisons are made in GMT. GMT conversion occurs if a time zone is associated with the timestamp input.

This function is CASESPECIFIC.

Result Type

- If the input arguments are numeric types, the function determines which argument has the highest precedence, converts the other arguments to that data type, and returns that data type.

  If the data type is DECIMAL/NUMERIC and the precision and scale of the input arguments are different, the precision and scale of the return data types are set to achieve the maximum precision possible. For example, if the input arguments are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return data type would need 3 digits to the left of the decimal point and 7 digits to the right of the decimal point to avoid any reduction in precision. In this case, the return data type is set to DECIMAL(10,7).

  In cases where it is not possible to maintain the maximum precision, the data is rounded according to the DBS Control RoundHalfWayMagUp field. For example, if the input arguments are DECIMAL(32, 8) and DECIMAL(30, 28), the return data type is DECIMAL(38,14). This allows for 24 digits to the left of the decimal point (required for the DECIMAL(32,8) parameter), and 14 digits to the right of the decimal point. If the DECIMAL(30,28) input argument is the least value, it is rounded to 14 places to the right of the decimal point.

  If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).
- If the input arguments are character types, the function converts the 2nd through 10th arguments to the data type of the first argument and returns the type as VARCHAR in the character set of the first argument.
- If the input arguments are DATE types, the function returns a DATE type.
- If the input arguments are TIMESTAMP types, the function returns a TIMESTAMP type. If the first parameter includes an explicit time zone, the result will also include a time zone.

If any of the input arguments is NULL, the function returns NULL.

Examples

Example: Querying for the Smallest Integer Value

The following query returns 6.

```
SELECT LEAST(13, 6);
```

Example: LEAST with DECIMAL Input

In the following query, if the input arguments have data types of DECIMAL(5,4) and DECIMAL(4,2), the return data type is DECIMAL(6,4) and the return value is 1.1234.

```
SELECT LEAST(1.1234, 36.12);
```

Example: LEAST with Character Input

The following query returns 'alpha'.

```
SELECT LEAST('apples', 'alpha');
```

Example: Comparing Time Zones in a TIMESTAMP

```
SELECT LEAST(CAST('2003-08-17 20:15:24-05:00' AS TIMESTAMP WITH TIME ZONE),
CAST('2003-08-17 20:15:24-02:00' AS TIMESTAMP WITH TIME ZONE));
```

Result:

```
LEAST('2003-08-17 20:15:24-05:00','2003-08-17 20:15:24-02:00')
--------------------------------------------------------------
                              2003-08-17 20:15:24.000000-02:00
```

Related Information

- For information on activating and invoking embedded services functions, see Embedded Services System Functions.
- For details about the order of precedence, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

CASE Expressions

Specifies alternate values for a conditional expression or expressions based on equality comparisons and conditions that evaluate to TRUE.

CASE provides an efficient and powerful method for application developers to change the representation of data, permitting conversion without requiring host program intervention.

For example, you could code employee status as 1 or 2, meaning full-time or part-time, respectively. For efficiency, the system stores the numeric code but prints or displays the appropriate textual description in reports. This storage and conversion is managed by Vantage.

In addition, CASE permits applications to generate nulls based on information derived from the database, again without host program intervention. Conversely, CASE can be used to convert a null into a value.

Valued CASE Expression

Evaluates a set of expressions for equality with a test expression and returns as its result the value of the scalar expression defined for the first WHEN clause whose value equals that of the test expression. If no equality is found, then CASE returns the scalar value defined by an optional ELSE clause, or if omitted, NULL.

Default Title

The default title for a CASE expression appears as:

```
<CASE expression>
```

Valued CASE Expression Syntax

```
CASE value_expression_1 when_clause [...] [ ELSE scalar_expression_m ] END
```

Syntax Elements

value_expression_1

An expression whose value is tested for equality with *value_expression_n*.

when_clause

```
WHEN value_expression_n THEN scalar_expression_n
```

scalar_expression_m

An expression whose value is returned if evaluation falls through to the ELSE clause.

value_expression_n

A set of expressions against which the value for *value_expression_1* is tested for equality.

scalar_expression_n

An expression whose value is returned on the first equality comparison of *value_expression_1* and *value_expression_n*.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Vantage does not enforce the ANSI restriction that *value_expression_1* must be a deterministic function. In particular, Vantage allows the function RANDOM to be used in *value_expression_1*.

Note that if RANDOM is used, nondeterministic behavior may occur, depending on whether *value_expression_1* is recalculated for each comparison to *value_expression_n*.

Usage Notes

WHEN clauses are processed sequentially.

The first WHEN clause *value_expression_n* that equates to *value_expression_1* returns the value of its associated *scalar_expression_n* as its result. The evaluation process then terminates.

If no *value_expression_n* equals *value_expression_1*, then *scalar_expression_m*, the argument of the ELSE clause, is the result.

If no ELSE clause is defined, then the result defaults to NULL.

The data type of *value_expression_1* must be comparable with the data types of all of the *value_expression_n* values.

For information on the result data type of a CASE expression, see Rules for the CASE Expression Result Type.

You can use a scalar subquery in the WHEN clause, THEN clause, and ELSE clause of a CASE expression. If you use a non-scalar subquery (a subquery that returns more than one row), a runtime error is returned.

Recommendation: Do not use the built-in functions CURRENT_DATE or CURRENT_TIMESTAMP in a CASE expression that is specified in a partitioning expression for a partitioned primary index (PPI). In this case, all rows are scanned during reconciliation.

Restrictions on the Data Types in a CASE Expression

The following restrictions apply to CLOB, BLOB, and UDT types in a CASE expression:

Data Type	Restrictions
BLOB	A BLOB can only appear in *value_expression_1*, *value_expression_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in *value_expression_1*, *value_expression_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to CHAR or VARCHAR.
UDT	Multiple UDTs can appear in a CASE expression, with the following restrictions: • The data type of value_expression_1 through value_expression_n must have the same UDT data type if one of them has a UDT data type. • scalar_expression_n and scalar_expression_m must be the same UDT data type if one them has a UDT data type. Vantage does not perform implicit type conversion on UDTs in CASE expressions. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs, and then explicitly invoke the CAST function in the CASE expression. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Examples

Example: Calculating the Fraction of Cost

The following example uses a Valued CASE expression to calculate the fraction of cost in the total cost of inventory represented by parts of type ‘1’:

```
SELECT SUM(CASE part
            WHEN '1'
            THEN cost
            ELSE 0
           END
          )/SUM(cost)
FROM t;
```

Example: Using a CASE Expression

A CASE expression can be used in place of any *value-expression*.

```
SELECT *
FROM t
WHERE x = CASE y
           WHEN 2
           THEN 1001
           WHEN 5
           THEN 1002
          END;
```

Example: Combining a CASE Expression with a Concatenation Operator

The following example shows how to combine a CASE expression with a concatenation operator:

```
SELECT prodID, CASE prodSTATUS
                 WHEN 1
                 THEN 'SENT'
                 ELSE 'BACK ORDER'
                 END || ' STATUS'
FROM t1;
```

Example: Using UDT Data Types in Value Expressions

You use *value_expression_1*through *value_expression_n*to test for equality in a valued CASE expression.

For these examples, the table is defined as follows:

```
create table udtval038_t1(id integer, udt1 testcircleudt, udt2
testrectangleudt) PRIMARY INDEX (id);
```

The following example shows a valued CASE expression, where all value expressions are of the same UDT data type:

```
SELECT CASE udt1
                 WHEN new testcircleudt('1,1,2,yellow,circ')
                 THEN 'Row 1'
                 WHEN new testcircleudt('2,2,4,purple,circ')
                 THEN 'Row 2'
                 WHEN new testcircleudt('3,3,9,green,circ')
                 THEN 'Row 3'
                 ELSE 'Row is NULL'
                 END
```

```
   FROM t1;
*** Query completed. 4 rows found. One column returned.
<CASE  expression>
------------------
Row 3
Row 1
Row is NULL
Row 2
```

However, the following example does not complete successfully because *testrectangleudt* does not match the other UDT data types:

```
   SELECT CASE udt1
                    WHEN new testcircleudt('1,1,2,yellow,circ')
                    THEN 'Row 1'
                    WHEN new testrectangleudt('2,2,4,4,purple,rect')
                    THEN 'Row 2'
                    WHEN new testcircleudt('3,3,9,green,circ')
                    THEN 'Row 3'
                    ELSE 'Row is NULL'
                    END
   FROM t1;
```

Example 1: Using UDT Data Types in Scalar Expressions

You use *scalar_expression_n*and *scalar_expression_m*as the expressions to return on when the equality comparison on a valued or searched CASE expression evaluates to TRUE, or the value to return on in an ELSE condition.

For these examples, the table is defined as follows:

```
create table udtval038_t1(id integer, udt1 testcircleudt, udt2
testrectangleudt) PRIMARY INDEX (id);
```

Following is an example of a searched CASE Expression where all scalar expressions are of the same UDT data type.

Note:

The *search_condition_n*can be a different UDT data type than the *scalar_expression_n*. SELECT * FROM udtval038_t1

```
        WHERE udt1 = CASE
        WHEN udt2 <> new testrectangleudt('2,2,4,4,pink,rect')
```

```
          THEN new testcircleudt('1,1,2,blue,circ')
          ELSE new testcircleudt('2,2,4,purple,circ')
*** Query completed. 2 rows found. 3 columns returned.
          END;
id udt1
----------- -----------------------------------------------
          1 1, 1, 2, yellow, circ
          2 2, 2, 4, purple, circ
```

However, the following example does not complete successfully because the scalar expressions are of different data types.

```
   SELECT * FROM udtval038_t1
          WHERE udt1 = CASE
          WHEN udt2 <> new testrectangleudt('2,2,4,4,pink,rect')
          THEN new testcircleudt('1,1,2,blue,circ')
          ELSE new testrectangleudt('2,2,4,4,purple,rect')
          END;
```

Related Information

- For information about error conditions, see Error Conditions.
- For information about the result data type of a CASE expression, see Rules for the CASE Expression Result Type.
- For information about format of the result of a CASE expression, see Default Format.
- For information about nulls and CASE expressions, see CASE and Nulls.

Searched CASE Expression

Evaluates a search condition and returns one of a WHEN clause-defined set of scalar values when it finds a value that evaluates to TRUE. If no TRUE test is found, then CASE returns the scalar value defined by an ELSE clause, or if omitted, NULL.

Default Title

The default title for a CASE expression appears as:

```
<CASE expression>
```

Searched CASE Expression Syntax

```
CASE when_clause [,...] [ ELSE scalar_expression_m ] END
```

Syntax Elements

when_clause

```
WHEN search_condition_n THEN scalar_expression_n
```

scalar_expression_m
A scalar expression whose value is returned when no *search_condition_n* evaluates to TRUE.

search_condition_n
A predicate condition to be tested for truth.

scalar_expression_n
A scalar expression whose value is returned when *search_condition_n* is the first search condition that evaluates to TRUE.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Usage Notes

WHEN clauses are processed sequentially.

The first WHEN clause *search_condition_n* that is TRUE returns the value of its associated *scalar_expression_n* as its result. The evaluation process then ends.

If no *search_condition_n* is TRUE, then *scalar_expression_m*, the argument of the ELSE clause, is the result.

If no ELSE clause is defined, then the default value for the result is NULL.

You can use a scalar subquery in the WHEN clause, THEN clause, and ELSE clause of a CASE expression. If you use a non-scalar subquery (a subquery that returns more than one row), a runtime error is returned.

Recommendation: Do not use the built-in functions CURRENT_DATE or CURRENT_TIMESTAMP in a CASE expression that is specified in a partitioning expression for a partitioned primary index (PPI). In this case, all rows are scanned during reconciliation.

Rules for WHEN Search Conditions

WHEN search conditions have the following properties:

- Can take the form of any comparison operator, such as LIKE, =, or <>.
- Can be a quantified predicate, such as ALL or ANY.
- Can contain a scalar subquery.
- Can contain joins of two tables.

  For example:

  ```
  SELECT CASE
  WHEN t1.x=t2.x THEN t1.y
  ELSE t2.y
  END FROM t1,t2;
  ```
- Cannot contain SELECT statements.

Restrictions on the Data Types in a CASE Expression

The following restrictions apply to CLOB, BLOB, and UDT types in a CASE expression:

Data Type	Restrictions
BLOB	A BLOB can only appear in *value_expression_1*, *value_expression_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in *value_expression_1*, *value_expression_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to CHAR or VARCHAR.
UDT	Multiple UDTs can appear in a CASE expression, with the following restrictions: • The data type of value_expression_1 through value_expression_n must have the same UDT data type if one of them has a UDT data type. • scalar_expression_n and scalar_expression_m must be the same UDT data type if one them has a UDT data type. Vantage does not perform implicit type conversion on UDTs in CASE expressions. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs, and then explicitly invoke the CAST function in the CASE expression. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Examples

Example: Evaluating a Search Condition

The following statement is equivalent to the first example of the valued form of CASE on “Example”:

```
SELECT SUM(CASE
            WHEN part='1'
```

```
              THEN cost
              ELSE 0
             END
            ) / SUM(cost)
    FROM t;
```

Example: Using a CASE Expression

CASE expressions can be used in place of any *value-expression*s.

Note that the following example does not specify an ELSE clause. ELSE clauses are always optional in a CASE expression. If an ELSE clause is not specified and none of the WHEN conditions are TRUE, then a null is returned.

```
SELECT *
FROM t
WHERE x = CASE
           WHEN y=2
           THEN 1
           WHEN (z=3 AND y=5)
           THEN 2
          END;
```

Example: Using an ELSE Clause

The following example uses an ELSE clause.

```
SELECT *
FROM t
WHERE x = CASE
           WHEN y=2
           THEN 1
           ELSE 2
          END;
```

Example: Using a CASE expression to Enhance Performance

The following example shows how using a CASE expression can result in significantly enhanced performance by eliminating multiple passes over the data. Without using CASE, you would have to perform multiple queries for each region and then consolidate the answers to the individual queries in a final report.

```
SELECT SalesMonth, SUM(CASE
                        WHEN Region='NE'
                        THEN Revenue
                        ELSE 0
                       END),
                   SUM(CASE
                        WHEN Region='NW'
                        THEN Revenue
                        ELSE 0
                       END),
                   SUM(CASE
                        WHEN Region LIKE 'N%'
                        THEN Revenue
                        ELSE 0
                       END)
AS NorthernExposure, NorthernExposure/SUM(Revenue),
SUM(Revenue)
FROM Sales
GROUP BY SalesMonth;
```

Example: Producing a Report to Show Employee Salary

All employees whose salary is less than $40000 are eligible for an across the board pay increase.

IF your salary is less than …	AND you have greater than this many years of service …	THEN you receive this percentage salary increase …
$30000.00	8	15
$35000.00	10	10
$40000.00		5

The following SELECT statement uses a CASE expression to produce a report showing all employees making under $40000, displaying the first 15 characters of the last name, the salary amount (formatted with $and punctuation), the number of years of service based on the current date (in the column named On_The_Job) and which of the four categories they qualify for: '15% Increase', '10% Increase', '05% Increase' or 'Not Qualified'.

```
SELECT CAST(last_name AS CHARACTER(15))
   ,salary_amount (FORMAT '$,$$9,999.99')
   ,(date - hire_date)/365.25 (FORMAT 'Z9.99') AS On_The_Job
   ,CASE
       WHEN salary_amount < 30000 AND On_The_Job > 8
```

```
           THEN '15% Increase'
           WHEN salary_amount < 35000 AND On_The_Job > 10
           THEN '10% Increase'
           WHEN salary_amount < 40000 AND On_The_Job > 10
           THEN '05% Increase'
           ELSE 'Not Qualified'
          END  AS Plan
    WHERE salary_amount < 40000
    FROM employee
    ORDER BY 4;
```

The result of this query appears in the following table:

last_name	salary_amount	On_The_Job	Plan
Trader	$37,850.00	20.61	05% Increase
Charles	$39,500.00	18.44	05% Increase
Johnson	$36,300.00	20.41	05% Increase
Hopkins	$37,900.00	19.99	05% Increase
Morrissey	$38,750.00	18.44	05% Increase
Ryan	$31,200.00	20.41	10% Increase
Machado	$32,300.00	18.03	10% Increase
Short	$34,700.00	17.86	10% Increase
Lombardo	$31,000.00	20.11	10% Increase
Phillips	$24,500.00	19.95	15% Increase
Rabbit	$26,500.00	18.03	15% Increase
Kanieski	$29,250.00	20.11	15% Increase
Hoover	$25,525.00	20.73	15% Increase
Crane	$24,500.00	19.15	15% Increase
Stein	$29,450.00	20.41	15% Increase

Related Information

- For information about error conditions, see Error Conditions.
- For information about the result data type of a CASE expression, see Rules for the CASE Expression Result Type.
- For information about format of the result of a CASE expression, see Default Format.

- For information about nulls and CASE expressions, see CASE and Nulls.

Error Conditions

The following conditions or expressions are considered illegal in a CASE expression:

Condition or Expression	Example

A condition after the keyword CASE is supplied.	`SELECT CASE a=1`
	`        WHEN 1`
	`        THEN 1`
	`        ELSE 0`
	`        END`
	`FROM t;`

A non valid WHEN expression is supplied in a valued CASE expression.	`SELECT CASE a`
	`        WHEN a=1`
	`        THEN 1`
	`        ELSE 0`
	`       END`
	`FROM t;`

A non valid WHEN condition is supplied in a searched CASE expression.	`SELECT CASE`
	`        WHEN a`
	`        THEN 1`
	`        ELSE 0`
	`       END`
	`FROM t;`
	`SELECT CASE`
	`        WHEN NULL`
	`        THEN 'NULL'`
	`       END`
	`FROM table_1;`

A non-scalar subquery is specified in a WHEN condition of a searched CASE expression.	`SELECT CASE`
	`        WHEN t.a IN`
	`         (SELECT u.a`
	`          FROM u)`
	`        THEN 1`
	`        ELSE 0`
	`       END`
	`FROM t;`

A CASE expression references multiple UDTs that are not identical to each other.	`SELECT CASE t.shape.gettype()`
	`        WHEN 1`
	`        THEN NEW circle('18,18,324')`
	`        WHEN 2`
	`        THEN NEW square('20,20,400')`
	`       END;`

Rules for the CASE Expression Result Type

Because the expressions in CASE THEN/ELSE clauses can be different data types, determining the result type is not always straightforward. You can use the TYPE attribute function with the CASE expression as the argument to find out the result data type. See TYPE.

The following rules apply to the data type of the CASE expression result.

THEN/ELSE Expressions Having the Same Non-Character Data Type

If all of the THEN and ELSE expressions have the same non-character data type, the result of the CASE expression is that type. For example, if all of the THEN and ELSE expressions have an INTEGER type, the result type of the CASE expression is INTEGER.

For information about how the precision and scale of DECIMAL results are calculated, see Binary Arithmetic Result Data Types.

THEN/ELSE Character Type Expressions

The following rules apply to CASE expressions where the data types of all of the THEN/ELSE expressions are character:

- The result of the CASE expression is also a character data type, with the length equal to the maximum length of the different character data types of the THEN/ELSE expressions.
- If the data types of all of the THEN/ELSE expressions are CHARACTER (or CHAR), the result data type is CHARACTER. If one or more expressions are VARCHAR (or LONG VARCHAR), the result data type is VARCHAR.
- The server character set of the result is determined as follows:
    - If the CASE expression contains 1 nonliteral character expression and 1 or more literals, then Vantage tries to translate every literal to the character set of the nonliteral. If the translations are successful, then the character set of the nonliteral is used for the result data type. If the translations are not successful, the server character set of the result is Unicode.
    - If the CASE expression contains more than 1 nonliteral character expression and 1 or more literals, then:

      If all of the nonliteral expressions have the same character set, then Vantage uses this character set as the common data type. Otherwise, if the nonliteral expressions have differing character sets, then Vantage uses the Unicode character set as the common data type.

      Vantage tries to translate every literal to the character set of the common data type. If the translations are successful, then the result data type has the character set of the common data type. If the translations are not successful, the server character set of the result is Unicode.

Examples

Examples of Character Data in a CASE Expression

For the following examples of CHARACTER data behavior, assume the default server character set is KANJI1 and the table definition for the CASE examples is as follow:

```
CREATE TABLE table_1
(
 i        INTEGER,
 column_l CHARACTER(10) CHARACTER SET LATIN,
 column_u CHARACTER(10) CHARACTER SET UNICODE,
 column_j CHARACTER(10) CHARACTER SET KANJISJIS,
 column_g CHARACTER(10) CHARACTER SET GRAPHIC,
 column_k CHARACTER(10) CHARACTER SET KANJI1
);
```

Note:

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see "KANJI1 Character Set" in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Examples of Character Data in a CASE Expression: Example 1

The server character set of the result of the following query is UNICODE because the CASE expression contains more than 1 nonliteral character expressions with differing character sets.

```
SELECT i, CASE
            WHEN i=2 THEN column_u
            WHEN i=3 THEN column_j
            WHEN i=4 THEN column_g
            WHEN i=5 THEN column_k
            ELSE column_l
          END
FROM table_1
ORDER BY 1;
```

In the following query, the CASE expression returns a VARCHAR result because the THEN and ELSE clause contains FLOAT and VARCHAR values. The length of the result is 30 since the default format for FLOAT is a string less than 30 characters, and USER is defined as VARCHAR(30) CHARACTER SET UNICODE. The result is CHARACTER SET UNICODE because USER is UNICODE.

```
SELECT a, CASE
             WHEN a=1
             THEN TIME
             ELSE USER
            END
FROM table_1
ORDER BY 1;
```

Examples of Character Data in a CASE Expression: Example 2

The result of the following query is a 5354 failure (Arguments must be of type KANJI1) because one THEN/ELSE expression is a KANJI1 literal, but the server character sets of all the other THEN/ELSE expressions are not KANJI1.

```
SELECT i, CASE
            WHEN i=1 THEN column_l
            WHEN i=2 THEN column_u
            WHEN i=3 THEN column_j
            WHEN i=4 THEN column_g
            WHEN i=5 THEN _Kanji1'4142'XC
            ELSE column_k
          END
FROM table_1
ORDER BY 1;
```

For this example, assume the following table definition:

```
CREATE table_1
  (i        INTEGER,
   column_l CHARACTER(10) CHARACTER SET LATIN,
   column_u CHARACTER(10) CHARACTER SET UNICODE,
   column_j CHARACTER(10) CHARACTER SET KANJISJIS,
   column_g CHARACTER(10) CHARACTER SET GRAPHIC,
   column_k CHARACTER(10) CHARACTER SET KANJI1);
```

The following query fails because the server character set is GRAPHIC (because the server character set of the first THEN with a character type is GRAPHIC):

```
SELECT i, CASE
            WHEN i=1 THEN 4
            WHEN i=2 THEN column_g
            WHEN i=3 THEN 5
            WHEN i=4 THEN column_l
            WHEN i=5 THEN column_k
            ELSE 10
          END
FROM table_1
ORDER BY 1;
```

Examples of Character Data in a CASE Expression: Example 3

One THEN/ELSE expression in the following query has a Unicode column. The query is successful and the result data type is UNICODE because the CASE expression contains 1 Unicode column and all other literals can be successfully translated to Unicode.

```
SELECT i, CASE
            WHEN i=1 THEN column_u
            WHEN i=2 THEN 'abc'
            WHEN i=3 THEN 8
            WHEN i=4 THEN _KanjiSJIS'4142'XC
            ELSE 10
          END
FROM table_1
ORDER BY 1;
```

Examples of Character Data in a CASE Expression: Example 4

One THEN/ELSE expression in the following query has a Latin column. The query is successful and the result data type is Latin because the other literals can be successfully translated to Latin.

```
SELECT i, CASE
            WHEN i=1 THEN 'abc'
            WHEN i=2 THEN column_l
            ELSE 'def'
          END
FROM table_1
ORDER BY 1;
```

THEN/ELSE Expressions Having Mixed Data Types

The rules for mixed data appear in the following table.

IF the THEN / ELSE clause expressions …	THEN …
consist of BYTE and/or VARBYTE data types	if the data types of all of the THEN/ELSE expressions are BYTE, the result data type is BYTE. If one or more expressions are VARBYTE, the result data type is VARBYTE. The result has a length equal to the maximum length of the different byte data types.
contain a DateTime or Interval data type	all of the THEN/ELSE clause expressions must have the same data type.
contain a FLOAT (approximate numeric) and no character strings	the CASE expression returns a FLOAT result. **Note:** Some inaccuracy is inherent and unavoidable when FLOAT data types are involved.
are composed only of DECIMAL data	the CASE expression returns a DECIMAL result. **Note:** A DECIMAL arithmetic result can have up to 38 digits. A result larger than 38 digits produces a numeric overflow error. For information about how the precision and scale of DECIMAL results are calculated, see Binary Arithmetic Result Data Types.
are composed only of mixed DECIMAL, BYTEINT, SMALLINT, INTEGER, and BIGINT data	all are implicitly converted to FLOAT and the CASE expression returns a FLOAT result. **Note:** Some inaccuracy is inherent and unavoidable when FLOAT data types are involved. Implicit conversion of DECIMAL and INTEGER values to FLOAT values may result in a loss of precision or produce a number that cannot be represented exactly.
are a mix of BYTEINT, SMALLINT, INTEGER, and BIGINT data	the resulting type is the largest type of any of the THEN/ELSE clause expressions, where the following list orders the types from largest to smallest: • BIGINT • INTEGER • SMALLINT • BYTEINT
are composed only of numeric and character data	the numeric data is converted to CHARACTER with a length as determined by the format associated with the numeric expression. Then, the rules for the result data type for character, length, and character set are applied. For details, see THEN/ELSE Character Type Expressions.

IF the THEN / ELSE clause expressions …	THEN …
	Note: An error is generated if the server character set is GRAPHIC.

Examples of Numeric Data in a CASE Expression

For the following examples of numeric data behavior, assume the following table definitions for the CASE examples:

```
CREATE TABLE dec22
   (column_1 INTEGER
   ,column_2 INTEGER
   ,column_3 DECIMAL(22,2) );
```

Example: CASE Expression Fails

In the following statement, the CASE expression fails when column_2 contains the value 1 and column_3 contains the value 11223344556677889900.12 because the result is a DECIMAL value that requires more than 38 digits of precision:

```
SELECT SUM (CASE
             WHEN column_2=1
             THEN column_3 * 6.112233445566778800000
             ELSE column_3
            END )
FROM dec22;
```

Example: Shortening the Scale of the Multiplier

The following query corrects the problem in Example: CASE Expression Fails by shortening the scale of the multiplier in the THEN expression:

```
SELECT SUM (CASE
             WHEN column_2=1
             THEN column_3 * 6.1122334455667788
             ELSE column_3
            END )
FROM dec22;
```

Example: Returning a DECIMAL(38,2) Result

In the following query, the CASE expression returns a DECIMAL(38,2) result because the THEN and ELSE clauses contain DECIMAL values:

```
SELECT SUM (CASE
             WHEN column_2=1
             THEN column_3 * 6
             ELSE column_3
            END )
FROM dec22;
```

Examples of Character and Numeric Data in a CASE Expression

The following examples illustrate the behavior of queries containing CASE expressions with a THEN/ ELSE clause composed of numeric and character data.

Examples of Character and Numeric Data in a CASE Expression: Example 1

In the following query, the CASE expression returns a VARCHAR result because the THEN and ELSE clause contains FLOAT and VARCHAR values. The length of the result is 30 since the default format for FLOAT is a string less than 30 characters, and USER is defined as VARCHAR(30) CHARACTER SET UNICODE. The result is CHARACTER SET UNICODE because USER is UNICODE.

```
SELECT a, CASE
             WHEN a=1
             THEN TIME
             ELSE USER
            END
FROM table_1
ORDER BY 1;
```

Examples of Character and Numeric Data in a CASE Expression: Example 2

For this example, assume the following table definition:

```
CREATE table_1
  (i        INTEGER,
   column_l CHARACTER(10) CHARACTER SET LATIN,
   column_u CHARACTER(10) CHARACTER SET UNICODE,
   column_j CHARACTER(10) CHARACTER SET KANJISJIS,
   column_g CHARACTER(10) CHARACTER SET GRAPHIC,
   column_k CHARACTER(10) CHARACTER SET KANJI1);
```

The following query fails because the server character set is GRAPHIC (because the server character set of the first THEN with a character type is GRAPHIC):

```
SELECT i, CASE
            WHEN i=1 THEN 4
            WHEN i=2 THEN column_g
            WHEN i=3 THEN 5
            WHEN i=4 THEN column_l
            WHEN i=5 THEN column_k
            ELSE 10
          END
FROM table_1
ORDER BY 1;
```

Related Information

- Binary Arithmetic Result Data Types

Format for a CASE Expression

Default Format

The result of a CASE expression is displayed using the default format for the resulting data type. The result of a CASE expression does not apply the explicit format that may be defined for a column appearing in a THEN/ELSE expression.

Consider the following table definition:

```
CREATE TABLE duration
   (i INTEGER
   ,start_date DATE FORMAT 'EEEEBMMMBDD,BYYYY'
   ,end_date DATE FORMAT 'DDBM3BY4' );
```

Assume the default format for the DATE data type is 'YY/MM/DD'.

The following query displays the result of the CASE expression using the 'YY/MM/DD' default DATE format, not the format defined for the *start_date* or *end_date* columns:

```
SELECT i, CASE
           WHEN i=1
           THEN start_date
           WHEN i=2
           THEN end_date
          END
FROM duration
ORDER BY 1;
```

Using Explicit Type Conversion to Change Format

To modify the format of the result of a CASE expression, use CAST and specify the FORMAT clause.

Here is an example that uses CAST to change the format of the result of the CASE expression in the previous query.

```
SELECT i, ( CAST ((CASE
             WHEN i=1
             THEN start_date
             WHEN i=2
             THEN end_date
            END) AS DATE FORMAT 'M4BDD,BYYYY'))
FROM duration
ORDER BY 1;
```

For information on the default data type formats and the FORMAT phrase, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

CASE and Nulls

The ANSI SQL:2011 standard specifies that the CASE expression and its related expressions COALESCE and NULLIF must be capable of returning a null result.

Nulls and CASE Expressions

The rules for null usage in CASE, NULLIF, and COALESCE expressions are as follows.

- If no ELSE clause is specified in a CASE expression and the evaluation falls through all the WHEN clauses, the result is null.
- Nulls and expressions containing nulls are valid as *value_expression_1* in a valued CASE expression.

  The following examples are valid.

  ```
  SELECT CASE NULL
          WHEN 10
          THEN 'TEN'
  ```

```
       END;

SELECT CASE NULL + 1
        WHEN 10
        THEN 'TEN'
       END;
```

Both of the preceding examples return NULL because no ELSE clause is specified, and the evaluation falls through the WHEN clause because NULL is not equal to any value or to NULL.

- Comparing NULL to any value or to NULL is always FALSE. When testing for NULL, it is best to use a searched CASE expression using IS NULL or IS NOT NULL in the WHEN condition.

  The following example is valid.

```
SELECT CASE
        WHEN column_1 IS NULL
        THEN 'NULL'
       END
FROM table_1;
```

Often, Vantage can detect when an expression that always evaluates to NULL is compared to some other expression or NULL, and gives an error that recommends using IS NULL or IS NOT NULL instead. Note that ANSI SQL does not consider this to be an error; however, Vantage reports an error since it is unlikely that comparing NULL in this manner is the intent of the user.

The following examples are not legal.

```
SELECT CASE column_1
        WHEN NULL
        THEN 'NULL'
       END
FROM table_1;

SELECT CASE column_1
        WHEN NULL + 1
        THEN 'NULL'
       END
FROM table_1;
SELECT CASE
        WHEN column_1 = NULL
        THEN 'NULL'
       END
FROM table_1;
SELECT CASE
```

```
            WHEN column_1 = NULL + 1
            THEN 'NULL'
           END
    FROM table_1;
```

- Nulls and expressions containing nulls are valid as THEN clause expressions.

  The following example is valid.

  ```
    SELECT CASE
            WHEN column_1 = 10
            THEN NULL
           END
    FROM table_1
  ```

  Note that, unlike the previous examples, the NULL in the THEN clause is an SQL keyword and not the value of a character literal.

CASE Shorthands

ANSI also defines two shorthand special cases of CASE specifically for handling nulls.

- COALESCE expression (see COALESCE Expression)
- NULLIF expression (see NULLIF Expression)

COALESCE Expression

Returns NULL if all its arguments evaluate to null. Otherwise, it returns the value of the first non-null argument in the *scalar_expression* list.

COALESCE is a shorthand expression for the following full CASE expression:

```
CASE
 WHEN   scalar_expression_1  IS NOT NULL
 THEN   scalar_expression_1
 ...
 WHEN   scalar_expression_n  IS NOT NULL
 THEN   scalar_expression_n
 ELSE NULL
END
```

Default Title

The default title for a COALESCE expression appears as:

```
<CASE expression>
```

COALESCE Expression Syntax

```
COALESCE ( scalar_expression_1, scalar_expression_n [,...] )
```

Syntax Elements

scalar_expression_1

A scalar expression.

scalar_expression_n

A scalar expression.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Usage Notes

A *scalar_expression_n* in the argument list may be evaluated twice: once as a search condition and again as a return value for that search condition.

Using a nondeterministic function, such as RANDOM, in a *scalar_expression_n* may have unexpected results, because if the first calculation of *scalar_expression_n* is not NULL, the second calculation of that *scalar_expression_n*, which is returned as the value of the COALESCE expression, might be NULL.

You can use a scalar subquery in a COALESCE expression. However, if you use a non-scalar subquery (a subquery that returns more than one row), a runtime error is returned.

Restrictions on the Data Types in a COALESCE Expression

The following restrictions apply to CLOB, BLOB, and UDT types in a COALESCE expression.

Data Type	Restrictions
BLOB	A BLOB can only appear in the argument list when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in the argument list when it is cast to CHAR or VARCHAR.
UDT	Multiple UDTs can appear in the argument list only when they are identical types because Vantage does not perform implicit type conversion on UDTs in a COALESCE expression.

Examples

Example: Querying for a Phone Number

The following example returns the home phone number of the named individual (if present), or office phone if HomePhone is null, or MessageService if present and both home and office phone values are null. Returns NULL if all three values are null.

```
SELECT Name, COALESCE (HomePhone, OfficePhone, MessageService)
FROM PhoneDir;
```

Example: Using COALESCE with an Arithmetic Operator

The following example uses COALESCE with an arithmetic operator.

```
SELECT COALESCE(Boxes,0) * 100
FROM Shipments;
```

Example: Using COALESCE with an Comparison Operator

The following example uses COALESCE with a comparison operator.

```
SELECT Name
FROM Directory
WHERE Organization <> COALESCE (Level1, Level2, Level3);
```

Related Information

- For additional information, such as the rules for evaluation and result data type, see CASE Expressions.

NULLIF Expression

Returns NULL if its arguments are equal. Otherwise, it returns its first argument, *scalar_expression_1*.

NULLIF is a shorthand expression for the following full CASE expression:

```
CASE
 WHEN  scalar_expression_1=scalar_expression_2
 THEN NULL
```

```
    ELSE  scalar_expression_1
   END
```

Default Title

The default title for a NULLIF expression appears as:

```
<CASE expression>
```

NULLIF Expression Syntax

```
NULLIF ( scalar_expression_1, scalar_expression_2 )
```

Syntax Elements

scalar_expression_1

The scalar expression to the left of the = in the expanded CASE expression:

```
CASE
  WHEN scalar_expression_1 = scalar_expression_2
  THEN NULL
  ELSE scalar_expression_1
END
```

scalar_expression_2

The scalar expression to the right of the = in the expanded CASE expression:

```
CASE
  WHEN scalar_expression_1 = scalar_expression_2
  THEN NULL
  ELSE scalar_expression_1
END
```

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Usage Notes

The *scalar_expression_1* argument may be evaluated twice: once as part of the search condition (see the preceding expanded CASE expression) and again as a return value for the ELSE clause.

Using a nondeterministic function, such as RANDOM, may have unexpected results if the first calculation of *scalar_expression_1* is not equal to *scalar_expression_2*, in which case the result of the CASE expression is the value of the second calculation of *scalar_expression_1*, which may be equal to *scalar_expression_2*.

You can use a scalar subquery in a NULLIF expression. However, if you use a non-scalar subquery (a subquery that returns more than one row), a runtime error is returned.

Restrictions on the Data Types in a NULLIF Expression

The following restrictions apply to CLOB, BLOB, and UDT types in a NULLIF expression.

Data Type	Restrictions
BLOB	A BLOB can only appear in the argument list when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in the argument list when it is cast to CHAR or VARCHAR.
UDT	Multiple UDTs can appear in the argument list only when they are identical types and have an ordering definition.

Examples

The following examples show queries on the following table:

```
CREATE TABLE Membership
   (FullName CHARACTER(39)
   ,Age SMALLINT
   ,Code CHARACTER(4) );
```

Example: Querying with the ANSI-Compliant Form

Here is the ANSI-compliant form of the Teradata SQL NULLIFZERO(Age) function, and is more versatile.

```
SELECT FullName, NULLIF (Age,0) FROM Membership;
```

Example: Blank Spaces

In the following query, blanks indicate no value.

```
SELECT FullName, NULLIF (Code, '    ') FROM Membership;
```

Example: Querying for NULLIF in an Expression with an Arithmetic Operator

The following example uses NULLIF in an expression with an arithmetic operator.

```
SELECT NULLIF(Age,0) * 100;
```

Related Information

- For additional information, such as the rules for evaluation and result data type, see CASE and Nulls.

Hash-Related Functions

Hash-related functions return information about the:

- Primary or fallback AMP that corresponds to a given hash bucket number
- Hash bucket number that corresponds to a given row hash value
- Row hash value for the primary index of a row
- Highest AMP number
- Highest hash bucket number
- Maximum value that can be generated by applying the hash function to an unsigned integer

Use the hash-related functions to identify the statistical properties of the current primary index or secondary index, or to evaluate these properties for other columns to determine their suitability as a future primary index or secondary index. The statistics can help you to minimize hash synonyms and enhance the uniformity of data distribution.

HASHAMP

Finds the primary AMP corresponding to the hash bucket number specified in the expression and returns the AMP ID. If no hash bucket number is specified, HASHAMP returns one less than the maximum number of AMPs in the system.

HASHAMP Function Syntax

```
HASHAMP ( [ hash_bucket_number_expr ] )
```

Syntax Elements

hash_bucket_number_expr

```
{ expression [ MAP = sparsemap_name COLOCATE USING = colocation_name
] |
  [ expression ] MAP = contiguousmap_name
}
```

expression

An expression that evaluates to a valid hash bucket number.

MAP

An object that specifies which AMPs store the rows of a table.

sparsemap_name

The name of the sparse map, the map that includes a subset of AMPs from a contiguous map.

COLOCATE USING

A clause that forces tables that use the same sparse map to be stored on the same subset of AMPs.

COLOCATE USING is required with a sparse map. It cannot be used with a contiguous map.

colocation_name

The colocation name, usually *databasename_tablename*.

contiguousmap_name

The name of the contiguous map, the map that includes all AMPs within a specified range.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

About Default Arguments

Arguments	Result
None	`HASHAMP ()` returns an INTEGER that is one less than the maximum number of AMPs in the system.
expression is not specified	For a contiguous map: The function returns an INTEGER representing the highest AMP number in the specified or default contiguous map. For a contiguous map starting at AMP zero, adding one to the result gives the total number of AMPs in the contiguous map. For a sparse map: *Expression* must be specified for a sparse map.
expression is specified	The function returns the ID of the primary AMP corresponding to the hash bucket number specified in *expression*, based on the specified or default map.

About the Default Map

To determine the default map, the system evaluates the following, in order, and uses the first map found:

1. User profile.
2. User.
3. System.

About *Expression*

The *expression* argument must evaluate to an INTEGER data type where the range of valid values depends on the system hash bucket size.

IF the hash bucket size is …	THEN the range of values for *expression* is …
16 bits	0 to 65,535.
20 bits	0 to 1,048,575.

For information on how to set the system hash bucket size, see “DBS Control utility” in *Teradata Vantage™ - Database Utilities*, B035-1102.

If *expression* cannot be implicitly converted to an INTEGER, an error is reported.

If *expression* results in a UDT, Vantage performs implicit type conversion on the UDT, if the UDT has an implicit cast that casts between the UDT and any of the following predefined types:

- Numeric
- Character
- DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including HASHAMP, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

Examples

Example Assumptions

The following two examples assume a table T with columns column_1, column_2, and an INTEGER column B populated with integer numbers from zero to the maximum number of hash buckets on the system. This table is created as follows:

```
CREATE TABLE T
   (column_1 INTEGER
   ,column_2 INTEGER
   ,B INTEGER)
UNIQUE PRIMARY INDEX (column_1, column_2);
```

Example: Querying the Distribution of Hash Buckets

The following query returns the distribution of the hash buckets among the primary AMPs.

```
    SELECT B, HASHAMP (B)
    FROM T
    ORDER BY 1;
```

Example: Querying the Number of Rows on Each Primary AMP

The following query returns the number of rows on each primary AMP where column_1 and column_2 are the primary index of table T.

```
    SELECT HASHAMP (HASHBUCKET (HASHROW (column_1,column_2))), COUNT (*)
    FROM T
    GROUP BY 1
    ORDER BY 1;
```

Example: HASHAMP with a Contiguous Map

The following query returns the ID of the highest AMP number in the specified contiguous map.

```
Select HashAmp (map = contiguousmap_name);
```

Example: HASHAMP with an Expression and a Contiguous Map

The following query returns the ID of the primary AMP corresponding to the specified hash bucket number:

```
Select HashAmp ( HashBucket(HashRow(column1, column2)) map = contiguousmap_name
) from t1;
```

Example: Which AMPs Contain the Rows of a Table

The following query shows which AMPs contain the rows of a table.

```
sel col1, hashamp(hashbucket(hashrow(col1))
          map = map1count3 colocate using u1_tabm1c3)
          (named "which amp?")
from u1.tabm1c3 order by 1;
```

```
       col1   which amp?
-----------  -----------
          0            3
          1            2
          2            3
          3            2
          4            2
          5            2
          6            3
          7            2
          8            3
          9            1
```

Related Information

For more information on implicit type conversion, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

HASHBAKAMP

Finds the fallback AMP corresponding to the hash bucket number specified in the expression and returns the AMP ID. If no hash bucket is specified, HASHBAKAMP returns one less than the maximum number of fallback AMPs in the system.

HASHBAKAMP Function Syntax

```
HASHBAKAMP ( [ hash_bucket_number_expr ] )
```

Syntax Elements

hash_bucket_number_expr

```
{ expression [ MAP = sparsemap_name COALESCE USING = colocation_name
] |
  [ expression ] MAP = contiguousmap_name
}
```

expression

An expression that evaluates to a valid hash bucket number.

MAP

An object that specifies which AMPs store the rows of a table.

sparsemap_name

The name of the sparse map, the map that includes a subset of AMPs from a contiguous map.

COLOCATE USING

A clause that forces tables that use the same sparse map to be stored on the same subset of AMPs.

COLOCATE USING is required with a sparse map. It cannot be used with a contiguous map.

colocation_name

The colocation name, usually *databasename_tablename*.

contiguousmap_name

The name of the contiguous map, the map that includes all AMPs within a specified range.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

About Default Arguments

Arguments	Result
None	`HASHBAKAMP ()` returns an INTEGER that is one less than the maximum number of AMPs in the system.
expression is not used	For a contiguous map: The function returns an INTEGER representing the highest fallback AMP number in the specified or default contiguous map. For a contiguous map starting at AMP zero, adding one to the result gives the total number of AMPs in the contiguous map. For a sparse map: *Expression* must be specified for a sparse map.
expression is specified	The function returns the ID of the fallback AMP corresponding to the hash bucket number specified in *expression*, based on the specified or default map.

About the Default Map

To determine the default map, the system evaluates the following, in order, and uses the first map found:

1. User profile.

2. User.
3. System.

About *Expression*

The *expression* argument must evaluate to an INTEGER data type where the range of valid values depends on the system hash bucket size.

IF the hash bucket size is …	THEN the range of values for *expression* is …
16 bits	0 to 65,535.
20 bits	0 to 1,048,575.

For information on how to set the system hash bucket size, see “DBS Control utility” in *Teradata Vantage™ - Database Utilities*, B035-1102.

If *expression* cannot be implicitly converted to an INTEGER, an error is reported.

If *expression* results in a UDT, Vantage performs implicit type conversion on the UDT, if the UDT has an implicit cast that casts between the UDT and any of the following predefined types:

- Numeric
- Character
- DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including HASHBAKAMP, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

Examples

Example Assumptions

The following example assumes a table T with an INTEGER column B populated with integer numbers from zero to the maximum number of hash buckets on the system.

Example: Distributing the Hash Buckets Among the Fallback AMPs

This query returns the distribution of the hash buckets among the fallback AMPs.

```
    SELECT B, HASHBAKAMP (B)
    FROM T
    ORDER BY 1;
```

Example: Which Fallback AMPs Contain the Rows of a Table

The following query shows which fallback AMPs contain the rows of a table.

```
sel col1, hashbakamp(hashbucket(hashrow(col1))
          map = map1count3 colocate using u1_tabm1c3)
          (named "which amp?")
from u1.tabm1c3 order by 1;

       col1   which amp?
-----------  -----------
          0            2
          1            3
          2            2
          3            3
          4            3
          5            3
          6            2
          7            3
          8            2
          9            0
```

Related Information

For more information on implicit type conversion, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

HASHBUCKET

Returns the hash bucket number that corresponds to a specified row hash value. If no row hash value is specified, HASHBUCKET returns the highest hash bucket number.

HASHBUCKET returns an INTEGER data type.

HASHBUCKET Function Syntax

```
HASHBUCKET ( expression )
```

Syntax Elements

expression

An optional expression that evaluates to a valid BYTE(4) row hash value.

If *expression* results in a UDT, Vantage performs implicit type conversion on the UDT, provided that the UDT has an implicit cast to a predefined byte type.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including HASHBUCKET, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

IF *expression* …	THEN …
does not appear in the argument list	HASHBUCKET returns an INTEGER value that is the highest hash bucket number.
evaluates to NULL	HASHBUCKET returns NULL.
evaluates to a valid BYTE(4) row hash value	HASHBUCKET returns the hash bucket number corresponding to the row hash value. The range of values for hash bucket numbers depends on the system setting of the hash bucket size. • If the hash bucket size is 16 bits, the hash bucket numbers can have a value from 0 to 65535. • If the hash bucket size is 20 bits, the hash bucket numbers can have a value from 0 to 1048575.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

HASHBUCKET Usage Notes

Using HASHBUCKET to Convert a BYTE Type to an INTEGER Type

When a byte data type is the source type of a conversion using CAST syntax or Teradata Conversion syntax, the target data type must also be a byte type.

To convert a BYTE(1) or BYTE(2) data type to INTEGER, you can use the HASHBUCKET function.

Consider the following table definition:

```
CREATE TABLE ByteData(b1 BYTE(1), b2 BYTE(2));
```

To convert column b1 to INTEGER regardless of the system setting of the hash bucket size, use the following:

```
SELECT HASHBUCKET('00'XB || b1 (BYTE(4))) / ((HASHBUCKET()+1)/65536)
FROM ByteData;
```

To convert column b2 to INTEGER regardless of the system setting of the hash bucket size, use the following:

```
SELECT HASHBUCKET(b2 (BYTE(4))) / ((HASHBUCKET()+1)/65536)
FROM ByteData;
```

Examples

Example Assumptions

The following examples assume a table T with columns C1 and C2 and possibly other columns.

Example

If you call HASHBUCKET without an argument, it returns the maximum hash bucket.

```
SELECT HASHBUCKET();
```

Example

If you call a HASHBUCKET function with an argument of NULL, the function returns NULL.

```
SELECT HASHBUCKET(NULL);
```

Example

Building on the previous example, you can nest a call to HASHROW within a HASHBUCKET call.

Calling HASHBUCKET (HASHROW (NULL)) returns the 0 hash bucket.

```
SELECT HASHBUCKET(HASHROW(NULL));
```

Example

The following example returns the number of rows in each hash bucket where C1 and C2 are to be the primary index of T.

```
SELECT HASHBUCKET (HASHROW (C1,C2)), COUNT (*)
   FROM T
   GROUP BY 1
   ORDER BY 1;
```

Example

The results of the following example lists each hash bucket that has one or more rows and its corresponding primary AMP.

```
SELECT HASHAMP (HASHBUCKET (HASHROW (C1, C2))),
   HASHBUCKET (HASHROW (C1,C2))
   FROM T
   GROUP BY 1,2
   ORDER BY 1,2 ;
```

HASHROW

Returns the hexadecimal row hash value for an expression or sequence of expressions. If no expression is specified, HASHROW returns the maximum hash code value.

The resulting row hash value is typed BYTE(4).

HASHROW Function Syntax

```
HASHROW ( [ expression [,...] ] )
```

Syntax Elements

expression

An optional expression or comma-separated list of expressions that can appear in the expression list of the select clause of a SELECT statement; typically a comma-separated list of column names that make up a (potential) index.

HASHROW does not support expressions that result in UDT data types.

IF *expression* is …	THEN HASHROW …
empty	returns the maximum hash code value.
an expression that evaluates to NULL	returns '00000000'XB.
a list of expressions where all the expressions evaluate to NULL	
an expression that evaluates to 0, '', ' ', or a similar value	
a valid, non-NULL expression that can appear in the select list of a SELECT statement	evaluates *expression* or the list of *expressions* and applies the hash function on the result. HASHROW returns the resulting row hash value.
a list of expressions that can appear in the select list of a SELECT statement, where some expressions can evaluate to NULL	

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

HASHROW is particularly useful for identifying the statistical properties of the current primary index, or to evaluate these properties for other columns to determine their suitability as a future primary index. You can also use these statistics to help minimize hash synonyms and enhance the uniformity of data distribution.

There are a maximum of 4,294,967,295 hash codes available in the system, ranging from '00000000'XB to 'FFFFFFFF'XB.

You can embed a HASHROW call within a HASHBUCKET call.

Examples

Example

If you call HASHROW without an argument, it returns 'FFFFFFFF'XB, which is the maximum hash code in the system.

```
SELECT HASHROW();
```

Example

The following example returns the average number of rows per row hash, where columns date_field and time_field constitute the primary index of the table eventlog.

```
SELECT COUNT(*) / COUNT(DISTINCT HASHROW (date_field,time_field))
FROM eventlog;
```

If columns date_field and time_field qualify for a unique index, this example returns the average number of rows with the same hash synonym.

Example

The following example evaluates the efficiency of changing the decimal format of a numeric field to eliminate synonyms.

Assume that column_1 and column_2 are declared as DECIMAL(2,2).

You can determine the effect of reformatting the columns to DECIMAL(8,6) and DECIMAL(8,4) on hash collisions by submitting these two queries.

```
SELECT COUNT (DISTINCT column_1(DECIMAL(8,6)) ||
column_2(DECIMAL(8,4))
FROM T;

SELECT COUNT (DISTINCT HASHROW (column_1(DECIMAL(8,6)),
column_2 (DECIMAL(8,4)))
FROM T;
```

If the result of the second query is significantly less than the result of the first query, there are a significant number of hash collisions. That is, the closer the second result is to the first value indicates elimination of more hash synonyms.

Related Information

- For information on HASHBUCKET, see HASHBUCKET.

Logical Predicates

The following sections describe SQL logical predicates.

Logical predicates are also known as comparison operators and conditional expressions. The ANSI SQL standard calls logical predicates search conditions.

Related Information:

Comparison Operators and Functions

About Logical Predicates

A logical predicate tests an operand against one or more other operands to evaluate to a logical (Boolean TRUE, FALSE, or UNKNOWN) result.

The tested operand can be one of the following:

- A column name
- A literal
- An arithmetic expression
- A Period expression, including a derived period
- The DEFAULT function
- A built-in function such as CURRENT_DATE or USER that evaluates to a system variable

Where Logical Predicates Are Used

Logical predicates are typically used in a WHERE, ON, or HAVING clause to qualify or disqualify rows as a table expression is evaluated in a SELECT statement.

Logical predicates can be used in a WHEN clause search condition in a searched CASE expression.

The type of test performed is a function of the predicate.

Conditional Expressions as a Collection of Logical Primitives

You can think of a conditional expression as a collection of logical predicate primitives where the order of evaluation is controlled by the use of the logical operators AND, OR, and NOT and by the placement of parentheses.

Superficially similar conditional expressions can produce radically different results depending on how you group their component primitives, so use caution in planning the logic of any conditional expressions.

SQL supports the logical predicate primitives listed in the following table. Note that Match and Unique conditions are not supported.

Logical Predicate Primitive Condition	SQL Logical Predicate	Function
Comparison	For a complete list of SQL comparison operators, see Supported Comparison Operators.	Tests for equality, inequality, or magnitude difference between two data values.
Range	BETWEEN NOT BETWEEN	Tests whether a data value is included within (or excluded from) a specified range of column data values.
Like	LIKE	Tests for a pattern match between a specified character string and a column data value.
In	IN NOT IN	Tests whether a data value is (or is not) a member of a specified set of column values. IN is equivalent to = ANY. NOT IN is equivalent to <> ALL.
All	ALL	Tests whether a data value compares TRUE to all column values in a specified set.
Any	ANY SOME	Tests whether a data value compares TRUE to any column value in a specified set.
Exists	EXISTS NOT EXISTS	Tests whether a specified table contains at least one row.
Period predicates	For a complete list of period predicate operators, see "Period Functions and Operators" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.	Operates on: • Two Period expressions • Two derived periods • One Period expression and one derived period • One Period expression and one DateTime expression Evaluates to TRUE, FALSE, or UNKNOWN.
	OVERLAPS	Tests whether two time periods, including derived periods, overlap
	IS UNTIL_CHANGED IS NOT UNTIL_CHANGED	Tests whether the ending bound of a Period expression, including a derived period is (or is not) UNTIL_CHANGED.

Restrictions on the Data Types Involved in Predicates

The restrictions in the following table apply to operations involving predicates and CLOB, BLOB, and UDT types.

Data Type	Restrictions
BLOB	Predicates do not support BLOB or CLOB data types. You can explicitly cast BLOBs to BYTE and VARBYTE types and CLOBs to CHARACTER and VARCHAR types, and use the results in a predicate.
CLOB	
UDT	The LIKE and OVERLAPS logical predicates do not support UDTs. For EXISTS and NOT EXISTS: Multiple UDTs involved as predicate operands must be identical types because Vantage does not perform implicit type conversion on UDTs involved as predicate operands. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs and then explicitly invoke the CAST function within the operation involving predicates. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. For BETWEEN/ NOT BETWEEN and IN/NOT: • Multiple UDTs involved as predicate operands must be identical types because Vantage does not perform implicit type conversion on UDTs involved as predicate operands. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs and then explicitly invoke the CAST function within the operation involving predicates. • UDTs involved as predicate operands must have ordering definitions. Vantage generates ordering functionality for distinct UDTs where the source types are not LOBs. To create an ordering definition for structured UDTs or distinct UDTs where the source types are LOBs, or to replace system-generated ordering functionality, use CREATE ORDERING. For more information on CREATE CAST and CREATE ORDERING, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Restrictions on the DEFAULT Function in a Predicate

The DEFAULT function returns the default value of a column. It has two forms: one that specifies a column name and one that omits the column name. Predicates support both forms of the DEFAULT function, but the following conditions must be true when the DEFAULT function omits the column name:

- The predicate uses a comparison operator
- The comparison involves a single column reference
- The DEFAULT function is not part of an expression

For example, the following statement uses DEFAULT to compare the values of the Dept_No column with the default value of the Dept_No column. Because the comparison operation involves a single column reference, Vantage can derive the column context of the DEFAULT function even though the column name is omitted.

```
SELECT * FROM Employee WHERE Dept_No < DEFAULT;
```

Note that if the DEFAULT function evaluates to null, the predicate is unknown and the WHERE condition is false.

Logical Operators and Search Conditions

A search condition, or conditional expression, consists of one or more conditional terms connected by one or more of the following logical predicates:

- Comparison operators
- BETWEEN/NOT BETWEEN
- LIKE
- IN/NOT IN
- ALL or ANY/SOME
- EXISTS/NOT EXISTS
- OVERLAPS
- IS NULL/IS NOT NULL

Logical Operators

An operator applied to the result of a predicate to determine the result of a search condition.

The logical operators are:

- AND
- NOT
- OR

For example:

```
expression_1 OR expression_2 OR expression_3
```

Use NOT to negate an expression, for example:

```
expression_1 AND NOT expression_2
```

Where To Use Search Conditions

A search condition can be used in various SQL clauses such as WHERE, ON, QUALIFY, RESET WHEN, or HAVING.

When used in a HAVING clause, a logical expression can be used with an aggregate operator.

For example, the following query uses a search condition in a HAVING clause to select from the Employee table those departments with the number 100, 300, 500, or 600, and with a salary average of at least $35,000 but not more than $55,000:

```
    SELECT AVG(Salary)
    FROM Employee
    WHERE DeptNo IN (100,300,500,600)
    GROUP BY DeptNo
    HAVING AVG(Salary) BETWEEN 35000 AND 55000 ;
```

Rules for Order of Evaluation

The following rules apply to evaluation order for conditional expressions:

- If an expression contains more than one of the same operator, the evaluation precedence is left to right.
- If an expression contains a combination of logical operators, the order of evaluation is as follows:

```
NOT
AND
OR
```

- Parentheses can be used to establish the desired evaluation precedence.
- The logical expressions in a conditional expression are not always evaluated left to right.

  Avoid using a conditional expression if its accuracy depends on the order in which its logical expressions are evaluated.

  For example, compare the following two expressions:

  ```
      F2/(NULLIF(F1,0)) > 500
      F1 <> 0 AND F2/F1 > 500
  ```

  The first expression guarantees exclusion of division by zero.

  The second allows the possibility of error, because the order of its evaluation determines the exclusion of zeros.

Evaluation Results

Each logical expression in a conditional expression evaluates to one of three results:

- TRUE
- FALSE
- UNKNOWN

AND Truth Table

The following table illustrates the AND logic used in evaluating search conditions.

	x FALSE	x UNKNOWN	x TRUE
y FALSE	FALSE	FALSE	FALSE

y UNKNOWN	FALSE	UNKNOWN	UNKNOWN
y TRUE	FALSE	UNKNOWN	TRUE

OR Truth Table

The following table illustrates the OR logic used in evaluating search conditions.

	x FALSE	x UNKNOWN	x TRUE
y FALSE	FALSE	UNKNOWN	TRUE
y UNKNOWN	UNKNOWN	UNKNOWN	TRUE
y TRUE	TRUE	TRUE	TRUE

NOT Truth Table

The following table illustrates the NOT logic used in evaluating search conditions.

	Result
x FALSE	TRUE
x UNKNOWN	UNKNOWN
x TRUE	FALSE

Subquery Restrictions

Predicates in search conditions cannot specify SELECT AND CONSUME statements in subqueries.

Examples of Logical Operators in Search Conditions

The following examples illustrate the use of logical operators in search conditions.

The following example uses a search condition to select from a user table named Profile the names of applicants who have either more than two years of experience or at least twelve years of schooling with a high school diploma:

```
SELECT Name
FROM Profile
WHERE YrsExp > 2
OR (EdLev >= 12 AND Grad = 'Y') ;
```

The following statement requests a list of all the employees who report to manager number 10007 or manager number 10012. The manager information is contained in the Department table, while the employee information is contained in the Employee table. The request is processed by joining the tables on DeptNo, their common column.

DeptNo must be fully qualified in every reference to avoid ambiguity and an extra set of parentheses is needed to group the ORed IN conditions. Without them, the result is a Cartesian product.

```
SELECT EmpNo,Name,JobTitle,Employee.DeptNo,Loc
FROM Employee,Department
WHERE (Employee.DeptNo=Department.DeptNo)
AND ((Employee.DeptNo IN
 (SELECT Department.DeptNo
  FROM Department
  WHERE MgrNo=10007))
  OR (Employee.DeptNo IN
   (SELECT Department.DeptNo
    FROM Department
    WHERE MgrNo=10012))) ;
```

Assuming that the Department table contains the following rows:

DeptNo	Department	Loc	MgrNo
100	Administration	NYC	10005
600	Manufacturing	CHI	10007
500	Engineering	ATL	10012
300	Exec Office	NYC	10018
700	Marketing	NYC	10021

The join statement returns:

EmpNo	Name	JobTitle	DeptNo	Loc
10012	Watson L	Vice Pres	500	ATL
10004	Smith T	Engineer	500	ATL
10014	Inglis C	Tech Writer	500	ATL
10009	Marston A	Secretary	500	ATL
10006	Kemper R	Assembler	600	CHI
10015	Omura H	Programmer	500	ATL
10007	Aguilar J	Manager	600	CHI

EmpNo	Name	JobTitle	DeptNo	Loc
10010	Reed C	Technician	500	ATL
10013	Regan R	Purchaser	600	CHI
10016	Carter J	Engineer	500	ATL
10019	Newman P	Test Tech	600	CHI

ANY/ALL/SOME

Enables quantification in a comparison operation or IN/NOT IN predicate.

ANY/ALL/SOME Predicate Syntax

```
{ expression quantifier ( literal [ {, | OR} ... ] ) |
  { expression | ( expression [,...] ) } quantifier ( subquery )
}
```

Syntax Elements

expression

An expression that specifies a value.

quantifier

```
{ comparison_operator [ NOT ] IN } { ALL |ANY | SOME }
```

literal

A literal value.

subquery

A subquery that selects the same number of expressions as are specified in the expression or list of expressions.

The subquery cannot specify a SELECT AND CONSUME statement.

comparison_operator

A comparison operator that compares the expression or list of expressions and the literals in the list (Literals syntax) or the subquery (Subquery syntax) to produce a TRUE, FALSE or UNKNOWN result.

[NOT] IN

A predicate that tests the existence of the expression or list of expressions in the list of literals (Literals syntax) or the subquery (Subquery syntax) to produce a TRUE, FALSE, or UNKNOWN result.

ANSI Compliance

ANY, SOME, and ALL are ANSI SQL:2011 compliant quantifiers.

ANY/ALL/SOME Usage Notes

ANY/ALL/SOME Quantifiers and Literal Syntax

When a list of literals is used with quantifiers and comparison operations or IN/NOT IN predicates, the results are determined as follows.

IF the predicate is …	AND specifies …	THEN the result is true when …
a comparison operation	ALL	the comparison of *expression* and every literal in the list produces true results.
	ANY	the comparison of *expression* and any literal in the list is true.
	SOME	
IN	ALL	*expression* is equal to every literal in the list.
	ANY	*expression* is equal to any literal in the list.
	SOME	
NOT IN	ALL	*expression* is not equal to any literal in the list.
	ANY	*expression* is not equal to every literal in the list.
	SOME	

For comparison operations, implicit conversion rules are the same as for the comparison operators.

If *expression* evaluates to NULL, the result is considered to be unknown.

ANY/ALL/SOME Quantifiers and Subquery Syntax

When subqueries are used with quantifiers and comparison operations or IN/NOT IN predicates, the results are determined as follows.

IF this quantifier is specified …	AND the predicate is …	THEN the result is …	WHEN …
ALL	a comparison operation	TRUE	the comparison of *expression* and every value in the set of values returned by *subquery* produces true results.
	IN	TRUE	*expression* is equal to every value in the set of values returned by *subquery*.
	NOT IN	TRUE	*expression* is not equal to any value in the set of values returned by *subquery*.
ALL	a comparison operation	TRUE	*subquery* returns no values.
	IN		
	NOT IN		
ANY SOME	a comparison operation	TRUE	the comparison of *expression* and at least one value in the set of values returned by *subquery* is true.
	IN	TRUE	*expression* is equal to at least one value in the set of values returned by *subquery*.
	NOT IN	TRUE	*expression* is not equal to at least one value in the set of values returned by *subquery*.
	a comparison operation	FALSE	*subquery* returns no values.
	IN		
	NOT IN		

Equivalences Using ANY/ALL/SOME and Comparison Operators

The following table provides equivalences for the ANY/ALL/SOME quantifiers, where *op* is a comparison operator.

This …	Is equivalent to …
x *op* ALL (:a, :b, :c)	(x *op* :a) AND (x *op* :b) AND (x *op* :c)
x *op* ANY (:a, :b, :c)	(x *op* :a) OR (x *op* :b) OR (x *op* :c)
x *op* SOME (:a, :b, :c)	

Here are some examples.

This expression …	Is equivalent to …
x < ALL (:a, :b, :c)	(x < :a) AND (x < :b) AND (x < :c)
x > ANY (:a, :b, :c)	(x > :a) OR (x > :b) OR (x > :c)
x > SOME (:a, :b, :c)	

Equivalences Using ANY/ALL/SOME and IN/NOT IN

The following table provides equivalences for the ANY/ALL/SOME quantifiers, where *op* is IN or NOT IN.

This …	Is equivalent to …
NOT (x *op* ALL (:a, :b, :c))	x NOT *op* ANY (:a, :b, :c)
	x NOT *op* SOME (:a, :b, :c)
NOT (x *op* ANY (:a, :b, :c))	x NOT *op* ALL (:a, :b, :c)
NOT (x *op* SOME (:a, :b, :c))	

If op is NOT IN, then NOT op is IN, not NOT NOT IN.

Here are some examples.

This expression …	Is equivalent to …
NOT (x IN ANY (:a, :b, :c))	x NOT IN ALL (:a, :b, :c)
NOT (x IN ALL (:a, :b, :c))	x NOT IN ANY (:a, :b, :c)
NOT (x NOT IN ANY (:a, :b, :c))	x IN ALL (:a, :b, :c)
NOT (x NOT IN ALL (:a, :b, :c))	x IN ANY (:a, :b, :c)

Examples

Example: ANY Quantifier

The following statement uses a comparison operator with the ANY quantifier to select the employee number, name, and department number of anyone in departments 100, 300, and 500:

This Expression …	Is Equivalent to this expression…

SELECT EmpNo, Name, DeptNo
FROM Employee
	SELECT EmpNo, Name, DeptNo
	FROM Employee

This Expression …	Is Equivalent to this expression…

WHERE DeptNo = ANY (100,300,500) ;	WHERE (DeptNo = 100)
	OR (DeptNo = 300)
	OR (DeptNo = 500) ;
	and
	SELECT EmpNo, Name, DeptNo
	FROM Employee
	WHERE DeptNo IN (100,300,500) ;

Example: ALL Quantifier

Here is an example that uses a subquery in a comparison operation that specifies the ALL quantifier:

```
SELECT EmpNo, Name, JobTitle, Salary, YrsExp
FROM Employee
WHERE (Salary, YrsExp) >= ALL
 (SELECT Salary, YrsExp FROM Employee) ;
```

Example: ANY/ALL/SOME

This example shows the behavior of ANY/ALL/SOME.

Consider the following table definition and contents:

```
CREATE TABLE t (x INTEGER);
INSERT t (1);
INSERT t (2);
INSERT t (3);
INSERT t (4);
INSERT t (5);
```

IF you use this query …	THEN the result is …
SELECT * FROM t WHERE x IN ANY (1,2)	1, 2
SELECT * FROM t WHERE x = SOME (1,2)	1, 2
SELECT * FROM t WHERE x NOT IN ALL (1,2)	3, 4, 5
SELECT * FROM t WHERE NOT (x IN ANY (1,2))	3, 4, 5
SELECT * FROM t WHERE NOT (x = SOME (1,2))	3, 4, 5
SELECT * FROM t WHERE x NOT IN SOME (1, 2)	1, 2, 3, 4, 5

IF you use this query …	THEN the result is …
SELECT * FROM t WHERE x NOT = ANY (1, 2)	1, 2, 3, 4, 5
SELECT * FROM t WHERE x IN ALL (1,2)	no rows
SELECT * FROM t WHERE NOT (x NOT IN SOME (1,2))	no rows
SELECT * FROM t WHERE x = ALL (1,2)	no rows
SELECT * FROM t WHERE NOT (x NOT = ANY (1,2))	no rows

BETWEEN/NOT BETWEEN

Tests whether an expression value is between two other expression values.

BETWEEN/NOT BETWEEN Predicate Syntax

```
expr1 [NOT] BETWEEN expr2 AND expr3
```

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Usage Notes

The BETWEEN test is satisfied if the following condition is true.

```
   expression_2 <= expression_1 <= expression_3
```

If the BETWEEN test fails, no rows are returned.

The BETWEEN test is treated as two separate logical comparisons.

```
   expression_1 >= expression_2 AND expression_1 <= expression_3.
```

This expression …	Is equivalent to …
x BETWEEN y AND z	((x >= y) AND (x <=z))

Note that because *expression_1* is actually evaluated twice, using a nondeterministic function, such as RANDOM, can produce unexpected results.

Example

The following example uses a search condition in a HAVING clause to select from the Employee table those departments with the number 100, 300, 500, or 600, and with a salary average of at least $35,000 but not more than $55,000:

```
SELECT AVG(Salary)
FROM Employee
WHERE DeptNo IN (100,300,500,600)
GROUP BY DeptNo
HAVING AVG(Salary) BETWEEN 35000 AND 55000 ;
```

EXISTS/NOT EXISTS

Tests a specified table (normally a derived table) for the existence of at least one row (that is, it tests whether the table in question is non-empty).

EXISTS is supported as the predicate of the search condition in a WHERE clause.

EXISTS/NOT EXISTS Predicate Syntax

```
[NOT] EXISTS subquery
```

Syntax Elements

subquery

A subquery that selects the same number of expressions as are specified in the expression or list of expressions.

The subquery cannot specify a SELECT AND CONSUME statement.

The function of the EXISTS predicate is to test the result of *subquery*.

If execution of the subquery returns response rows then the where condition is considered satisfied.

Using the NOT qualifier for the EXISTS predicate reverses the sense of the test. Execution of the subquery does not, in fact, return any response rows. Instead, it returns a boolean result to indicate whether responses would or would not have been returned had they been requested.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

EXISTS/NOT EXISTS Usage Notes

Relationship Between EXISTS/NOT EXISTS and IN/NOT IN

EXISTS predicate tests the existence of specified rows of a subquery. In general, EXISTS can be used to replace comparisons with IN and NOT EXISTS can be used to replace comparisons with NOT IN. However, the reverse is not true. Some problems can be solved only by using EXISTS and/or NOT EXISTS predicate. For an example, see For ALL.

Example

To select rows of t1 whose values in column x1 are equal to the value in column x2 of t2, one of the following queries can be used:

```
SELECT *
FROM t1
WHERE x1 IN
 (SELECT x2
  FROM t2);
SELECT *
FROM t1
WHERE EXISTS
 (SELECT *
  FROM t2
  WHERE t1.x1=t2.x2);
```

To select rows of t1 whose values in column x1 are not equal to any value in column x2 of t2, you can use any one of the following queries:

```
SELECT *
FROM t1
WHERE x1 NOT IN
 (SELECT x2
  FROM t2);

SELECT *
FROM t1
WHERE NOT EXISTS
 (SELECT *
  FROM t2
  WHERE t1.x1=t2.x2);
```

```
SELECT 'T1 is not empty'
WHERE EXISTS
 (SELECT *
  FROM t1);

SELECT 'T1 is empty'
WHERE NOT EXISTS
 (SELECT *
  FROM t1);
```

EXISTS Predicate Versus NOT IN and Nulls

Use the NOT EXISTS predicate instead of NOT IN if the following conditions are true:

- Some column of the NOT IN condition is defined as nullable.
- Any rows from the main query with a null in any column of the NOT IN condition should always be returned.
- Any nulls returned in the select list of the subquery should not prevent any rows from the main query from being returned.

For example, if all of the previous conditions are true for the following query, use NOT EXISTS instead of NOT IN:

```
SELECT dept, DeptName
FROM Department
WHERE Dept NOT IN
 (SELECT Dept
  FROM Course);
```

The NOT EXISTS version looks like this:

```
SELECT dept, DeptName
FROM Department
WHERE NOT EXISTS
 (SELECT Dept
  FROM Course
  WHERE Course.Dept=Department.Dept);
```

That is, either Course.Dept or Department.Dept is nullable and a row from Department with a null for Dept should be returned and a null in Course.Dept should not prevent rows from Department from being returned.

For ALL

Two nested NOT EXISTS can be used to express a SELECT statement that embodies the notion of “for all (logical ∀) the values in a column, there exists (logical ∃) …”

For example the query to select a ‘true’ value if the library has at least one book for all the publishers can be expressed as follows:

```
SELECT 'TRUE'
WHERE NOT EXISTS
 (SELECT *
  FROM publisher pb
  WHERE NOT EXISTS
   (SELECT *
    FROM book bk
    WHERE pb.PubNum=bk.PubNum);
```

NOT EXISTS Clauses and Stored Procedures

You cannot specify a NOT EXISTS clause in a stored procedure conditional expression if that expression also references an alias for a local variable, parameter, or cursor.

NOT EXISTS and Recursive Queries

NOT EXISTS cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify the NOT EXISTS predicate.

Examples

Example: EXISTS with Correlated Subqueries

Select all student names who have registered in at least one class offered by some department.

```
SELECT SName, SNo
FROM student s
WHERE EXISTS
 (SELECT *
  FROM department d
  WHERE EXISTS
   (SELECT *
    FROM course c, registration r, class cl
    WHERE c.Dept=d.Dept
    AND c.CNo=r.CNo
    AND s.SNo=r.SNo
    AND r.CNo=cl.CNo
    AND r.Sec=cl.Sec));
```

The content of the student table is as follows.

Sname	SNo
Helen Chu	1
Alice Clark	2
Kathy Kim	3
Tom Brown	4

The content of the department table is as follows.

Dept	DeptName
100	Computer Science
200	Physic
300	Math
400	Science

The content of course table is as follows.

CNo	Dept
10	100
11	100
12	200
13	200
14	300

The content of the class table is as follows.

CNo	Sec
10	1
11	1

CNo	Sec
12	1
13	1
14	1

The content of the registration table is as follows.

CNo	SNo	Sec
10	1	1
10	2	1
11	3	1
12	1	1
13	2	1
14	1	1

The following rows are returned:

```
SName              SNo
-----------        ---
Helen Chu1           *
Alice Clark          2
Kathy Kim            3
```

Example: NOT EXISTS with Correlated Subqueries

Select the names of all students who have registered in at least one class offered by each department that offers a course.

```
SELECT SName, SNo
FROM student s
WHERE NOT EXISTS
 (SELECT *
  FROM department d
  WHERE d.Dept IN
   (SELECT Dept
    FROM course) AND NOT EXISTS
     (SELECT *
      FROM course c, registration r, class cl
```

```
          WHERE c.Dept=d.Dept
          AND c.CNo=r.CNo
          AND s.SNo=r.SNo
          AND r.CNo=cl.CNo
          AND r.Sec=cl.Sec)));
```

With the contents of the tables as in “Example: EXISTS with Correlated Subqueries”, the following rows are returned:

```
    SName          SNo
    -----          ---
    Helen Chu      1
```

Related Information

- For a full explanation of correlated subqueries, see “Correlated Subqueries” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

IN/NOT IN

Tests the existence of the value of an expression or expression list in a comparable set in one of two ways:

- Compares the value of an expression with values in an explicit list of literals.
- Compares values in a list of expressions with values and in a set of corresponding expressions in a subquery.

IN/NOT IN Predicate Syntax

```
expression_1 [NOT] IN
  { expression_2 | ( literal_specification [ { OR | , }...] ) }
```

Syntax Elements

expression_1

The value of the expression whose existence is to be tested in *expression_2* or in an explicit list of literals named by *literal*, *signed_literal* TO *signed_literal*, or *datetime_literal*.

The *expression_1* data type and the *literal* values must be compatible. Implicit conversion rules are the same as for the comparison operators.

IN

Specifies whether the test is inclusive or exclusive.

You can substitute any of the following for IN unless a list of literals is specified and includes *signed_literal_1* TO *signed_literal_2*:

- IN ANY
- IN SOME
- = ANY
- = SOME5

You can substitute:

- <> ALL
- NOT IN ALL

expression_2

The value in which the existence of *expression_1* is to be tested.

literal_specification

```
{ literal |
  signed_literal_1 TO signed_literal_2 |
  datetime_literal
}
```

literal

- literal
- macro parameter
- built-in value such as TIME or DATE

signed_literal_1* TO *signed_literal_2

A range of literals.

datetime_literal

An ANSI DateTime literal.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Using TO in a list of literals is a Teradata extension to the ANSI standard.

expression IN and NOT IN *expression* or literals

IN/NOT IN Usage Notes

Result

If IN is used with a list of literals, the result is true if the value of *expression_1* is:

- equal to any *literal* in the list,
- between *signed_literal_1* and *signed_literal_2*, inclusively, when *signed_literal_1* is less than or equal to *signed_literal_2*, or
- between *signed_literal_2* and *signed_literal_1*, inclusively, when *signed_literal_2* is less than *signed_literal_1*

If the value of *expression_1* is null, then the result is considered to be unknown.

If the value of *expression_1* is not null, and none of the conditions are satisfied for the result to be true, then the result is false.

Using this form, the IN search condition is satisfied if the expression is equal to any of the values in the list of literals; the NOT IN condition is satisfied if none of the values in the list of literals are equal to the expression.

THE condition is true for this form …	WHEN …
expression_1 IN *expression_2*	*expression_1* = *expression_2*
expression_1 NOT IN *expression_2*	*expression_1* <> *expression_2*
expression_1 IN (*const_1*, *const_2*)	(*expression_1* = *const_1*) OR (*expression_1* = *const_2*)
expression_1 NOT IN (*const_1*, *const_2*)	(*expression_1* <> *const_1*) AND (*expression_1* <> *const_2*)
expression_1 IN (*signed_const_1* TO *signed_const_2*) where *signed_const_1* <= *signed_const_2*	(*signed_const_1* <= *expression_1*) AND (*expression_1* <= *signed_const_2*)
expression_1 IN (*signed_const_1* TO *signed_const_2*) where *signed_const_2* < *signed_const_1*	(*signed_const_2* <= *expression_1*) AND (*expression_1* <= *signed_const_1*)
expression_1 NOT IN (*signed_const_1* TO *signed_const_2*) where *signed_const_1* <= *signed_const_2*	(*expression_1* < *signed_const_1*) OR (*expression_1* > *signed_const_2*)
expression_1 NOT IN (*signed_const_1* TO *signed_const_2*) where *signed_const_2* < *signed_const_1*	(*expression_1* < *signed_const_2*) OR (*expression_1* > *signed_const_1*)

Here are some examples.

This statement ...	Is equivalent to this statement ...
`SELECT DeptNo FROM Department WHERE DeptNo IN (500, 600);`	`SELECT DeptNo FROM Department WHERE DeptNo IN (500) OR (DeptNo = 600);`
`UPDATE Employee SET Salary=Salary + 200 WHERE DeptNo NOT IN (100, 700);`	`UPDATE Employee SET Salary=Salary + 200 WHERE (DeptNo ^= 100) AND (DeptNo ^= 700);`

Relationship Between IN/NOT IN and EXISTS/NOT EXISTS

In general, you can use EXISTS to replace comparisons with IN, and NOT EXISTS to replace comparisons with NOT IN. However, the reverse is not true. The solutions to some problems require using the EXISTS or NOT EXISTS predicate. For information on EXISTS and NOT EXISTS, see EXISTS/NOT EXISTS.

Equivalences Using IN/NOT IN, NOT, and ANY/ALL/SOME

The following table provides equivalences for the ANY/ALL/SOME quantifiers, where *op* is IN or NOT IN.

This usage ...	Is equivalent to ...
NOT (x *op* ALL (:a, :b, :c))	x NOT *op* ANY (:a, :b, :c)
	x NOT *op* SOME (:a, :b, :c)
NOT (x *op* ANY (:a, :b, :c))	x NOT *op* ALL (:a, :b, :c)
NOT (x *op* SOME (:a, :b, :c))	
NOT (x *op* (:a, :b, :c))	x NOT *op* (:a, :b, :c)

In the equivalences, if *op* is NOT IN, then NOT *op* is IN, not NOT NOT IN.

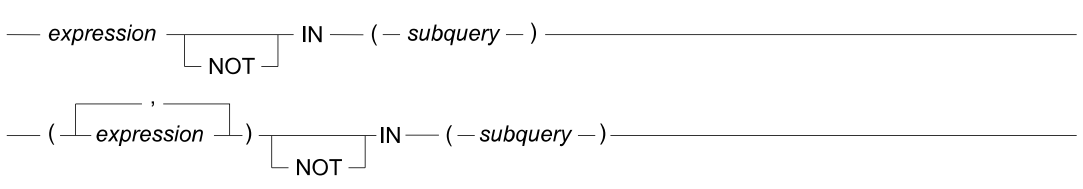
Here are some examples.

This expression ...	Is equivalent to ...
NOT (x IN ANY (:a, :b, :c))	x NOT IN ALL (:a, :b, :c)
NOT (x IN ALL (:a, :b, :c))	x NOT IN ANY (:a, :b, :c)
NOT (x NOT IN ANY (:a, :b, :c))	x IN ALL (:a, :b, :c)
NOT (x NOT IN ALL (:a, :b, :c))	x IN ANY (:a, :b, :c)

This expression …	Is equivalent to …
NOT (x IN (:a, :b, :c))	x NOT IN (:a, :b, :c)
NOT (x NOT IN (:a, :b, :c))	x IN (:a, :b, :c)

Syntax 2: *expression* IN and NOT IN *subquery*

This syntax for IN and NOT IN is correct in either of the following two forms:



Behavior of Nulls for IN

A statement result does not include column nulls when IN is used with a subquery.

Behavior of Nulls for NOT IN

The following table explains the behavior of nulls for NOT IN for queries of various forms.

FOR a query of the following form …	IF …	THEN …
`SELECT ... FROM T1 WHERE x NOT IN (SELECT y FROM T2);`	one of the y values is null	no T1 rows are returned for the entire query.
	some rows are returned by the subquery, and if x contains some nulls	those T1 rows that contain a null in x are not returned.
`SELECT ... FROM T1 WHERE` *expression_list_1* `NOT IN (SELECT` *expression_list_2* `FROM T2);`	a null is the first field in *expression_list_2*	no rows from T1 are returned.
	a null is in a field other than the first field of *expression_list_2*	some rows may be returned
	the subquery returns some rows, and if a null is in the first field in *expression_list_1*	the T1 rows containing a null in the first field of *expression_list_1* are not returned.
`SELECT ... FROM T1WHERE` *expression_list_1* `NOT IN (SELECT` *expression_list_2*	the *search_condition* on T2 returns no rows	all T1 rows, including those containing a NULL in the first field of *expression_list_1*, are returned.

FOR a query of the following form …	IF …	THEN …

`FROM T2`
`WHERE` *search_condition*`);`		

NOT IN Clauses and Stored Procedures

You cannot specify a NOT IN clause in a stored procedure conditional expression if that expression also references an alias for a local variable, parameter, or cursor.

NOT IN and Recursive Queries

NOT IN cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify the NOT IN predicate.

Queries With Large NOT IN Clauses Can Fail

Queries that contain thousands of arguments within an IN or NOT IN clause sometimes fail.

For example, suppose you ran the following query with 16000 IN clause arguments, and it failed.

```
SELECT MAX(emp_num)
FROM employee
WHERE emp_num IN(1,2,7,8,...,121347);
```

A workaround when this problem occurs is to rewrite the query using a temporary or volatile table to contain the arguments within the IN clause.

The following statements allow you to make the same selection, but without failure.

```
CREATE VOLATILE TABLE temp_IN_values (
 in_value INTEGER) ON COMMIT PRESERVE ROWS;

INSERT INTO temp_IN_values
SELECT emp_num
FROM table_with_emp_num_values;
```

The new query is as follows:

```
SELECT MAX(emp_num)
FROM employee AS e JOIN temp_IN_values AS en
ON (e.emp_num = en.in_value);
```

Examples

Example: Searching for Atlanta Employees

The following statement searches for the names of all employees who work in Atlanta.

```
SELECT Name
FROM Employee
WHERE DeptNo IN
 (SELECT DeptNo
  FROM Department
  WHERE Loc = 'ATL');
```

Example: Searching when DeptNo Has Two Columns

Using a similar example but assuming that the DeptNo is divided into two columns, the following statement could be used:

```
SELECT Name
FROM Employee
WHERE (DeptNoA, DeptNoB) IN
 (SELECT DeptNoA, DeptNoB
  FROM Department
  WHERE Loc = 'LAX') ;
```

Example: Using IN/NOT IN with a List of Literals

This example shows the behavior of IN/NOT IN with a list of literals.

Consider the following table definition and contents:

```
CREATE TABLE t (x INTEGER);
INSERT t (1);
INSERT t (2);
INSERT t (3);
INSERT t (4);
INSERT t (5);
```

IF you use this query …	THEN the result is …
SELECT * FROM t WHERE x IN (1,2)	1, 2

IF you use this query …	THEN the result is …
SELECT * FROM t WHERE x IN ANY (1,2)	1, 2
SELECT * FROM t WHERE NOT (x NOT IN (1,2))	1, 2
SELECT * FROM t WHERE x NOT IN (1,2)	3, 4, 5
SELECT * FROM t WHERE x NOT IN ALL (1,2)	3, 4, 5
SELECT * FROM t WHERE NOT (x IN (1, 2))	3, 4, 5
SELECT * FROM t WHERE NOT (x IN ANY (1,2))	3, 4, 5
SELECT * FROM t WHERE x IN (3 TO 5)	3, 4, 5
SELECT * FROM t WHERE x NOT IN SOME (1, 2)	1, 2, 3, 4, 5
SELECT * FROM t WHERE x IN (1, 2 TO 4, 5)	1, 2, 3, 4, 5
SELECT * FROM t WHERE x IN ALL (1,2)	no rows
SELECT * FROM t WHERE NOT (x NOT IN SOME (1,2))	no rows
SELECT * FROM t WHERE x NOT IN (1 TO 5)	no rows

IS NULL/IS NOT NULL

Searches for or excludes nulls in an expression.

IS NULL/IS NOT NULL Predicate Syntax

```
expression IS [NOT] NULL
```

Syntax Elements

expression

An expression that specifies a value that is tested for nulls.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Examples

Example

To search for the names of all employees who have not been assigned to a department, enter the following statement:

```
SELECT Name
FROM Employee
WHERE DeptNo IS NULL;
```

The result of this query is the names of all employees with a null in the DeptNo field.

Example

Conversely, to search for the names of all employees who have been assigned to a department, you could enter the following statement:

```
SELECT Name
FROM Employee
WHERE DeptNo IS NOT NULL;
```

This query returns the names of all employees with a value that is non-NULL in the DeptNo field.

Example: Searching for NULL and NOT-NULL in the Same Statement

If you are searching for values that are NULL and non-NULL in the same statement, the search condition for the NULLs must appear separately.

For example, to select the names of all employees without the job title of “Manager” or “Vice Pres”, plus the names of all employees with a null in the JobTitle column, you must enter the statement as follows:

```
SELECT Name, JobTitle
FROM Employee
WHERE (JobTitle NOT IN ('Manager' OR 'Vice Pres'))
OR (JobTitle IS NULL) ;
```

Example: Searching a Table That Might Contain Nulls

You must be careful when searching a table that might contain nulls. For example, if the EdLev column contains nulls and you submit the following query, the result contains only the names of employees with an education level of less than 16 years.

```
SELECT Name, EdLev
FROM Employee
WHERE (EdLev < 16) ;
```

To ensure that the result of a statement contains nulls, you must structure it as follows.

```
SELECT Name, EdLev
FROM Employee
WHERE (EdLev < 16)
OR (EdLev IS NULL) ;
```

LIKE/NOT LIKE

Searches for a character string pattern within another character string or character string expression.

LIKE/NOT LIKE Predicate Syntax

```
{ expression [NOT] LIKE
    { pattern_expression | [ ALL | ANY | SOME ] ( subquery ) } |

  ( expression [,...]) } [NOT] LIKE [ ALL | ANY | SOME ]
    { ( subquery ) | ( pattern_expression [,...] ) }

} [ ESCAPE escape_character ]
```

Syntax Elements

expression

A character string or character string expression argument to be searched for the substring pattern_expression.

pattern_expression

A character expression for which expression is to be searched.

ALL
ANY
SOME

A quantifier that allows one or more expressions to be searched for one or more patterns or for one or more values returned by a subquery.

subquery

```
{ expr | ( expr [,...] ) } [ NOT ] LIKE quantifier (subquery)
```

A subquery that selects the same number of expressions as are specified in the expression or list of expressions.

The subquery cannot specify a SELECT AND CONSUME statement.

escape_character

Keyword/variable combination specifying a single escape character (single or multibyte).

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

LIKE/NOT LIKE Usage Notes

Optimized Performance Using a NUSI

If it is cost-effective, the Optimizer may choose to evaluate a LIKE expression by scanning a NUSI with or without accessing the base table. The cost of using a NUSI depends on the selectivity of the LIKE expression, the size of the NUSI subtable, and if the NUSI is a covering index or a partially covering index. For a partially covering index, the cost of sorting the RowID spool is also included. For details on NUSIs and query covering, see *Teradata Vantage™ - Database Design*, B035-1094.

The Optimizer can perform a better cost comparison between using a NUSI and using an all-rows scan if the following are true:

- There are statistics collected for both the base table primary index and for the NUSI columns against which the *expression* string is evaluated.
- The *expression* string is either the mode or max value in at least one interval in the base table statistics histogram.

You cannot use a NUSI with a VARCHAR field for processing a LIKE expression when:

- the NUSI contains a VARCHAR field, and the VARCHAR field is used in a NOT LIKE operation.
- the NUSI contains a VARCHAR field, and the VARCHAR field is used in a string function.For example, the following is not allowed if d1 is a NUSI column of VARCHAR type.

```
d1||‘ab’ LIKE ‘b ab’
```

In addition, a NUSI with a VARCHAR field cannot be used as a partially covering index for an unconstrained aggregate query.

Null Expressions

If any expression in a comparison is null, the result of the comparison is unknown.

For a LIKE operation to provide a true result when searching fields that may contain nulls, the statement must include the IS [NOT] NULL operator.

Case Specification

If neither *pattern_expression* nor *expression* has been designated CASESPECIFIC, any lowercase letters in *pattern_expression* and *expression* are converted to uppercase before the comparison operation occurs. If ESCAPE is specified and the escape character is a lowercase character, it is also converted to uppercase before the comparison operation occurs.

If either *expression* or *pattern_expression* has been designated CASESPECIFIC, two letters match only if they are the same letters and the same case.

Wildcard Characters

The % and _ characters may be used in any combination in *pattern_expression*.

Character	Description
% (PERCENT SIGN)	Represents any string of zero or more arbitrary characters. Any string of characters is acceptable as a replacement for the percent.
_ (LOW LINE)	Represents exactly one arbitrary character. Any single character is acceptable in the position in which the underscore character appears.

The underscore and percent characters cannot be used in a pattern. To get around this, specify a single escape character in addition to *pattern_expression*. For details, see ESCAPE Feature of LIKE.

The following table describes how the metacharacters % and _ (and their fullwidth equivalents) behave when matching strings for various server character sets. Note that ANSI only defines the single byte spacing underscore and percent sign metacharacters.

Teradata SQL extends the permissible metacharacter set for the LIKE predicate to include the fullwidth underscore and the fullwidth percent sign.

FOR this server character set …	USE this metacharacter …	TO match this character or characters …	
		ANSI Mode	Teradata Mode
KANJI1	spacing underscore	any one single- or multibyte character.	any one single byte character.
	fullwidth spacing underscore	any one single byte character or multibyte character.	any one single byte character or multibyte character.
	percent sign	any sequence of single or multibyte characters.	any sequence of single byte characters or multibyte characters.

FOR this server character set …	USE this metacharacter …	TO match this character or characters …	
		ANSI Mode	Teradata Mode
	fullwidth percent sign	any sequence of single or multibyte characters.	any sequence of single byte characters or multibyte characters.
UNICODE LATIN KANJISJIS	fullwidth spacing underscore	none. These characters are not treated as metacharacters in order to maintain compliance with the ANSI SQL standard.	
	fullwidth percent		
GRAPHIC	fullwidth spacing underscore	any one single GRAPHIC character.	
	fullwidth percent sign	any sequence of GRAPHIC characters.	

ESCAPE Feature of LIKE

When the defined ESCAPE character is in the pattern string, it must be immediately followed by an underscore, percent sign, or another ESCAPE character.

In a left-to-right scan of the pattern string the following rules apply when ESCAPE is specified:

- Until an instance of the ESCAPE character occurs, characters in the pattern are interpreted at face value.
- When an ESCAPE character immediately follows another ESCAPE character, the two character sequence is treated as though it were a single instance of the ESCAPE character, considered as a normal character.
- When an underscore metacharacter immediately follows an ESCAPE character, the sequence is treated as a single underscore character (not a wildcard character).
- When a percent metacharacter immediately follows an ESCAPE character, the sequence is treated as a single percent character (not a wildcard character).
- When an ESCAPE character is not immediately followed by an underscore metacharacter, a percent metacharacter, or another instance of itself, the scan stops and an error is reported.

Pad Characters

The following notes apply to pad characters and how they are treated in strings:

- Pad characters are significant in both the character expression, and in the pattern string.
- When using pattern matching, be aware that both leading and trailing pad characters in the field or expression must match exactly with the pattern.

  For example, ‘A%BC’ matches ‘AxxBC’, but not ‘AxxBCΔ’, and ‘A%BCΔ’ matches ‘AxxBCΔ’, but not ‘AxxBC’ or ‘AxxBCΔΔ’ (Δ indicates a pad character).

- To retrieve the row in all cases, consider using the TRIM function, which removes both leading and trailing pad characters from the source string before doing the pattern match.

  For example, to remove trailing pad characters:

  ```
  TRIM (TRAILING FROM expression) LIKE pattern-string
  ```

  To remove leading and trailing pad characters:

  ```
  TRIM (BOTH FROM expression) LIKE pattern-string
  ```

- If *pattern_expression* is forced to a fixed length, trailing pad characters might be appended. In such cases, the field must contain the same number of trailing pad characters in order to match.

  For example, the following statement appends trailing pad characters to pattern strings shorter than 5 characters long.

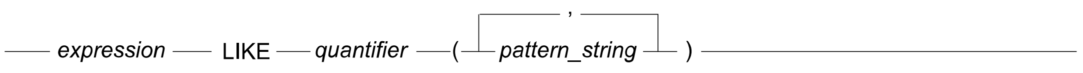
  ```
  CREATE MACRO (pattern (CHAR(5)) AS
  field LIKE :pattern…
  ```

- To retrieve the row in all cases, apply the TRIM function to the pattern string (TRIM (TRAILING FROM :pattern)), or the macro parameter can be defined as VARCHAR.

  These two methods do not always return the same results.TRIM removes pad characters, while the VARCHAR method maintains the data pattern exactly as entered.

ANY/ALL/SOME Quantifiers

SQL recognizes the quantifiers ANY (or SOME) and ALL. A quantifier allows one or more expressions to be compared with one or more values such as shown by the following generic example.



IF you specify this quantifier …	THEN the search condition is satisfied if *expression* LIKE *pattern_string* … is true for …
ALL	every string in the list.
ANY	any string in the list.

The ALL quantifier is the logical statement FOR ∀.

The ANY quantifier is the logical statement FOR ∃.

The following table restates this.

THIS expression …	IS equivalent to this expression …
x LIKE ALL ('A%','%B','%C%')	x LIKE 'A%' AND x LIKE '%B' AND x LIKE '%C%'
x LIKE ANY ('A%','%B','%C%')	x LIKE 'A%' OR x LIKE '%B' OR x LIKE '%C%'

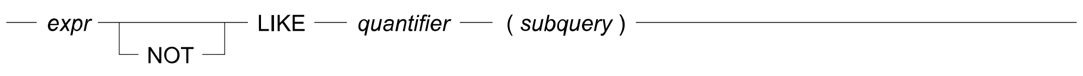
The following statement selects from the employee table the row of any employee whose job title includes the characters "Pres" or begins with the characters "Man":

```
SELECT *
FROM Employee
WHERE JobTitle LIKE ANY ('%Pres%', 'Man%');
```

The result of this statement is:

EmpNo	Name	DeptNo	JobTitle	Salary
10021	Smith T	700	Manager	45, 000.00
10008	Phan A	300	Vice Pres	55, 000.00
10007	Aguilar J	600	Manager	45, 000.00
10018	Russell S	300	President	65, 000.00
10012	Watson L	500	Vice Pres	56, 000.00

For the following forms, if you specify the ALL or ANY/SOME quantifier, then the subquery may return none, one, or several rows.



If, however, a quantifier is not used, then the subquery must return either no value or a single value as described in the following table.

This expression …	Is TRUE when *expression* matches …
expression LIKE (*subquery*)	the single value returned by subquery.
expression LIKE ANY (*subquery*)	at least one value of the set of values returned by subquery; is false if subquery returns no values.
expression LIKE ALL (*subquery*)	each individual value in the set of values returned by subquery, and is true if subquery returns no values.

Behavior of the ESCAPE Character

When *escape_character* is used in (generic) *string_2*, it must be followed immediately by a metacharacter of the appropriate server character set or another *escape_character*.

The resultant two-character sequence matches a single character in *string_1* if and only if the character in *string_1* collates identically to the character following the *escape_character* in *string_2*.

In other words, *escape_character* is ignored for matching purposes and the character following *escape_character* is matched for a single occurrence of itself.

When *string_1* and *string_2* do not share a common server character set, then the valid metacharacters are SPACING UNDERSCORE and PERCENT SIGN because the arguments are translated to UNICODE automatically when mismatched. Their behavior then follows the rules described in “Implicit Character-to-Character Translation”.

LIKE/NOT LIKE Examples

Example: ESCAPE

The following example illustrates the use of ESCAPE:

To look for the pattern ‘95%’ in a string such as ‘Result is 95% effective’, if Result is the field to be checked, use:

```
WHERE Result LIKE '%95Z%%' ESCAPE 'Z'
```

This clause finds the value ‘95%’.

Example: ANY

The following statement uses the ANY quantifier to retrieve every row from the Project table, which contains either the Accounts Payable or the Accounts Receivable project code:

```
   SELECT * FROM Project
   WHERE Proj_Id LIKE ANY
    (SELECT Proj_Id
     FROM Charges
     WHERE Proj_Id LIKE ANY ('A%')) ;
```

Example: Matching Patterns from Another Table

The following form of subquery might return none, one, or several values:

```
expr [ NOT ] LIKE quantifier (subquery)
```

The following example shows how you can match using patterns selected from another table.

There are two base tables.

This table ...	Defines these things ...
Project	• Unique project ID • Project description
Department_Proj	The association between project ID patterns and departments.

Department_Proj has two columns: Proj_pattern and Department. The rows in this table look like the following.

Proj_pattern	Department
AP%	Finance
AR%	Finance
Nut%	R&D
Screw%	R&D

The following query uses LIKE to match patterns selected from the Department_Proj table to select all rows in the Project table that have a Proj_Id that matches project patterns associated with the Finance department as defined in the Department_Proj table.

```
SELECT *
FROM Project
WHERE Proj_Id LIKE ANY
 (SELECT Proj_Pattern
  FROM Department_Proj
  WHERE Department = 'Finance');
```

When this syntax is used, the subquery must select the same number of expressions as are in the expression list:

```
( expr [,...] ) [ NOT ] LIKE quantifier (subquery)
```

For example:

```
(x,y) LIKE ALL (SELECT a,b FROM c)
```

is equivalent to:

```
(x LIKE c.a) AND (y LIKE c.b)
```

Example: LIKE Predicate

The following example uses the LIKE predicate to select a list of employees whose job title contains the string “Pres”:

```
SELECT Name, DeptNo, JobTitle
FROM Employee
WHERE JobTitle LIKE '%Pres%' ;
```

The form %string% requires Vantage to examine much of each string x. If x is long and there are many rows in the table, the search for qualifying rows may take a long time.

The result returned is:

Name	DeptNo	JobTitle
Watson L	500	Vice President
Phan A	300	Vice President
Russel S	300	President

Example: Last Name Spelling

This example selects a list of all employees whose last name begins with the letter P.

```
SELECT Name
FROM Employee
WHERE Name LIKE 'P%';
```

The result returned is:

```
Name
----------
```

```
Phan A
Peterson J
```

Example: % and _ Characters

This example uses the % and _ characters to select a list of employees with the letter A as the second letter in the last name. The length of the return string may be two or more characters.

```
SELECT Name
FROM Employee
WHERE Name LIKE '_a%';
```

returns the result:

```
Name
----------
Marston A
Watson L
Carter J
```

Replacing _a% with _a_ changes the search to a three-character string with the letter a as the second character. Because none of the names in the Employee table fit this description, the query returns no rows.

Both leading and trailing pad characters in a pattern are significant to the matching rules.

Example: Pad Characters and Letter

LIKE ’ΔΔZ%’ locates only those fields that start with two pad characters followed by Z.

KanjiEBCDIC Examples

The following examples indicate the behavior of LIKE with KanjiEBCDIC strings using the function (*expression* LIKE *pattern_expression*).

expression	*pattern_expression*	Server Character Set	Result
MN<AB>	%<B>	KANJI1	TRUE
MN<AB>P	<%B>%	KANJI1	TRUE
MN<AB>P	%P	KANJI1	TRUE
MN<AB>P	%<__C>%	KANJI1	FALSE

expression	*pattern_expression*	Server Character Set	Result
__ represents a FULLWIDTH UNDERSCORE.			

KanjiEUC Examples

The following examples indicate the behavior of LIKE with KanjiEUC strings using the function (*expression* LIKE *pattern_expression*).

expression	*pattern_expression*	Server Character Set	Result
ss3A ss2B ss3C ss2D	% ss2B%	UNICODE	TRUE
M ss2B N ss2D	M __%	GRAPHIC	TRUE
ss3A ss2B ss3C ss2D	__%	KANJISJIS	TRUE
ss3A ss2B ss3C ss2D	_ %	KANJISJIS	TRUE
__ represents a FULLWIDTH UNDERSCORE.			
_ represents a SPACING UNDERSCORE.			

KanjiShift-JIS Examples

The following examples indicate the behavior of LIKE with KanjiShift-JIS strings using the function (*expression* LIKE *pattern_expression*).

expression	*pattern_ expression*	Server Character Set	ANSI Mode Result	Teradata Mode Result
ABCD	__B%	GRAPHIC	TRUE	TRUE
mnABCI	%B%	UNICODE	TRUE	TRUE
mnABCI	%I	UNICODE	TRUE	TRUE
mnABCI	mn_%I	KANJI1	TRUE The underscore in *pattern_expression* matches a single byte- or multibyte character in ANSI mode.	FALSE The underscore in *pattern_expression* matches a single byte character in Teradata mode.
mnABCI	mn__%I	KANJI1	TRUE	TRUE
__ represents a FULLWIDTH UNDERSCORE.				
_ represents a SPACING UNDERSCORE.				

Miscellaneous Examples

Function	Result
_KanjiSJIS ‘92年abc’ LIKE _Unicode ‘%abc’	TRUE
_KanjiSJIS ‘92年abc’ LIKE _Unicode ‘%abc’	FALSE % (FULLWIDTH PERCENT SIGN) is not a metacharacter in either KanjiSJIS or Unicode.
‘c%’ LIKE ‘c%%’ ESCAPE ‘%’	TRUE
‘c%’ LIKE ‘c%%’ ESCAPE ‘%’	FALSE % (FULLWIDTH PERCENT SIGN) does not match % (PERCENT SIGN).

OVERLAPS

For OVERLAPS, see "Period Functions and Operators" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

12

Null-Handling Functions

The following sections describe functions that handle null input values.

NVL

Replaces a NULL with a numeric or a string value as the result value.

NVL is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

NVL Function Syntax

```
[TD_SYSFNLIB.] NVL ( expr1, expr2 )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expr1

A numeric or character expresssion.

- If *expr1* is NULL, *expr2* is returned.
- If *expr1* is not NULL, *expr1* is returned.

expr2

A numeric or character expresssion.

Argument Types and Rules

Expressions passed to this function must have the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, NUMBER, CHAR, VARCHAR

All of the input arguments must be the same data type or else the types must be compatible.

Result Type

NVL is a scalar function whose return value data type depends on the data type associated with the arguments passed to it.

- If the input arguments are numeric types, the function determines which argument has the highest precedence, converts the other argument to that data type, and returns that data type. For details about the order of precedence, see "Compatible Types" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

  If that data type is DECIMAL/NUMERIC and the precision and scale of the input arguments are different, the precision and scale of the return type is set to achieve the maximum precision possible. For example, if the input arguments are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return type would need three digits to the left of the decimal point and seven digits to the right of the decimal point to avoid any reduction in precision.

  In cases where it is not possible to maintain the maximum precision, the data is rounded according to the DBS Control Record RoundHalfWayMagUp field. For example, if the input arguments are DECIMAL(32, 8) and DECIMAL(30, 28), the return type will be DECIMAL(38,14). This will allow for 24 digits to the left of the decimal point (required for the DECIMAL(32,8) argument), and 14 digits to the right of the decimal point.

  If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).
- If the two arguments are character data types, the function converts the second argument to the data type of the first argument and returns the type as VARCHAR.
- If all input character types are LATIN, the result is LATIN. If any input is not LATIN, the function converts all input to Unicode and the return character set is Unicode.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

Example

The following query:

```
SELECT department_name,
   NVL(last_name,'NO EMPLOYEE') "LAST NAME"
```

```
FROM employee E
   FULL OUTER JOIN Department D
ON department_number=d.department_number;
WHERE department_number IN(402,600);
```

returns:

```
Department_name      LAST NAME
--------- --------------------
Software Support     Crane
New Department       NO EMPLOYEE
```

In this example, the NVL function returned the result as last_name column value from the employee table and for the cases where the last_name value is NULL, the function returned the result as NO EMPLOYEE.

Related Information

- For details, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

NVL2

Returns one of two values based on whether or not *expr1* is NULL.

NVL2 is an embedded services system function.

NVL2 Function Syntax

```
[TD_SYSFNLIB.] NVL2 ( expr1, expr2, expr3 )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expr1

A numeric or character expresssion.

- If *expr1* is not NULL, *expr2* is returned.
- If *expr1* is NULL, *expr3* is returned.

expr2

A numeric or character expresssion.

expr3

A numeric or character expresssion.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, NUMBER, CHAR, VARCHAR

expr2 and *expr3* must be the same data type or else the types must be compatible.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

NVL2 is a scalar function whose return data type depends on the data types associated with the arguments passed to the function.

- If *expr2* and *expr3* are numeric types, the function determines which argument has the highest precedence, converts the other argument to that data type, and returns that data type. For details about the order of precedence, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

  If that data type is DECIMAL/NUMERIC and the precision and scale of the two arguments are different, the precision and scale of the return type will be set to achieve the maximum precision possible. For example, if the input arguments are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return type would need three digits to the left of the decimal point and seven digits to the right of the decimal point to avoid any reduction in precision. In this case, the return data type is set to DECIMAL(10,7)

  In cases where it is not possible to maintain the maximum precision, the data will be rounded according to the DBS Control Record RoundHalfWayMagUp field. For example, if the two arguments are DECIMAL(32, 8) and DECIMAL(30, 28), the return type will be DECIMAL(38,14). This will allow for 24 digits to the left of the decimal point (required for the DECIMAL(32,8) argument), and 14 digits to the right of the decimal point.

  If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).
- If *expr2* and *expr3* are character types, the function converts the second argument to the data type of the first argument and returns the type as VARCHAR.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

Examples

Example

The following query:

```
SELECT NVL2('England', 'France', 'Spain');
```

returns the second argument, 'France', because the first argument is not NULL.

Example

The following query:

```
SELECT NVL2(NULL, 'France', 'Spain');
```

returns the third argument, 'Spain', because the first argument is NULL.

Related Information

- For information on activating and invoking embedded services functions, see Embedded Services System Functions.
- For details, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Ordered Analytical/Window Aggregate Functions

The following sections describe:

- Ordered analytical functions
- Window Aggregate Functions

Ordered Analytical Functions

Ordered analytical functions provide support for many common operations in analytical processing and data mining that require an ordered set of results rows or depend on values in a previous row. Ordered analytical functions enable and expedite the processing of queries containing On Line Analytical Processing (OLAP) style decision support requests.

For example, computing a seven-day running sum requires:

- First, that rows be ordered by date.
- Then, that the value for the running sum be computed by:
  - Adding the current row value to the value of the sum from the previous row, and
  - Subtracting the value from the row eight days ago.

Benefits

Ordered analytical functions use values from multiple rows to compute a new value.

The result of an ordered analytical function is handled the same as any other SQL expression. It can be a result column or part of a more complex arithmetic expression within its SELECT.

Each of the ordered analytical functions permit you to specify the sort ordering column or columns on which to sort the rows retrieved by the SELECT statement. The sort order and any other input parameters to the functions are specified the same as arguments to other SQL functions and can be any normal SQL expression.

Ordered Analytical Calculations at the SQL Level

Performing ordered analytical computations at the SQL level rather than through a higher-level OLAP calculation engine provides four distinct advantages.

- Reduced programming effort.
- Elimination of the need for external sort routines.
- Elimination of the need to export large data sets to external tools because ordered analytical functions enable you to target the specific data for analysis within the warehouse itself by specifying conditions in the query.

- Marked enhancement of analysis performance over the slow, single-threaded operations that external tools perform on large data sets.

Teradata Warehouse Miner

You need not directly code SQL queries to take advantage of ordered analytical functions. Both Vantage and many third-party query management and analytical tools have full access to the Teradata SQL ordered analytical functions. Teradata Warehouse Miner, for example, a tool that performs data mining preprocessing inside the database engine, relies on these features to perform functions in the database itself rather than requiring data extraction.

Teradata Warehouse Miner includes approximately 40 predefined data mining functions in SQL based on the Teradata SQL-specific functions. For example, the Teradata Warehouse Miner FREQ function uses the Teradata SQL-specific functions CSUM, RANK, and QUALIFY to determine frequencies.

Example

The following example shows how the SQL query to calculate a frequency of gender to marital status would appear using Teradata Warehouse Miner.

```
SELECT gender, marital_status, xcnt,xpct
   ,CSUM(xcnt, xcnt DESC, gender, marital_status) AS xcum_cnt
   ,CSUM(xpct, xcnt DESC, gender, marital_status) AS xcum_pct
   ,RANK(xcnt DESC, gender ASC, marital_status ASC) AS xrank
FROM
   (SELECT gender, marital_status, COUNT(*) AS xcnt
      ,100.000 * xcnt / xall (FORMAT 'ZZ9.99') AS xpct
   FROM customer_table A,
      (SELECT COUNT(*) AS xall
      FROM customer_table) B
GROUP BY gender, marital_status, xall
HAVING xpct >= 1) T1
QUALIFY xrank <= 8
ORDER BY xcnt DESC, gender, marital_status
```

The result for this query looks like the following table.

gender	marital_status	xcnt	xpct	xcum_cnt	xcum_pct	xrank
F	Married	3910093	36.71	3910093	36.71	1
M	Married	2419511	22.71	6329604	59.42	2
F	Divorced	1612130	15.13	7941734	74.55	3
M	Divorced	1412624	3.26	9354358	87.81	4

gender	marital_status	xcnt	xpct	xcum_cnt	xcum_pct	xrank
F	Single	491224	4.61	9845582	92.42	5
F	Widowed	319881	3.01	10165463	95.43	6
M	Single	319794	3.00	10485257	98.43	7
M	Widowed	197131	1.57	10652388	100.00	8

Characteristics of Ordered Analytical Functions

The Function Value

The function value for a column in a row considers that row (and a subset of all other rows in the group) and produces a new value.

The generic function describing this operation is as follows:

```
new_column_value = FUNCTION(column_value,rows_defined_by_window)
```

Use of QUALIFY Clause

Rows can be eliminated by applying conditions on the new column value. The QUALIFY clause is analogous to the HAVING clause of aggregate functions. The QUALIFY clause eliminates rows based on the function value, returning a new value for each of the participating rows. For example:

```
SELECT StoreID, SUM(profit) OVER (PARTITION BY StoreID)
FROM facts
QUALIFY SUM(profit) OVER (PARTITION BY StoreID) > 2;
```

An SQL query that contains both ordered analytical functions and aggregate functions can have both a QUALIFY clause and a HAVING clause, as in the following example:

```
SELECT StoreID, SUM(sale),
SUM(profit) OVER (PARTITION BY StoreID)
FROM facts
GROUP BY StoreID, sale, profit
HAVING SUM(sale) > 15
QUALIFY SUM(profit) OVER (PARTITION BY StoreID) > 2;
```

DISTINCT Clause Restriction

The DISTINCT clause is not permitted in window aggregate functions.

Permitted Query Objects

Ordered analytical functions are permitted in the following database query objects:

- Views
- Macros
- Derived tables
- INSERT ... SELECT

Where Ordered Analytical Functions are Not Permitted

Ordered analytical functions are not permitted in:

- Subqueries
- WHERE clauses
- SELECT AND CONSUME statements

Use of Standard SQL Features

You can use standard SQL features within the same query to make your statements more sophisticated.

For example, you can use ordered analytical functions in the following ways.

Use an analytical function in this operation …	To …
INSERT … SELECT	populate a new column.
derived table	create a new table to participate in a complex query.

Ordered analytical functions having different sort expressions are evaluated one after another, reusing the same spool file. Different functions having the same sort expression are evaluated simultaneously.

Unsupported Data Types

Ordered analytical functions do not operate on the following data types:

- CLOB or BLOB data types
- UDT data types

Note that CLOB, BLOB, or UDT data types are usable inside an expression if the result is a supported data type. For example:

```
SELECT
RANK() OVER
(PARTITION BY(CASE WHEN b IS NULL THEN 1 ELSE 0 END) ORDER BY id)
FROM btab;
```

However, the following example results in an error because the function cannot sort by BLOB:

```
SELECT
RANK() OVER
(PARTITION BY b ORDER BY id)
FROM btab;
```

Ordered Analytical Functions and Period Data Types

Expressions that evaluate to Period data types can be specified for any expression within the following ordered analytical functions: QUANTILE, RANK (Teradata-specific function), and RANK (ANSI SQL Window function).

Ordered Analytical Functions and Recursive Queries

Ordered analytical functions cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify an ordered analytical function.

Ordered Analytical Functions and Hash or Join Indexes

When a single table query specifies an ordered analytical function on columns that are also defined for a single table compressed hash or join index, the Optimizer does not select the hash or join index to process the query.

Ordered Analytical Functions and Row Level Security Tables

When a request that includes an ordered analytical function, such as MAVG, CSUM, or RANK, references a table protected by row level security, the operation is based on only the rows that are accessible to the requesting user. In order to apply all rows of the table to the function, the user must have one of the following:

- The required security credentials to access all rows of the table.
- The required OVERRIDE privileges on the security constraints in the table.

Computation Sort Order and Result Order

The sort order that you specify in the window specification defines the sort order of the rows over which the function is applied; it does not define the ordering of the results.

For example, to compute the average sales for the months following the current month, order the rows by month:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID ORDER BY SMonth
                 ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
```

```
FROM sales_tbl;

StoreID  SMonth  ProdID      Sales  Remaining Avg(Sales)
-------  ------  ------  ---------  --------------------
   1001       6  C        30000.00                     ?
   1001       5  C        30000.00              30000.00
   1001       4  C        25000.00              30000.00
   1001       3  C        40000.00              28333.33
   1001       2  C        25000.00              31250.00
   1001       1  C        35000.00              30000.00
```

The default sort order is ASC for the computation. However, the results are returned in the reverse order.

To order the results, use an ORDER BY phrase in the SELECT statement. For example:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID ORDER BY SMonth
                 ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
FROM sales_tbl
ORDER BY SMonth;

StoreID  SMonth  ProdID      Sales  Remaining Avg(Sales)
-------  ------  ------  ---------  --------------------
   1001       1  C        35000.00              30000.00
   1001       2  C        25000.00              31250.00
   1001       3  C        40000.00              28333.33
   1001       4  C        25000.00              30000.00
   1001       5  C        30000.00              30000.00
   1001       6  C        30000.00                     ?
```

Data in Partitioning Column of Window Specification and Resource Impact

The columns specified in the PARTITION BY clause of a window specification determine the partitions over which the ordered analytical function executes. For example, the following query specifies the StoreID column in the PARTITION BY clause to compute the group sales sum for each store:

```
SELECT StoreID, SMonth, ProdID, Sales,
SUM(Sales) OVER (PARTITION BY StoreID)
FROM sales_tbl;
```

At execution time, Vantage moves all of the rows that fall into a partition to the same AMP. If a very large number of rows fall into the same partition, the AMP can run out of spool space. For example, if the sales_tbl table in the preceding query has millions or billions of rows, and the StoreID column contains only a few distinct values, an enormous number of rows are going to fall into the same partition, potentially resulting in out-of-spool errors.

To avoid this problem, examine the data in the columns of the PARTITION BY clause. If necessary, rewrite the query to include additional columns in the PARTITION BY clause to create smaller partitions that Vantage can distribute more evenly among the AMPs. For example, the preceding query can be rewritten to compute the group sales sum for each store for each month:

```
SELECT StoreID, SMonth, ProdID, Sales,
SUM(Sales) OVER (PARTITION BY StoreID, SMonth)
FROM sales_tbl;
```

Using Ordered Analytical Functions

Example: Using RANK and AVG

Consider the result of the following SELECT statement using the following ordered analytical functions, RANK and AVG.

```
SELECT item, smonth, sales,
RANK() OVER (PARTITION BY item ORDER BY sales DESC),
AVG(sales) OVER (PARTITION BY item
                 ORDER BY smonth
                 ROWS 3 PRECEDING)
FROM sales_tbl
ORDER BY item, smonth;
```

The results table might look like the following.

Item	SMonth	Sales	Rank(Sales)	Moving Avg(Sales)
A	1996-01	110	13	110
A	1996-02	130	10	120
A	1996-03	170	6	137
A	1996-04	210	3	155
A	1996-05	270	1	195
A	1996-06	250	2	225
A	1996-07	190	4	230
A	1996-08	180	5	222
A	1996-09	160	7	195
A	1996-10	140	9	168

Item	SMonth	Sales	Rank(Sales)	Moving Avg(Sales)
A	1996-11	150	8	158
A	1996-12	120	11	142
A	1997-01	120	11	132
B	1996-02	30	5	30
...	...	...	...	...

Example: Using QUALIFY With RANK

Adding a QUALIFY clause to a query eliminates rows from an unqualified table.

For example, if you wanted to see whether the high sales months were unusual, you could add a QUALIFY clause to the previous query.

```
SELECT item, smonth, sales,
RANK() OVER (PARTITION BY item ORDER BY sales DESC),
AVG(sales) OVER (PARTITION BY item ORDER BY smonth ROWS 3 PRECEDING)
FROM sales_tbl
ORDER BY item, smonth
QUALIFY RANK() OVER(PARTITION BY item ORDER BY sales DESC) <=5;
```

This additional qualifier produces a results table that might look like the following.

Item	SMonth	Sales	Rank(Sales)	Moving Avg(Sales)
A	1996-04	210	3	155
A	1996-05	270	1	195
A	1996-06	250	2	225
A	1996-07	190	4	230
A	1996-08	180	5	222
B	1996-02	30	1	30
...	...	...	...	...

The result indicates that sales had probably been fairly low prior to the start of the current sales season.

Example: Using QUALIFY With RANK

Consider the following sales table named sales_tbl.

Store	ProdID	Sales
1003	C	20000.00
1003	D	50000.00
1003	A	30000.00
1002	C	35000.00
1002	D	25000.00
1002	A	40000.00
1001	C	60000.00
1001	D	35000.00
1001	A	100000.00
1001	B	10000.00

Now perform the following simple SELECT statement against this table, qualifying answer rows by rank.

```
SELECT store, prodID, sales,
RANK() OVER (PARTITION BY store ORDER BY sales DESC)
FROM sales_tbl
QUALIFY RANK() OVER (PARTITION BY store ORDER BY sales DESC) <=3;
```

The result appears in the following typical output table.

Store	ProdID	Sales	Rank(Sales)
1001	A	100000.00	1
1001	C	60000.00	2
1001	D	35000.00	3
1002	A	40000.00	1
1002	C	35000.00	2
1002	D	25000.00	3
1003	D	50000.00	1
1003	A	30000.00	2

Store	ProdID	Sales	Rank(Sales)
1003	C	20000.00	3

Note that every row in the table is returned with the computed value for RANK except those that do not meet the QUALIFY clause (sales rank is less than third within the store).

Related Information

- For more information about row level security, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.
- For details on the QUALIFY clause, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

The Window Feature

The ANSI SQL:2011 window feature provides a way to dynamically define a subset of data, or *window*, in an ordered relational database table. A window is specified by the OVER() phrase, which can include the following clauses inside the parentheses:

- PARTITION BY
- ORDER BY
- RESET WHEN
- ROWS

PARTITION BY Phrase

PARTITION BY takes a column reference list and groups the rows based on the specified column reference list over which the ordered analytical function executes. Such a grouping is static. To define a group or partition based on a condition, use the RESET WHEN phrase. For more information, see RESET WHEN Phrase.

If there is no PARTITION BY phrase or RESET WHEN phrase, then the entire result set, delivered by the FROM clause, constitutes a single partition, over which the ordered analytical function executes.

Consider the following table named sales_tbl.

StoreID	SMonth	ProdID	Sales
1001	1	C	35000.00
1001	2	C	25000.00
1001	3	C	40000.00
1001	4	C	25000.00
1001	5	C	30000.00

StoreID	SMonth	ProdID	Sales
1001	6	C	30000.00
1002	1	C	40000.00
1002	2	C	35000.00
1002	3	C	110000.00
1002	4	C	60000.00
1002	5	C	35000.00
1002	6	C	100000.00

The following SELECT statement, which does not include PARTITION BY, computes the average sales for all the stores in the table:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER ()
FROM sales_tbl;

StoreID  SMonth  ProdID      Sales  Group Avg(Sales)
-------  ------  ------  ---------  ----------------
   1001       1  C        35000.00          47083.33
   1001       2  C        25000.00          47083.33
   1001       3  C        40000.00          47083.33
   1001       4  C        25000.00          47083.33
   1001       5  C        30000.00          47083.33
   1001       6  C        30000.00          47083.33
   1002       1  C        40000.00          47083.33
   1002       2  C        35000.00          47083.33
   1002       3  C       110000.00          47083.33
   1002       4  C        60000.00          47083.33
   1002       5  C        35000.00          47083.33
   1002       6  C       100000.00          47083.33
```

To compute the average sales for each store, partition the data in sales_tbl by StoreID:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID)
FROM sales_tbl;

StoreID  SMonth  ProdID      Sales  Group Avg(Sales)
-------  ------  ------  ---------  ----------------
   1001       3  C        40000.00          30833.33
   1001       5  C        30000.00          30833.33
   1001       6  C        30000.00          30833.33
   1001       4  C        25000.00          30833.33
   1001       2  C        25000.00          30833.33
   1001       1  C        35000.00          30833.33
   1002       3  C       110000.00          63333.33
   1002       5  C        35000.00          63333.33
   1002       6  C       100000.00          63333.33
   1002       4  C        60000.00          63333.33
   1002       2  C        35000.00          63333.33
   1002       1  C        40000.00          63333.33
```

ORDER BY Phrase

ORDER BY specifies how the rows are ordered in a partition, which determines the sort order of the rows over which the function is applied.

To add the monthly sales for a store in the sales_tbl table to the sales for previous months, compute the cumulative sales sum and order the rows in each partition by SMonth:

```
SELECT StoreID, SMonth, ProdID, Sales,
SUM(Sales) OVER (PARTITION BY StoreID ORDER BY SMonth
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM sales_tbl;

StoreID  SMonth  ProdID      Sales  Cumulative Sum(Sales)
-------  ------  ------  ---------  ---------------------
   1001       1  C        35000.00               35000.00
   1001       2  C        25000.00               60000.00
   1001       3  C        40000.00              100000.00
   1001       4  C        25000.00              125000.00
   1001       5  C        30000.00              155000.00
   1001       6  C        30000.00              185000.00
   1002       1  C        40000.00               40000.00
   1002       2  C        35000.00               75000.00
   1002       3  C       110000.00              185000.00
   1002       4  C        60000.00              245000.00
   1002       5  C        35000.00              280000.00
   1002       6  C       100000.00              380000.00
```

RESET WHEN Phrase

RESET WHEN is a Teradata extension to the ANSI SQL standard.

Depending on the evaluation of the specified condition, RESET WHEN determines the group or partition, over which the ordered analytical function operates. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. To define a partition based on a column reference list, use the PARTITION BY phrase. For more information, see PARTITION BY Phrase.

If there is no RESET WHEN phrase or PARTITION BY phrase, then the entire result set, delivered by the FROM clause, constitutes a single partition, over which the ordered analytical function executes.

You can have different RESET WHEN clauses in the same SELECT list.

Note:

A window specification that specifies a RESET WHEN clause must also specify an ORDER BY clause.

RESET WHEN Condition Rules

The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify conditional partitioning.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

The following rules apply for RESET WHEN conditions.

A RESET WHEN condition can contain the following:

- Ordered analytical functions that do not include the RESET WHEN clause
- Scalar subqueries
- Aggregate operators
- DEFAULT functions

  However, DEFAULT without an explicit column specification is valid only if it is specified as a standalone condition in the predicate. For more information, see Rules For Using a DEFAULT Function As Part of a RESET WHEN Condition.

A RESET WHEN condition cannot contain the following:

- Ordered analytical functions that include the RESET WHEN clause
- The SELECT statement
- LOB columns
- UDT expressions, including UDFs that return a UDT value

  However, a RESET WHEN condition can include an expression that contains UDTs as long as that expression returns a result that has a predefined data type.

Rules For Using a DEFAULT Function As Part of a RESET WHEN Condition

The following rules apply to the use of the DEFAULT function as part of a RESET WHEN condition:

- You can specify a DEFAULT function with a column name argument within a predicate. The system evaluates the DEFAULT function to the default value of the column specified as its argument. After the system evaluates the DEFAULT function, it treats it like a literal in the predicate.
- You can specify a DEFAULT function without a column name argument within a predicate only if there is one column specification and one DEFAULT function as the terms on each side of the comparison operator within the expression.
- Following existing comparison rules, a condition with a DEFAULT function used with comparison operators other than IS [NOT] NULL is unknown if the DEFAULT function evaluates to null.

  A condition other than IS [NOT]NULL with a DEFAULT function compared with a null evaluates to unknown.

IF a DEFAULT function is used with...	THEN the comparison is...
IS NULL	TRUE if the default is null, else it is FALSE.
IS NOT NULL	FALSE if the default is null, else it is TRUE.

Examples

Example

This example finds cumulative sales for all periods of increasing sales for each region.

```
SUM(sales) OVER (
     PARTITION BY region
     ORDER BY day_of_calendar
     RESET WHEN sales < /* preceding row */ SUM(sales) OVER (
          PARTITION BY region
          ORDER BY day_of_calendar
          ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
     ROWS UNBOUNDED PRECEDING
)
```

Example

This example finds sequences of increasing balances. This implies that we reset whenever the current balance is less than or equal to the preceding balance.

```
SELECT account_key, month, balance,
ROW_NUMBER() over
     (PARTITION BY account_key
      ORDER BY month
      RESET WHEN balance /* current row balance */ <=
      SUM(balance) over (PARTITION BY account_key ORDER BY month
      ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) /* prev row */
      ) - 1 /* to get the count started at 0 */ as balance_increase
FROM accounts;
```

The possible results of the preceding SELECT appear in the table below:

```
account_key        month      balance  balance_increase
-----------        -----      -------  ----------------
          1            1           60                 0
          1            2           99                 1
          1            3           94                 0
          1            4           90                 0
          1            5           80                 0
          1            6           88                 1
          1            7           90                 2
          1            8           92                 3
          1            9           10                 0
          1           10           60                 1
          1           11           80                 2
          1           12           10                 0
```

Example

The following example illustrates a window function with a nested aggregate. The query is processed as follows:

1. We use the SUM(balance) aggregate function to calculate the sum of all the balances for a given account in a given quarter.
2. We check to see if a balance in a given quarter (for a given account) is greater than the balance of the previous quarter.

3. If the balance increased, we track a cumulative count value. As long as the RESET WHEN condition evaluates to false, the balance is increasing over successive quarters, and we continue to increase the count.
4. We use the ROW_NUMBER() ordered analytical function to calculate the count value. When we reach a quarter whose balance is less than or equal to that of the previous quarter, the RESET WHEN condition evaluates to true, and we start a new partition and ROW_NUMBER() restarts the count from 1. We specify ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING to access the previous value.
5. Finally, we subtract 1 to ensure that the count values start with 0.

The balance_increase column shows the number of successive quarters where the balance was increasing. In this example, we only have one quarter (1->2) where the balance has increased.

```
SELECT account_key, quarter, sum(balance),
ROW_NUMBER() over
     (PARTITION BY account_key
      ORDER BY quarter
      RESET WHEN sum(balance) /* current row balance */ <=
      SUM(sum(balance)) over (PARTITION BY account_key ORDER BY quarter
      ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)/* prev row */
      ) - 1 /* to get the count started at 0 */ as balance_increase
FROM accounts
GROUP BY account_key, quarter;
```

The possible results of the preceding SELECT appear in the table below:

```
account_key    quarter    balance    balance_increase
-----------    -------    -------    ----------------
          1          1        253                   0
          1          2        258                   1
          1          3        192                   0
          1          4        150                   0
```

Example

In the following example, the condition in the RESET WHEN clause contains SELECT as a nested subquery. This is not allowed and results in an error.

```
SELECT SUM(a1) OVER
     (ORDER BY 1
      RESET WHEN 1 in (SELECT 1))
FROM t1;
$
```

```
*** Failure 3706 Syntax error: SELECT clause not supported in
RESET...WHEN clause.
```

ROWS Phrase

ROWS defines the rows over which the aggregate function is computed for each row in the partition.

If ROWS is specified, the computation of the aggregate function for each row in the partition includes only the subset of rows in the ROWS phrase.

If there is no ROWS phrase, then the computation includes all the rows in the partition.

To compute the three-month moving average sales for each store in the sales_tbl table, partition by StoreID, order by SMonth, and perform the computation over the current row and the two preceding rows:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID
                 ORDER BY SMonth
                 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM sales_tbl;

StoreID  SMonth  ProdID      Sales  Moving Avg(Sales)
-------  ------  ------  ---------  -----------------
   1001       1  C        35000.00           35000.00
   1001       2  C        25000.00           30000.00
   1001       3  C        40000.00           33333.33
   1001       4  C        25000.00           30000.00
   1001       5  C        30000.00           31666.67
   1001       6  C        30000.00           28333.33
   1002       1  C        40000.00           40000.00
   1002       2  C        35000.00           37500.00
   1002       3  C       110000.00           61666.67
   1002       4  C        60000.00           68333.33
   1002       5  C        35000.00           68333.33
   1002       6  C       100000.00           65000.00
```

Multiple Window Specifications

In an SQL statement using more than one window function, each window function can have a unique window specification.

For example,

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID
                 ORDER BY SMonth
```

```
                  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW),
RANK() OVER (PARTITION BY StoreID ORDER BY Sales DESC)
FROM sales_tbl;
```

Related Information

- See DEFAULT for more information about the DEFAULT function.
- The window specification can also be applied to a user-defined aggregate function. For details, see SQL UDF.
- To see the syntax for the OVER() phrase and the associated clauses, see Window Aggregate Functions.

Window Aggregate Functions

An aggregate function on which a window specification is applied is called a window aggregate function. Without a window specification, aggregate functions return one value for all qualified rows examined. Window aggregate functions return a new value for each of the qualifying rows participating in the query.

Thus, the following SELECT statement, which includes the aggregate AVG, returns one value only: the average of sales.

```
SELECT AVG(sale)
FROM monthly_sales;

Average(sale)
-------------
         1368
```

The AVG window function retains each qualifying row.

The following SELECT statement might return the results that follow.

```
SELECT territory, smonth, sales,
AVG(sales) OVER (PARTITION BY territory
                 ORDER BY smonth ROWS 2 PRECEDING)
FROM sales_history;

territory  smonth   sales  Moving Avg(sales)
---------  -------  -----  -----------------
East        199810     10                 10
East        199811      4                  7
East        199812     10                  8
East        199901      7                  7
East        199902     10                  9
```

```
    West        199810      8                   8
    West        199811     12                  10
    West        199812      7                   9
    West        199901     11                  10
    West        199902      6                   8
```

The Window Specification

Cumulative, group, moving, or remaining computation of an aggregate function.

Window Specification Syntax

```
{ AVG ( value_expression ) |

  CORR ( value_expression_1, value_expression_2 ) |

  COUNT ( { value_expression | * } ) |

  COVAR_POP ( value_expression_1, value_expression_2 ) |

  COVAR_SAMP ( value_expression_1, value_expression_2 ) |

  MAX ( value_expression ) |

  MIN ( value_expression ) |

  REGR_AVGX ( dependent_variable_expression, independent_variable_expression ) |

  REGR_AVGY ( dependent_variable_expression, independent_variable_expression ) |

  REGR_COUNT ( dependent_variable_expression, independent_variable_expression ) |

  REGR_INTERCEPT ( dependent_variable_expression, independent_variable_expression ) |

  REGR_R2 ( dependent_variable_expression, independent_variable_expression ) |

  REGR_SLOPE ( dependent_variable_expression, independent_variable_expression ) |

  REGR_SXX ( dependent_variable_expression, independent_variable_expression ) |

  REGR_SXY ( dependent_variable_expression, independent_variable_expression ) |
  REGR_SYY ( dependent_variable_expression, independent_variable_expression ) |

  STDDEV_POP ( value_expression ) |

  STDDEV_SAMP ( value_expression ) |

  VAR_POP ( value_expression ) |

  VAR_SAMP ( value_expression )

} window
```

Syntax Elements

window

```
OVER ( [ partition_by_clause ] [ order_by_clause ] [ rows_clause ] )
```

partition_by_clause

```
PARTITION BY column_reference [,...]
```

In its *column_reference*, or comma-separated list of column references, the group, or groups, over which the function operates.

PARTITION BY is optional. If there are no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a single group, or partition.

PARTITION BY clause is also called the window partition clause.

order_by_clause

```
ORDER BY value_specification [,...] [ RESET WHEN condition ]
```

In its *value_expression* the order in which the values in a group, or partition, are sorted.

rows_clause

```
{ ROWS { { UNBOUNDED | value } PRECEDING | CURRENT ROW } |

  ROWS BETWEEN { { UNBOUNDED | value } PRECEDING AND

                    { { UNBOUNDED | value }
                          FOLLOWING | value PRECEDING | CURRENT ROW } |
                     CURRENT ROW AND { { UNBOUNDED | value }
FOLLOWING } |

                     value FOLLOWING AND { UNBOUNDED | value
} FOLLOWING
               }
}
```

value_specification

```
value_expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

OVER

How values are grouped, ordered, and considered when computing the cumulative, group, or moving function.

Values are grouped according to the PARTITION BY and RESET WHEN clauses, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.

RESET WHEN

The group or partition, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition.

RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set, delivered by the FROM clause, constitutes a single partition.

If RESET WHEN is specified, then the ORDER BY clause must be specified also.

condition

A conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

ROWS

The starting point for the aggregation group within the partition. The aggregation group end is the current row.

The aggregation group of a row R is a set of rows, defined relative to R in the ordering of the rows within the partition.

If there are no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

The default when there is no ROWS clause for FIRST_VALUE/LAST_VALUE is different. For more information, see FIRST_VALUE/LAST_VALUE.

ROWS BETWEEN

The aggregation group start and end, which defines a set of rows relative to the current row in the ordering of the rows within the partition.

The row specified by the group start must precede the row specified by the group end.

If there are no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

UNBOUNDED PRECEDING

The entire partition preceding the current row.

UNBOUNDED FOLLOWING

The entire partition following the current row.

CURRENT ROW

The start or end of the aggregation group as the current row.

***value* PRECEDING**

The number of rows preceding the current row.

The value for *value* is always a positive integer literal.

The maximum number of rows in an aggregation group is 4096 when *value* PRECEDING appears as the group start or group end.

***value* FOLLOWING**

The number of rows following the current row.

The value for *value* is always a positive integer literal.

The maximum number of rows in an aggregation group is 4096 when *value* FOLLOWING appears as the group start or group end.

ASC

That the results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

DESC

That the results are to be ordered in descending sort order.

If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

NULLS FIRST

NULL results are to be listed first.

NULLS LAST

NULL results are to be listed last.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

In the presence of an ORDER BY clause and the absence of a ROWS or ROWS BETWEEN clause, ANSI SQL:2011 window aggregate functions use ROWS UNBOUNDED PRECEDING as the default aggregation group, whereas Teradata SQL window aggregate functions use ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

Type of Computation

To compute this type of function …	Use this aggregation group …
Cumulative	• ROWS UNBOUNDED PRECEDING • ROWS BETWEEN UNBOUNDED PRECEDING AND *value* PRECEDING • ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW • ROWS BETWEEN UNBOUNDED PRECEDING AND *value* FOLLOWING
Group	ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
Moving	• ROWS *value* PRECEDING • ROWS CURRENT ROW • ROWS BETWEEN *value* PRECEDING AND *value* PRECEDING • ROWS BETWEEN *value* PRECEDING AND CURRENT ROW • ROWS BETWEEN *value* PRECEDING AND *value* FOLLOWING • ROWS BETWEEN CURRENT ROW AND CURRENT ROW • ROWS BETWEEN CURRENT ROW AND *value* FOLLOWING

To compute this type of function …	Use this aggregation group …
	• ROWS BETWEEN *value* FOLLOWING AND *value* FOLLOWING
Remaining	• ROWS BETWEEN *value* PRECEDING AND UNBOUNDED FOLLOWING • ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING • ROWS BETWEEN *value* FOLLOWING AND UNBOUNDED FOLLOWING

Arguments to Window Aggregate Functions

Window aggregate functions can take literals, literal expressions, column names (sales, for example), or column expressions (sales + profit) as arguments.

Window aggregates can also take regular aggregates as input parameters to the PARTITION BY and ORDER BY clauses. The RESET WHEN clause can take an aggregate as part of the RESET WHEN condition clause.

COUNT can take "*" as an input argument, as in the following SQL query:

```
SELECT city, kind, sales, profit,
COUNT(*) OVER (PARTITION BY city, kind
               ROWS BETWEEN UNBOUNDED PRECEDING AND
               UNBOUNDED FOLLOWING)
FROM activity_month;
```

Result Type and Format

The result data type and format for window aggregate functions are as follows.

Function	Result Type	Format
AVG(*x*) where *x* is a character type	FLOAT	Default format for FLOAT
AVG(*x*) where *x* is a numeric, DATE, or INTERVAL type	FLOAT	Same format as operand *x*

CORR(*x*,*y*)
COVAR_POP(*x*,*y*)
COVAR_SAMP(*x*,*y*)
REGR_AVGX(*y*,*x*)
REGR_AVGY(*y*,x)
REGR_INTERCEPT(*x*,*y*)
REGR_R2(*x*,*y*)
	FLOAT	Default format for FLOAT

Function	Result Type	Format

REGR_SLOPE(*x*,*y*)
REGR_SXX(*x*,*y*)
REGR_SXY(*x*,*y*)
REGR_SYY(*x*,*y*)
STDDEV_POP(*x*,)
STDDEV_SAMP(*x*,)
VAR_POP(*x*,)
VAR_SAMP(*x*)
where *x* is a character type		

CORR(*x*,*y*)
COVAR_POP(*x*,*y*)
COVAR_SAMP(*x*,*y*)
REGR_AVGX *(y,x)*
REGR_AVGY(*y*,*x*)
REGR_INTERCEPT(*x*,*y*)
REGR_R2(*x*,*y*)
REGR_SLOPE(*x*,*y*)
REGR_SXX(*x*,*y*)
REGR_SXY(*x*,*y*)
REGR_SYY(*x*,*y*)
STDDEV_POP(*x*)
STDDEV_SAMP(*x*)
VAR_POP(*x*)
VAR_SAMP(*x*)
where *x* is one of the following types:
• Numeric
• DATE
• Interval
	Same data type as operand *x*.	Default format for the data type of operand *x*

REGR_AVGX(*y*,*x*)
REGR_AVGY(*y*, *x*)
where x is a UDT
		Default format for the data type to which the UDT is implicitly cast.

COUNT(*x*)
COUNT(*)
REGR_COUNT(*x* ,*y*)
where the transaction mode is ANSI
	If MaxDecimal in DBSControl is…
	• 0 or 15, then the result type is DECIMAL(15,0) and the format is -(15)9.
	• 18, then the result type is DECIMAL(18,0) and the format is -(18)9.
	• 38, then the result type is DECIMAL(38,0) and the format is -(38)9.
	ANSI transaction mode uses DECIMAL because tables frequently have a cardinality exceeding the range of INTEGER.	

COUNT(*x*)
COUNT(*)
	INTEGER	Default format for INTEGER

Function	Result Type	Format
REGR_COUNT(*x*,*y*) where the transaction mode is Teradata	Teradata transaction mode uses INTEGER to avoid regression problems. **Note:** You can cast the final result of a COUNT window aggregate function; however, the cast is not used as part of the window function computation as it is for the COUNT aggregate function and, therefore, cannot be used to avoid numeric overflow errors that might occur during the computation.	
MAX(*x*), MIN(*x*)	Same data type as operand *x*.	Same format as operand *x*
SUM(*x*) where *x* is a character type	Same as operand *x*.	Default format for FLOAT
SUM(*x*) where *x* is a DECIMAL(*n*,*m*) type	DECIMAL(*p*,*m*), where *p* is determined according to the following rules: If MaxDecimal in DBSControl is 0 or 15 and • *n* ≤ 15, then *p* = 15. • 15 < *n* ≤ 18, *p* = 18. • *n* > 18, then *p* = 38. If MaxDecimal in DBSControl is 18 and • *n* ≤ 18, then *p* = 18. • *n* > 18, then *p* = 38. If MaxDecimal in DBSControl is 38 and *n* = any value, the *p* = 38.	Default format for DECIMAL
SUM(*x*) where *x* is any numeric type other than DECIMAL	Same as operand *x*.	Default format for the data type of the operand

Result Title

The default title that appears in the heading for displayed or printed results depends on the type of computation performed.

IF the type of computation is …	THEN the result title is …
cumulative	Cumulative *Function_name* (*argument_list*) For example, consider the following computation:

IF the type of computation is …	THEN the result title is …

	`SELECT AVG(sales) OVER (PARTITION BY region`
	`   ORDER BY smonth ROWS UNBOUNDED PRECEDING)`
	`FROM sales_history;`
	The title that appears in the result heading is:
	Cumulative Avg(sales)

group	Group *Function_name* (*argument_list*)
	For example, consider the following computation:
	`SELECT AVG(sales) OVER (PARTITION BY region`
	`   ORDER BY smonth ROWS BETWEEN UNBOUNDED`
	`   PRECEDING AND UNBOUNDED FOLLOWING)`
	`FROM sales_history;`
	The title that appears in the result heading is:
	Group Avg(sales)

moving	Moving *Function_name* (*argument_list*)
	For example, consider the following computation:
	`SELECT AVG(sales) OVER (PARTITION BY region`
	`   ORDER BY smonth ROWS 2 PRECEDING)`
	`FROM sales_history;`
	The title that appears in the result heading is:
	Moving Avg(sales)

remaining	Remaining *Function_name* (*argument_list*)
	For example, consider the following computation:
	`SELECT AVG(sales) OVER (PARTITION BY region`
	`   ORDER BY smonth ROWS BETWEEN CURRENT ROW`
	`   AND UNBOUNDED FOLLOWING)`
	`FROM sales_history;`
	The title that appears in the result heading is:
	Remaining Avg(sales)

Problems with Missing Data

Make sure that data you analyze has no missing data points. Computing a moving function over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Nesting Aggregates in Window Functions

Although you can nest aggregates in window functions, including the select list, HAVING, QUALIFY, and ORDER BY clauses, the HAVING clause can only contain aggregate function references because HAVING cannot contain nested syntax like RANK() OVER (ORDER BY SUM(x)).

Aggregate functions cannot be specified with Teradata-specific functions.

Example

The following query nests the SUM aggregate function within the RANK ordered analytical function in the select list:

```
SELECT state, city, SUM(sale),
RANK() OVER (PARTITION BY state ORDER BY SUM(sale))
FROM T1
WHERE T1.cityID = T2.cityID
GROUP BY state, city
HAVING MAX(sale) > 10;
```

Alternative: Using Derived Tables

Although only window functions allow aggregates specified together in the same SELECT list, window functions and Teradata-specific functions can be combined with aggregates using derived tables or views. Using derived tables or views also clarifies the semantics of the computation.

Example

The following example shows the sales rank of a particular product in a store and its percent contribution to the store sales for the top three products in each store.

```
SELECT RT.storeid, RT.prodid, RT.sales,
RT.rank_sales, RT.sales * 100.0/ST.sum_store_sales
FROM (SELECT storeid, prodid, sales, RANK(sales) AS rank_sales
FROM sales_tbl
GROUP BY storeID
QUALIFY RANK(sales) <=3) AS RT,
(SELECT storeID, SUM(sales) AS sum_store_sales
FROM sales_tbl
GROUP BY storeID) AS ST
WHERE RT.storeID = ST.storeID
ORDER BY RT.storeID, RT.sales;
```

The results table might look something like the following.

storeID	prodID	sales	rank_sales	sales*100.0/sum_store_sales
1001	D	35000.00	3	17.949
1001	C	60000.00	2	30.769
1001	A	100000.00	1	51.282
1002	D	25000.00	3	25.000
1002	C	35000.00	2	35.000
1002	A	40000.00	1	40.000
1003	C	20000.00	3	20.000
1003	A	30000.00	2	30.000
1003	D	50000.00	1	50.000
...	...	...	...	

Teradata-Specific Alternatives to Ordered Analytical Functions

Teradata SQL supports two syntax alternatives for ordered analytical functions:

- Teradata-specific
- ANSI SQL:2011 compliant

Window aggregate, rank, distribution, and row number functions are ANSI SQL:2011 compliant. Teradata-specific functions are not.

Teradata-Specific Functions and ANSI SQL:2011 Window Functions

The following table identifies equivalent ANSI SQL:2011 window functions for Teradata-specific functions.

Note:

The use of the Teradata-specific functions listed in the following table is strongly discouraged. These functions are retained only for backward compatibility with existing applications. Be sure to use the ANSI-compliant window functions for any new applications you develop.

Teradata-Specific Functions	Equivalent ANSI SQL:2011 Window Functions
CSUM	SUM
MAVG	AVG
MDIFF(x, w, y)	composable from SUM
MLINREG	composable from SUM and COUNT

Teradata-Specific Functions	Equivalent ANSI SQL:2011 Window Functions
QUANTILE	composable from RANK and COUNT
RANK	RANK
MSUM	SUM

Comparing Window Aggregate Functions and Teradata-Specific Functions

Avoid using Teradata-specific functions such as MAVG, CSUM, and MSUM for applications intended to be ANSI-compliant and portable.

ANSI Function	Teradata Function	Relationship
AVG	MAVG	The form of the AVG window function that specifies an aggregation group of ROWS *value* PRECEDING is the ANSI equivalent of the MAVG Teradata-specific function. Note that the ROWS *value* PRECEDING phrase specifies the number of rows preceding the current row that are used, together with the current row, to compute the moving average. The total number of rows in the aggregation group is *value* + 1. For the MAVG function, the total number of rows in the aggregation group is the value of *width*. For AVG window function, an aggregation group of ROWS 5 PRECEDING, for example, means that the 5 rows preceding the current row, plus the current row, are used to compute the moving average. Thus the moving average for the 6th row of a partition would have considered row 6, plus rows 5, 4, 3, 2, and 1 (that is, 6 rows in all). For the MAVG function, a *width* of 5 means that the current row, plus 4 preceding rows, are used to compute the moving average. The moving average for the 6th row would have considered row 6, plus rows 4, 5, 3, and 2 (that is, 5 rows in all).
SUM	CSUM MSUM	Be sure to use the ANSI-compliant SUM window function for any new applications you develop. Avoid using CSUM and MSUM for applications intended to be ANSI-compliant and portable. The following defines the relationship between the SUM window function and the CSUM and MSUM Teradata-specific functions, respectively: • The SUM window function that uses the ORDER BY clause and specifies ROWS UNBOUNDED PRECEDING is the ANSI equivalent of CSUM. • The SUM window function that uses the ORDER BY clause and specifies ROWS *value* PRECEDING is the ANSI equivalent of MSUM. Note that the ROWS *value* PRECEDING phrase specifies the number of rows preceding the current row that are used, together with the current row, to compute the moving average. The total number of rows in the aggregation group is *value* + 1. For the MSUM function, the total number of rows in the aggregation group is the value of *width*. Thus for the SUM window function that computes a moving sum, an aggregation group of ROWS 5 PRECEDING means that the 5 rows preceding the current row, plus the current row, are used to compute the moving sum. The moving sum for the 6th row of a partition, for example,

ANSI Function	Teradata Function	Relationship
		would have considered row 6, plus rows 5, 4, 3, 2, and 1 (that is, 6 rows in all). For the MSUM function, a *width* of 5 means that the current row, plus 4 preceding rows, are used to compute the moving sum. The moving sum for the 6th row, for example, would have considered row 6, plus rows 5, 4, 3, and 2 (that is, 5 rows in all). Moreover, for data having fewer than *width* rows, MSUM computes the sum using all the preceding rows. MSUM returns the current sum rather than nulls when the number of rows in the sample is fewer than *width*.

Example: Group Count

The following SQL query might yield the results that follow it, where the group count for sales is returned for each of the four partitions defined by city and kind. Notice that rows that have no sales are not counted.

```
SELECT city, kind, sales, profit,
COUNT(sales) OVER (PARTITION BY city, kind
                   ROWS BETWEEN UNBOUNDED PRECEDING AND
                   UNBOUNDED FOLLOWING)
FROM activity_month;

city     kind      sales  profit  Group Count(sales)
-------  --------  -----  ------  ------------------
LA       Canvas       45     320                   4
LA       Canvas      125     190                   4
LA       Canvas      125     400                   4
LA       Canvas       20     120                   4
LA       Leather      20      40                   1
LA       Leather       ?       ?                   1
Seattle  Canvas       15      30                   3
Seattle  Canvas       20      30                   3
Seattle  Canvas       20     100                   3
Seattle  Leather      35      50                   1
Seattle  Leather       ?       ?                   1
```

Example: Remaining Count

To count all the rows, including rows that have no sales, use COUNT(*). Here is an example that counts the number of rows remaining in the partition after the current row:

```
SELECT city, kind, sales, profit,
COUNT(*) OVER (PARTITION BY city, kind ORDER BY profit DESC
               ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
FROM activity_month;

city     kind      sales  profit  Remaining Count(*)
-------  --------  -----  ------  ------------------
LA       Canvas       20     120                   ?
LA       Canvas      125     190                   1
LA       Canvas       45     320                   2
LA       Canvas      125     400                   3
LA       Leather       ?       ?                   ?
LA       Leather      20      40                   1
Seattle  Canvas       15      30                   ?
Seattle  Canvas       20      30                   1
Seattle  Canvas       20     100                   2
Seattle  Leather       ?       ?                   ?
Seattle  Leather      35      50                   1
```

Note that the sort order that you specify in the window specification defines the sort order of the rows over which the function is applied; it does not define the ordering of the results.

In the example, the DESC sort order is specified for the computation, but the results are returned in the reverse order.

To order the results, use the ORDER BY phrase in the SELECT statement:

```
SELECT city, kind, sales, profit,
COUNT(*) OVER (PARTITION BY city, kind ORDER BY profit DESC
               ROWS BETWEEN 1 FOLLOWING AND
               UNBOUNDED FOLLOWING)
FROM activity_month
ORDER BY city, kind, profit DESC;

city     kind      sales  profit  Remaining Count(*)
-------  --------  -----  ------  ------------------
LA       Canvas      125     400                   3
LA       Canvas       45     320                   2
LA       Canvas      125     190                   1
LA       Canvas       20     120                   ?
LA       Leather      20      40                   1
LA       Leather       ?       ?                   ?
Seattle  Canvas       20     100                   2
Seattle  Canvas       20      30                   1
Seattle  Canvas       15      30                   ?
Seattle  Leather      35      50                    1
Seattle  Leather       ?       ?                    ?
```

Example: Cumulative Maximum

The following SQL query might yield the results that follow it, where the cumulative maximum value for sales is returned for each partition defined by city and kind.

```
SELECT city, kind, sales, week,
MAX(sales) OVER (PARTITION BY city, kind
                 ORDER BY week ROWS UNBOUNDED PRECEDING)
FROM activity_month;

city     kind      sales  week  Cumulative Max(sales)
-------  --------  -----  ----  ---------------------
LA       Canvas      263    16                    263
LA       Canvas      294    17                    294
LA       Canvas      321    18                    321
LA       Canvas      274    20                    321
LA       Leather     144    16                    144
LA       Leather     826    17                    826
LA       Leather     489    20                    826
LA       Leather     555    21                    826
Seattle  Canvas      100    16                    100
Seattle  Canvas      182    17                    182
Seattle  Canvas       94    18                    182
Seattle  Leather     933    16                    933
Seattle  Leather     840    17                    933
Seattle  Leather     899    18                    933
Seattle  Leather     915    19                    933
Seattle  Leather     462    20                    933
```

Example: Cumulative Minimum

The following SQL query might yield the results that follow it, where the cumulative minimum value for sales is returned for each partition defined by city and kind.

```
SELECT city, kind, sales, week,
MIN(sales) OVER (PARTITION BY city, kind
                 ORDER BY week
                 ROWS UNBOUNDED PRECEDING)
FROM activity_month;
```

```
city     kind      sales  week  Cumulative Min(sales)
-------  --------  -----  ----  ---------------------
LA       Canvas      263    16                    263
LA       Canvas      294    17                    263
LA       Canvas      321    18                    263
LA       Canvas      274    20                    263
LA       Leather     144    16                    144
LA       Leather     826    17                    144
LA       Leather     489    20                    144
LA       Leather     555    21                    144
Seattle  Canvas      100    16                    100
Seattle  Canvas      182    17                    100
Seattle  Canvas       94    18                     94
Seattle  Leather     933    16                    933
Seattle  Leather     840    17                    840
Seattle  Leather     899    18                    840
Seattle  Leather     915    19                    840
Seattle  Leather     462    20                    462
```

Example: Cumulative Sum

The following query returns the cumulative balance per account ordered by transaction date:

```
SELECT acct_number, trans_date, trans_amount,
SUM(trans_amount) OVER (PARTITION BY acct_number
                        ORDER BY trans_date
                        ROWS UNBOUNDED PRECEDING) as balance
FROM ledger
ORDER BY acct_number, trans_date;
```

Here are the possible results of the preceding SELECT.

acct_number	trans_date	trans_amount	balance
73829	1998-11-01	113.45	113.45
73829	1988-11-05	-52.01	61.44
73929	1998-11-13	36.25	97.69
82930	1998-11-01	10.56	10.56
82930	1998-11-21	32.55	43.11
82930	1998-11-29	-5.02	38.09

Example: Group Sum

The query below finds the total sum of meat sales for each city.

```
SELECT city, kind, sales,
SUM(sales) OVER (PARTITION BY city ROWS BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING) FROM monthly;
```

The possible results of the preceding SELECT appear in the following table.

city	kind	sales	Group Sum (sales)
Omaha	pure pork	45	220
Omaha	pure pork	125	220
Omaha	pure pork	25	220
Omaha	variety pack	25	220
Chicago	variety pack	55	175
Chicago	variety pack	45	175
Chicago	pure pork	50	175
Chicago	variety pack	25	175

Example: Group Sum

The following query returns the total sum of meat sales for all cities. Note there is no PARTITION BY clause in the SUM function, so all cities are included in the group sum.

```
SELECT city, kind, sales,
SUM(sales) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND
                 UNBOUNDED FOLLOWING)
FROM monthly;
```

The possible results of the preceding SELECT appear in the table below.

city	kind	sales	Group Sum (sales)
Omaha	pure pork	45	395
Omaha	pure pork	125	395
Omaha	pure pork	25	395

city	kind	sales	Group Sum (sales)
Omaha	variety pack	25	395
Chicago	variety pack	55	395
Chicago	variety pack	45	395
Chicago	pure pork	50	395
Chicago	variety pack	25	395

Example: Moving Sum

The following query returns the moving sum of meat sales by city. Notice that the query returns the moving sum of sales by city (the partition) for the current row (of the partition) and three preceding rows where possible.

The order in which each meat variety is returned is the default ascending order according to profit.

Where no sales figures are available, no moving sum of sales is possible. In this case, there is a null in the sum(sales) column.

```
SELECT city, kind, sales, profit,
SUM(sales) OVER (PARTITION BY city, kind
                 ORDER BY profit ROWS 3 PRECEDING)
FROM monthly;
```

city	kind	sales	profit	Moving sum (sales)
Omaha	pure pork	25	40	25
Omaha	pure pork	25	120	50
Omaha	pure pork	45	140	95
Omaha	pure pork	125	190	220
Omaha	pure pork	45	320	240
Omaha	pure pork	1255	400	340
Omaha	variety pack	?	?	?
Omaha	variety pack	25	40	25
Omaha	variety pack	25	120	50
Chicago	pure pork	?	?	?
Chicago	pure pork	15	10	15

city	kind	sales	profit	Moving sum (sales)
Chicago	pure pork	54	12	69
Chicago	pure pork	14	20	83
Chicago	pure pork	54	24	137
Chicago	pure pork	14	34	136
Chicago	pure pork	95	80	177
Chicago	pure pork	95	140	258
Chicago	pure pork	15	220	219
Chicago	variety pack	23	39	23
Chicago	variety pack	25	40	48
Chicago	variety pack	125	70	173
Chicago	variety pack	125	100	298
Chicago	variety pack	23	100	298
Chicago	variety pack	25	120	298

Example: Remaining Sum

The following query returns the remaining sum of meat sales for all cities. Note there is no PARTITION BY clause in the SUM function, so all cities are included in the remaining sum.

```
SELECT city, kind, sales,
SUM(sales) OVER (ORDER BY city, kind
                 ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
FROM monthly;
```

The possible results of the preceding SELECT appear in the table below.

```
city     kind           sales    Remaining Sum(sales)
-------  -------------  -------  --------------------
Omaha    variety pack   25       ?
Omaha    pure pork      125      25
Omaha    pure pork      25       150
Omaha    pure pork      45       175
Chicago  variety pack   55       220
Chicago  variety pack   25       275
Chicago  variety pack   45       300
Chicago  pure pork      50       345
```

Note that the sort order for the computation is alphabetical by city, and then by kind. The results, however, appear in the reverse order.

The sort order that you specify in the window specification defines the sort order of the rows over which the function is applied; it does not define the ordering of the results. To order the results, use an ORDER BY phrase in the SELECT statement.

For example:

```
    SELECT city, kind, sales,
    SUM(sales) OVER (ORDER BY city, kind
                     ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
    FROM monthly
    ORDER BY city, kind;
```

The possible results of the preceding SELECT appear in the table below:

```
city     kind           sales    Remaining Sum(sales)
-------  -------------  -------  --------------------
Chicago  pure pork      50       345
Chicago  variety pack   55       265
Chicago  variety pack   25       320
Chicago  variety pack   45       220
Omaha    pure pork      25       70
Omaha    pure pork      125      95
Omaha    pure pork      45       25
Omaha    variety pack   25       ?
```

IF you want to compute the …	THEN use this function …
cumulative sum	• SUM window function • CSUM
cumulative, group, or moving count	COUNT window function
group sum	SUM window function
moving average	• AVG window function • MAVG
moving difference between the current row-column value and the preceding *n* th row-column value	MDIFF
moving linear regression	MLINREG

IF you want to compute the …	THEN use this function …
moving sum	• SUM window function • MSUM
quantile scores for the values in a column	QUANTILE
ordered rank of all rows in a group	• RANK window function • RANK
relative rank of a row in a group	PERCENT_RANK window function
sequential row number of the row within its window partition according to the window ordering of the window	ROW_NUMBER
cumulative, group, or moving maximum value	MAX window function
cumulative, group, or moving minimum value	MIN window function

GROUP BY Clause

GROUP BY and Window Functions

For window functions, the GROUP BY clause must include all the columns specified in the:

- Select list of the SELECT clause
- Window functions in the select list of a SELECT clause
- Window functions in the search condition of a QUALIFY clause
- The condition in the RESET WHEN clause

For example, the following SELECT statement specifies the column City in the select list and the column StoreID in the COUNT window function in the select list and QUALIFY clause. Both columns must also appear in the GROUP BY clause:

```
SELECT City, StoreID, COUNT(StoreID) OVER ()
FROM sales_tbl
GROUP BY City, StoreID
QUALIFY COUNT(StoreID) >=3;
```

For window functions, GROUP BY collapses all rows with the same value for the group-by columns into a single row.

For example, the following statement uses the GROUP BY clause to collapse all rows with the same value for City and StoreID into a single row:

```
SELECT City, StoreID, COUNT(StoreID) OVER ()
FROM sales_tbl
GROUP BY City, StoreID;
```

The results look like this:

```
City   StoreID  Group Count(StoreID)
-----  -------  --------------------
Pecos     1001                     3
Pecos     1002                     3
Ozona     1003                     3
```

Without the GROUP BY, the results look like this:

```
City   StoreID  Group Count(StoreID)
-----  -------  --------------------
Pecos     1001                     9
Pecos     1001                     9
Pecos     1001                     9
Pecos     1001                     9
Pecos     1002                     9
Pecos     1002                     9
Pecos     1002                     9
Ozona     1003                     9
Ozona     1003                     9
```

GROUP BY and Teradata-Specific Functions

For Teradata-specific functions, GROUP BY determines the partitions over which the function executes. The clause does not collapse all rows with the same value for the group-by columns into a single row. Thus, the GROUP BY clause in these cases need only specify the partitioning column for the function.

For example, the following statement computes the running sales for each store by using the GROUP BY clause to partition the data in sales_tbl by StoreID:

```
SELECT StoreID, Sales, CSUM(Sales, StoreID)
FROM sales_tbl
GROUP BY StoreID;
```

The results look like this:

```
StoreID     Sales  CSum(Sales,StoreID)
-------  --------  -------------------
```

```
      1001   1100.00             1100.00
      1001    400.00             1500.00
      1001   1000.00             2500.00
      1001   2000.00             4500.00
      1002    500.00              500.00
      1002   1500.00             2000.00
      1002   2500.00             4500.00
      1003   1000.00             1000.00
      1003   3000.00             4000.00
```

Combining Window Functions, Teradata-Specific Functions, and GROUP BY

The following table provides the semantics of the allowable combinations of window functions, Teradata-specific functions, aggregate functions, and the GROUP BY clause.

Combination				Semantics
Window Function	**Teradata-Specific Function**	**Aggregate Function**	**GROUP BY Clause**	
X				A value is computed for each row.
	X			A value is computed for each row. The entire table constitutes a single group, or partition, over which the Teradata-specific function executes.
		X		One aggregate value is computed for the entire table.
X			X	GROUP BY collapses all rows with the same value for the group-by columns into a single row, and a value is computed for each resulting row.
	X		X	GROUP BY determines the partitions over which the Teradata-specific function executes. The clause does not collapse all rows with the same value for the group-by columns into a single row.
		X	X	An aggregation is performed for each group.
X	X			Teradata-specific functions do not have partitions. The whole table is one partition.
X	X		X	GROUP BY determines partitions for Teradata-specific functions. GROUP BY does not collapse all rows with the same value for

Combination				Semantics
Window Function	Teradata-Specific Function	Aggregate Function	GROUP BY Clause	
				the group-by columns into a single row, and does not affect window function computation.
X		X	X	GROUP BY collapses all rows with the same value for the group-by columns into a single row. For window functions, a value is computed for each resulting row; for aggregate functions, an aggregation is performed for each group.

Possible Result Overflow with SELECT Sum

When using this function, the result can create an overflow when the data type and format are not in sync. For a column defined as:

```
Salary Decimal(15,2) Format '$ZZZ,ZZ9.99'
```

The following query:

```
SELECT SUM (Salary) FROM Employee;
```

causes an overflow because the decimal operand and the format are not in sync.

To avoid possible overflows, explicitly specify the format for decimal sum to specify a format large enough to accommodate the decimal sum resultant data type.

```
SELECT Sum(Salary) (format '$Z,ZZZ,ZZZ,ZZ9.99) FROM Employee;
```

Related Information

- For descriptions of aggregate functions and arguments, see Aggregate Functions.
- For more information, see “RESET WHEN Condition Rules” and “QUALIFY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*.

CSUM

Returns the cumulative (or running) sum of a value expression for each row in a partition, assuming the rows in the partition are sorted by the sort_expression list.

CSUM accumulates a sum over an ordered set of rows, providing the current value of the SUM on each row.

Type

Teradata-specific function.

CSUM Function Syntax

```
CSUM ( value_expression, sort_spec [,...] )
```

Syntax Elements

sort_spec

```
sort_expression [ ASC | DESC ]
```

value_expression

A numeric literal or column expression for which a running sum is to be computed.

By default, CSUM uses the default data type of value_expression. Larger numeric values are supported by casting it to a higher data type.

The expression cannot contain any ordered analytical or aggregate functions.

sort_expression

A literal or column expression or comma-separated list of literal or column expressions to be used to sort the values.

The expression cannot contain any ordered analytical or aggregate functions.

ASC

Ascending sort order.

DESC

Descending sort order.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for CSUM are as follows:

Data Type: Same as operand x

- If operand x is character, the format is the default format for FLOAT.
- If operand x is numeric, the format is the same format as x.

CSUM Usage Notes

Using SUM Instead of CSUM

The use of CSUM is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is equivalent to the ANSI-compliant SUM window function that specifies ROWS UNBOUNDED PRECEDING as its aggregation group. CSUM is retained only for backward compatibility with existing applications.

Possible Result Overflow with SELECT Sum

Possible Result Overflow with SELECT Sum

When using this function, the result can create an overflow when the data type and format are not in sync. For a column defined as:

```
Salary Decimal(15,2) Format ‘$ZZZ,ZZ9.99’
```

The following query:

```
SELECT SUM (Salary) FROM Employee;
```

causes an overflow because the decimal operand and the format are not in sync.

To avoid possible overflows, explicitly specify the format for decimal sum to specify a format large enough to accommodate the decimal sum resultant data type.

```
SELECT Sum(Salary) (format ‘$Z,ZZZ,ZZZ,ZZ9.99) FROM Employee;
```

Examples

Example

Report the daily running sales total for product code 10 for each month of 1998.

```
SELECT cmonth, CSUM(sumPrice, cdate)
FROM
(SELECT a2.month_of_year,
a2.calendar_date,a1.itemID, SUM(a1.price)
FROM Sales a1, SYS_CALENDAR.Calendar a2
WHERE a1.calendar_date=a2.calendar_date
AND a2.calendar_date=1998
AND a1.itemID=10
GROUP BY a2.month_of_year, a1.calendar_date,
a1.itemID) AS T1(cmonth, cdate, sumPrice)
GROUP BY cmonth;
```

Grouping by month allows the total to accumulate until the end of each month, when it is then set to zero for the next month. This permits the calculation of cumulative totals for each item in the same query.

Example

Provide a running total for sales of each item in store 5 in January and generate output that is ready to export into a graphing program.

```
SELECT Item, SalesDate, CSUM(Revenue,Item,SalesDate) AS CumulativeSales
FROM
(SELECT Item, SalesDate, SUM(Sales) AS Revenue
FROM DailySales
WHERE StoreId=5 AND SalesDate BETWEEN
'1/1/1999' AND '1/31/1999'
GROUP BY Item, SalesDate) AS ItemSales
ORDER BY SalesDate;
```

The result might like something like the following table.

Item	SalesDate	CumulativeSales
InstaWoof dog food	01/01/1999	972.99
InstaWoof dog food	01/02/1999	2361.99
InstaWoof dog food	01/03/1999	5110.97
InstaWoof dog food	01/04/1999	7793.91

CUME_DIST

Calculates the cumulative distribution of a value in a group of values.

CUME_DIST is similar to PERCENT_RANK. Unlike PERCENT_RANK, which considers the RANK value in the presence of ties, CUME_DIST uses the highest tied rank, that is, the position of the last tied value when there are peers. CUME_DIST is the ratio of that position in the partition (RANK-HIGH/NUM ROWS).

Type

ANSI SQL:2011 window function.

CUME_DIST Function Syntax

```
CUME_DIST() OVER (
  [ PARTITION BY column_reference [,...] ]
  ORDER BY value_spec [,...]
  [ RESET WHEN condition ]
)
```

Syntax Elements

OVER

Specifies how values are grouped, ordered, and considered when computing the cumulative, group, or moving function.

Values are grouped according to the PARTITION BY BEGIN and RESET WHEN clauses END, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.

PARTITION BY *column_reference* [,...]

The group or groups over which the function operates.

If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a partition.

PARTITION BY clause is also called the window partition clause.

ORDER BY *value_spec* [,...]

```
value_expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

The order in which the values in a group or partition are sorted.

RESET WHEN *condition*

A conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

For more information, see “RESET WHEN Condition Rules” and the “QUALIFY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

ASC

That the results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

DESC

That the results are to be ordered in descending sort order.

If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

NULLS FIRST

NULL results are to be listed first.

NULLS LAST

NULL results are to be listed last.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Results

The range of values returned by CUME_DIST is >0 to <=1.

Example

The following SELECT statement:

```
SELECT lname, serviceyrs,
 CUME_DIST()OVER(ORDER BY serviceyrs)
 FROM schooltbl
 GROUP BY 1,2;
```

returns the cumulative distribution by service years for teachers listed in *schooltbl*.

lname	serviceyrs	CUME_DIST
Adams	10	0.333333
Peters	10	0.333333

lname	serviceyrs	CUME_DIST
Murray	10	0.333333
Rogers	15	0.444333
Franklin	16	0.555333
Smith	20	0.888889
Ford	20	0.888889
Derby	20	0.888889
Baker	20	1.000000

DENSE_RANK (ANSI)

Returns an ordered ranking of rows based on the *value_expression* in the ORDER BY clause.

The ranks are consecutive integers beginning with 1. Rows with equal values receive the same rank. Rank values are not skipped in the event of ties.

Type

ANSI SQL:2011 window function.

DENSE_RANK Function Syntax (ANSI)

```
DENSE_RANK() OVER (
  [ PARTITION BY column_reference [,...] ]
  ORDER BY value_spec [,...]
  [ RESET WHEN condition ]
)
```

Syntax Elements

OVER

Specifies how values are grouped, ordered, and considered when computing the cumulative, group, or moving function.

Values are grouped according to the PARTITION BY BEGIN and RESET WHEN clauses END, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.

PARTITION BY *column_reference* [,...]

The group or groups over which the function operates.

If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a partition.

PARTITION BY clause is also called the window partition clause.

ORDER BY *value_spec* [,...]

```
value_expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

The order in which the values in a group or partition are sorted.

RESET WHEN *condition*

The group, or groups, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition.

If there are no RESET WHEN or PARTITION BY clauses, then the entire result set constitutes a single partition.

A conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

For more information, see “RESET WHEN Condition Rules” and the “QUALIFY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

ASC

That the results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

DESC

That the results are to be ordered in descending sort order.

If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

NULLS FIRST

NULL results are to be listed first.

NULLS LAST

NULL results are to be listed last.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is INTEGER.

Example

The following SELECT statement:

```
SELECT lname, serviceyrs,
 DENSE_RANK()OVER(ORDER BY serviceyrs)
 FROM schooltbl
 GROUP BY 1,2;
```

returns the ordered ranking by service years for teachers listed in *schooltbl*.

lname	serviceyrs	DENSE_RANK
Adams	10	1
Peters	10	1
Murray	10	1
Rogers	15	2
Franklin	16	3
Smith	20	4
Ford	20	4
Derby	20	4
Baker	25	5

FIRST_VALUE/LAST_VALUE

Returns the first value or last value in an ordered set of values.

Type

ANSI SQL:2011 window function.

FIRST_VALUE/LAST_VALUE Function Syntax

```
{ FIRST_VALUE | LAST_VALUE } (
  value_expression [ { IGNORE | RESPECT } NULLS ]
) window
```

Syntax Elements

value_expression

A column expression.

FIRST_VALUE and LAST_VALUE use the default data type of *value_expression*.

Larger numeric values are supported by casting them to a higher data type.

The expression cannot contain any ordered analytical or aggregate functions.

IGNORE NULLS

Keyword that specifies not to return NULL.

- IGNORE NULLS (with FIRST_VALUE) = returns the first non-null value in the set, or NULL if all values are NULL.
- If IGNORE NULLS (with LAST_VALUE) = returns the last non-null value in the set, or NULL if all values are NULL.

RESPECT NULLS

Optional keyword that specifies whether to return NULL.

- RESPECT NULLS (with FIRST_VALUE) = returns the first value, whether or not it is null.
- RESPECT NULLS (with LAST_VALUE) = returns the last value, whether or not it is null.

If all values are null, NULL is returned.

window

A group, cumulative, or moving computation.

See The Window Specification.

In presence of ties in the sort key of the Window Aggregate Function syntax, FIRST_VALUE and LAST_VALUE are non-deterministic. They return *value_expression* from any one of the rows with tied order by value.

Note:

If the ROWS phrase is omitted and there is an ORDER BY phrase, the default ROWS is UNBOUNDED PRECEDING AND CURRENT ROW.

If the ROWS phrase is omitted and there is no ORDER BY phrase, the default ROWS is UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

FIRST_VALUE and LAST_VALUE are especially valuable because they are often used as the baselines in calculations. For instance, with a partition holding sales data ordered by day, you may want to know how much the sales for each day were compared to the first sales day (FIRST_VALUE) for the period, or you may want to know, for a set of rows in increasing sales order, what the percentage size of each sale in the region was compared to the largest sale (LAST_VALUE) in the region.

IGNORE NULLS is particularly useful in populating an inventory table properly.

Selecting neither IGNORE NULLS or RESPECT NULLS is equivalent to selecting RESPECT NULLS.

Example

The following example returns by start date the salary, moving average (ma), and first and last salary in the moving average group.

Note:

The functions are going to return the first/last value in the window. In the example, the first and last rows fall within the window. If the window were between 3 preceding and 2 preceding rows, you would see NULL for first value in the 1st two rows.

```
SELECT start_date, salary,
       AVG(salary) OVER(ORDER BY start_date
       ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING) ma,
       FIRST_VALUE(salary) OVER(ORDER BY start_date
       ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING) first,
       LAST_VALUE(salary) OVER(ORDER BY start_date
       ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING) last
FROM employee
ORDER BY start_date;
```

start_date	salary	ma	first	last
21-MAR-76	6661.78	6603.280	6661.78	6544.78
12-DEC-78	6544.78	5183.780	6661.78	2344.78
24-OCT-82	2344.78	4471.530	6661.78	2344.78
15-JAN-84	2334.78	4441.780	6661.78	4322.78
30-JUL-87	4322.980	4688.980	6544.78	7897.78
31-DEC-90	7897.78	3626.936	2344.78	1234.56
25-JUL-96	1234.56	3404.536	2334.78	1232.78
17-SEP-96	1232.78	3671.975	4322.78	1232.78

LAG/LEAD

Ordered analytic functions calculate an aggregate or non-aggregate value on a window of rows within a group of rows. The window of rows is defined by the Window Framing clause, also called the ROWS clause. Window sizes are based on the size specified in the ROWS clause. The group of rows is defined by the PARTITION BY clause of the Window function.

The LAG function accesses data from the row preceding the current row at a specified offset value in a window group, while the LEAD function returns data from the row following the current row. If the offset value is outside the scope of the window, the user-specified default value is returned.

The LAG and LEAD functions are used for OLAP and decision support queries.

LAG/LEAD Function Syntax

ANSI

```
{ LAG | LEAD } ( value_expression [, offset_spec ] ] )
  [ { RESPECT | IGNORE } NULLS ]
  OVER ( [ PARTITION BY expression ] [ order_by_clause ] )
```

offset_spec

```
[ offset_value ] [, [ default_value_expression ] ]
```

order_by_clause

```
ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
  [ RESET WHEN expression ]
```

Teradata

```
{ LAG | LEAD }
  ( value_expression [ { IGNORE | RESPECT } NULLS ] [, offset_spec ] ] )
  OVER ( [ PARTITION BY expression ] [ order_by_clause ] )
```

offset_spec

```
[ offset_value ] [, [ default_value_expression ] ]
```

order_by_clause

```
ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
  [ RESET WHEN condition ]
```

Syntax Elements

value_expression

The expression cannot contain any ordered analytical functions.

value_expression is mandatory and can be any expression that returns a scalar value. It cannot be a table function.

offset_value

A literal unsigned integer value between 0 and 4096. If not specified, the default value is 1.

offset_value specifies the physical row position relative to the current row in a given window of rows. The row position is the row following the current row for the LEAD function, and the preceding row for the LAG function.

An *offset_value* of 0 specifies the current row.

default_value_expression

Any expression that returns a scalar value.

If not specified, the value is assumed to be NULL.

When running in ANSI mode, the *default_value_expression* data type must match *value_expression*. An error occurs if the data types do not match.

In Teradata mode, the database attempts to match the *default_value_expression* data type to *value_expression* by doing a cast to *value_expression* data type to execute the query. If there are casting rule violations, Vantage displays an error message.

IGNORE NULLS

If *value_expression* returns a NULL value where the preceding or following row, as determined by the specified *offset_value*, is within the scope of the window group, LAG or LEAD ignores the NULL value.

LAG or LEAD then continues searching for the non-NULL *value_expression* in the preceding or following row, which may be far from the current row but within the scope of the window group. The search terminates at the window boundaries:

- For LAG, the search terminates at the first row of the window group.
- For LEAD, the search terminates at the last row of the window group.

At the end of the search, LAG or LEAD returns *default_value_expression* if no non-NULL *value_expression* is found.

If the preceding or following row is outside the scope of the window group, LAG or LEAD returns *default_value_expression*.

If the optional NULL clause is not specified, the default option is RESPECT NULLS.

RESPECT NULLS

If the preceding or following row determined by *offset_value* is within the scope of the window group, and if the *value_expression* evaluation returns a NULL, LAG or LEAD returns NULL. This setting indicates that the NULL value is not ignored.

If the preceding or following row is outside the scope of the window group, LAG or LEAD returns *default_value_expression*.

If the optional NULL clause is not specified, the default option is RESPECT NULLS.

OVER

Specifies how values are grouped, ordered, and considered while computing the LAG or LEAD function.

Values are grouped by the optional PARTITION BY clause and the optional RESET WHEN clause. Values are sorted according to the ORDER BY clause in a given partition of rows.

PARTITION BY *expression*

The group or groups over which the function operates.

This is a comma-separated value expression list.

ORDER BY *expression*

The order in which the values in a group or partition are sorted.

This is a comma-separated value expression list.

ASC

That the results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

DESC

Descending sort order.

NULLS FIRST

NULL results are to be listed first.

NULLS LAST

NULL results are to be listed last.

RESET WHEN *condition*

The group, or groups, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition.

If there are no RESET WHEN or PARTITION BY clauses, then the entire result set constitutes a single partition.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type

The data type of the LEAD or LAG function's returned values is the same as the specified value of *value_expression*. If *default_value_expression* and *value_expression* have different data types, Teradata recommends explicitly casting *default_value_expression* to the data type of *value_expression*.

In ANSI mode, an error occurs if *default_value_expression* and *value_expression* data types do not match.

In the Teradata Transaction (BTET) mode, if the data types do not match, the database attempts to cast the *default_value_expression* to the *value_expression* data type based on the internal casting rules. If this results in casting rule violations, an error message displays.

Usage Notes

Because the LEAD and LAG functions do not support the ROWS clause in the syntax, the window size is the same as the size of the group of rows defined by the PARTITION BY clause. If the PARTITION BY clause is absent, the entire table becomes a single group, and the size of the group of rows is the same as the total number of rows in the table.

The RESET WHEN clause, which is applicable to all window functions in Vantage, is extended to the LEAD and LAG functions.

The RESET WHEN clause is a Teradata Extension to ANSI. The LEAD and LAG functions support performance-driven rewrites, and support both Teradata syntax and ANSI syntax to simplify data migration from other databases.

In ANSI Transaction mode, the *value_expression* data type must match the default value expression data type, or else an error occurs.

Examples

Example: LAG with IGNORE NULLS

ANSI style syntax:

```
SELECT empno, empname, job, sal,
       LAG(sal, 1, 0) IGNORE NULLS
       OVER (PARTITION BY job ORDER BY empno) AS sal_prev
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME      JOB       SAL      SAL_PREV
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          0
   13        GRACE      ANALYST    3000       0
   1         JOHN       CLERK      800        0
   2         ERIC       CLERK      950        800
   3         KURT       CLERK      ?          950
   6         JULIE      CLERK      1300       950
   9         NICHOLAS   MANAGER    2450       0
   10        NOVAK      MANAGER    ?          2450
   11        ROGER      MANAGER    2850       2450
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       0
   5         LYNN       SALESMAN   ?          1250
   7         TERESA     SALESMAN   1500       1250
   8         MATTHEW    SALESMAN   1600       1500
```

Teradata style syntax:

```
SELECT empno, empname, job, sal,
       LAG(sal IGNORE NULLS, 1, 0)
       OVER (PARTITION BY job ORDER BY empno) AS sal_prev
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME      JOB       SAL      SAL_PREV
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          0
   13        GRACE      ANALYST    3000       0
   1         JOHN       CLERK      800        0
   2         ERIC       CLERK      950        800
   3         KURT       CLERK      ?          950
   6         JULIE      CLERK      1300       950
   9         NICHOLAS   MANAGER    2450       0
   10        NOVAK      MANAGER    ?          2450
   11        ROGER      MANAGER    2850       2450
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       0
   5         LYNN       SALESMAN   ?          1250
   7         TERESA     SALESMAN   1500       1250
   8         MATTHEW    SALESMAN   1600       1500
```

Example: LAG with RESPECT NULLS

ANSI style syntax:

```
SELECT empno, empname, job, sal,
       LAG(sal, 1, 0) RESPECT NULLS
       OVER (PARTITION BY job ORDER BY empno) AS sal_prev
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME     JOB        SAL      SAL_PREV
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          0
   13        GRACE      ANALYST    3000       ?
   1         JOHN       CLERK      800        0
   2         ERIC       CLERK      950        800
   3         KURT       CLERK      ?          950
   6         JULIE      CLERK      1300       ?
   9         NICHOLAS   MANAGER    2450       0
   10        NOVAK      MANAGER    ?          2450
   11        ROGER      MANAGER    2850       ?
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       0
   5         LYNN       SALESMAN   ?          1250
   7         TERESA     SALESMAN   1500       ?
   8         MATTHEW    SALESMAN   1600       1500
```

Teradata style syntax:

```
SELECT empno, empname, job, sal,
       LAG(sal RESPECT NULLS, 1, 0)
       OVER (PARTITION BY job ORDER BY empno) AS sal_prev
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME     JOB        SAL      SAL_PREV
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          0
   13        GRACE      ANALYST    3000       ?
   1         JOHN       CLERK      800        0
   2         ERIC       CLERK      950        800
   3         KURT       CLERK      ?          950
   6         JULIE      CLERK      1300       ?
   9         NICHOLAS   MANAGER    2450       0
   10        NOVAK      MANAGER    ?          2450
   11        ROGER      MANAGER    2850       ?
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       0
   5         LYNN       SALESMAN   ?          1250
   7         TERESA     SALESMAN   1500       ?
   8         MATTHEW    SALESMAN   1600       1500
```

Example: LAG with RESPECT NULLS without Explicitly Specifying RESPECT NULLS

ANSI style syntax:

```
SELECT empno, empname, job, sal,
       LAG (sal, 1, 0)
       OVER (PARTITION BY job ORDER BY empno) AS sal_prev
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME      JOB       SAL      SAL_PREV
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          0
   13        GRACE      ANALYST    3000       ?
   1         JOHN       CLERK      800        0
   2         ERIC       CLERK      950        800
   3         KURT       CLERK      ?          950
   6         JULIE      CLERK      1300       ?
   9         NICHOLAS   MANAGER    2450       0
   10        NOVAK      MANAGER    ?          2450
   11        ROGER      MANAGER    2850       ?
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       0
   5         LYNN       SALESMAN   ?          1250
   7         TERESA     SALESMAN   1500       ?
   8         MATTHEW    SALESMAN   1600       1500
```

Teradata style syntax:

```
SELECT empno, empname, job, sal,
       LAG (sal, 1, 0)
       OVER (PARTITION BY job ORDER BY empno) AS sal_prev
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME      JOB       SAL      SAL_PREV
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          0
   13        GRACE      ANALYST    3000       ?
   1         JOHN       CLERK      800        0
   2         ERIC       CLERK      950        800
   3         KURT       CLERK      ?          950
   6         JULIE      CLERK      1300       ?
   9         NICHOLAS   MANAGER    2450       0
   10        NOVAK      MANAGER    ?          2450
   11        ROGER      MANAGER    2850       ?
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       0
   5         LYNN       SALESMAN   ?          1250
   7         TERESA     SALESMAN   1500       ?
   8         MATTHEW    SALESMAN   1600       1500
```

Example: LEAD with RESPECT NULLS

ANSI style syntax:

```
SELECT empno, empname, job, sal,
       LEAD(sal, 1, 0) RESPECT NULLS
       OVER (PARTITION BY job ORDER BY empno) AS sal_next
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME     JOB        SAL      SAL_NEXT
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          3000
   13        GRACE      ANALYST    3000       0
   1         JOHN       CLERK      800        950
   2         ERIC       CLERK      950        ?
   3         KURT       CLERK      ?          1300
   6         JULIE      CLERK      1300       0
   9         NICHOLAS   MANAGER    2450       ?
   10        NOVAK      MANAGER    ?          2850
   11        ROGER      MANAGER    2850       0
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       ?
   5         LYNN       SALESMAN   ?          1500
   7         TERESA     SALESMAN   1500       1600
   8         MATTHEW    SALESMAN   1600       0
```

Teradata style syntax:

```
SELECT empno, empname, job, sal,
       LEAD(sal RESPECT NULLS, 1, 0)
       OVER (PARTITION BY job ORDER BY empno) AS sal_next
FROM   emp
```

```
ORDER BY job, empno;

 EMPNO      EMPNAME     JOB        SAL      SAL_NEXT
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          3000
   13        GRACE      ANALYST    3000       0
   1         JOHN       CLERK      800        950
   2         ERIC       CLERK      950        ?
   3         KURT       CLERK      ?          1300
   6         JULIE      CLERK      1300       0
   9         NICHOLAS   MANAGER    2450       ?
   10        NOVAK      MANAGER    ?          2850
   11        ROGER      MANAGER    2850       0
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       ?
   5         LYNN       SALESMAN   ?          1500
   7         TERESA     SALESMAN   1500       1600
   8         MATTHEW    SALESMAN   1600       0
```

Example: LEAD with IGNORE NULLS

ANSI style syntax:

```
SELECT empno, empname, job, sal,
       LEAD(sal, 1, 0) IGNORE NULLS
       OVER (PARTITION BY job ORDER BY empno) AS sal_next
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME      JOB       SAL      SAL_NEXT
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          3000
   13        GRACE      ANALYST    3000       0
   1         JOHN       CLERK      800        950
   2         ERIC       CLERK      950        1300
   3         KURT       CLERK      ?          1300
   6         JULIE      CLERK      1300       0
   9         NICHOLAS   MANAGER    2450       2850
   10        NOVAK      MANAGER    ?          2850
   11        ROGER      MANAGER    2850       0
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       1500
   5         LYNN       SALESMAN   ?          1500
   7         TERESA     SALESMAN   1500       1600
   8         MATTHEW    SALESMAN   1600       0
```

Teradata style syntax:

```
SELECT empno, empname, job, sal,
       LEAD(sal IGNORE NULLS, 1, 0)
       OVER (PARTITION BY job ORDER BY empno) AS sal_next
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME      JOB       SAL      SAL_NEXT
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          3000
   13        GRACE      ANALYST    3000       0
   1         JOHN       CLERK      800        950
   2         ERIC       CLERK      950        1300
   3         KURT       CLERK      ?          1300
   6         JULIE      CLERK      1300       0
   9         NICHOLAS   MANAGER    2450       2850
   10        NOVAK      MANAGER    ?          2850
   11        ROGER      MANAGER    2850       0
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       1500
   5         LYNN       SALESMAN   ?          1500
   7         TERESA     SALESMAN   1500       1600
   8         MATTHEW    SALESMAN   1600       0
```

Example: LEAD with RESPECT NULLS without Explicitly Specifying RESPECT NULLS

ANSI style syntax:

```
SELECT empno, empname, job, sal,
       LEAD (sal, 1, 0)
       OVER (PARTITION BY job ORDER BY empno) AS sal_next

FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME      JOB       SAL      SAL_NEXT
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          3000
   13        GRACE      ANALYST    3000       0
   1         JOHN       CLERK      800        950
   2         ERIC       CLERK      950        ?
   3         KURT       CLERK      ?          1300
   6         JULIE      CLERK      1300       0
   9         NICHOLAS   MANAGER    2450       ?
   10        NOVAK      MANAGER    ?          2850
   11        ROGER      MANAGER    2850       0
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       ?
   5         LYNN       SALESMAN   ?          1500
   7         TERESA     SALESMAN   1500       1600
   8         MATTHEW    SALESMAN   1600       0
```

Teradata style syntax:

```
SELECT empno, empname, job, sal,
       LEAD (sal, 1, 0)
       OVER (PARTITION BY job ORDER BY empno) AS sal_next
FROM   emp
ORDER BY job, empno;

 EMPNO      EMPNAME      JOB       SAL      SAL_NEXT
---------- ---------- --------- ---------- ----------
   12        PAUL       ANALYST    ?          3000
   13        GRACE      ANALYST    3000       0
   1         JOHN       CLERK      800        950
   2         ERIC       CLERK      950        ?
   3         KURT       CLERK      ?          1300
   6         JULIE      CLERK      1300       0
   9         NICHOLAS   MANAGER    2450       ?
   10        NOVAK      MANAGER    ?          2850
   11        ROGER      MANAGER    2850       0
   14        RICH       PRESIDENT  5000       0
   4         KENT       SALESMAN   1250       ?
   5         LYNN       SALESMAN   ?          1500
   7         TERESA     SALESMAN   1500       1600
   8         MATTHEW    SALESMAN   1600       0
```

MAVG

Computes the moving average of a value expression for each row in a partition using the specified value expression for the current row and the preceding *width*-1 rows.

Type

Teradata-specific function.

MAVG Function Syntax

```
MAVG ( value_expression, width, sort_spec [,...] )
```

Syntax Elements

value_expression

The expression cannot contain any ordered analytical or aggregate functions.

width

The number of previous rows to be used in the computation.

The value is always a positive integer literal.

The maximum is 4096.

sort_spec

```
sort_expression [ ASC | DESC ]
```

sort_expression

A literal or column expression or comma-separated list of literal or column expressions to be used to sort the values.

For example, MAVG(Sale, 6, Region ASC, Store DESC), where Sale is the *value_expression,* 6 is the *width*, and Region ASC, Store DESC is the *sort_expression* list.

The expression cannot contain any ordered analytical or aggregate functions.

ASC

That the results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

DESC

That the results are to be ordered in descending sort order.

If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for MAVG are as follows:

Data Type: Same as operand x

- If operand x is character, the format is the default format for FLOAT.
- If operand x is numeric, date, or interval, the format is the same format as x.

MAVG Usage Notes

Using AVG Instead of MAVG

The use of MAVG is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is equivalent to the ANSI-compliant AVG window function that specifies ROWS *value* PRECEDING as its aggregation group. MAVG is retained only for backward compatibility with existing applications.

Problems With Missing Data

Ensure that data you analyze using MAVG has no missing data points. Computing a moving average over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Computing the Moving Average When Number of Rows < width

For the (possibly grouped) resulting relation, the moving average considering *width* rows is computed where the rows are sorted by the *sort_expression* list.

When there are fewer than *width* rows, the average is computed using the current row and all preceding rows.

Examples

Example

Compute the 7-day moving average of sales for product code 10 for each day in the month of October, 1996.

```
SELECT cdate, itemID, MAVG(sumPrice, 7, date)
FROM (SELECT a1.calendar_date, a1.itemID,
SUM(a1.price)
FROM Sales a1
WHERE a1.itemID=10 AND a1.calendar_date
BETWEEN 96-10-01 AND 96-10-31
GROUP BY a1.calendar_date, a1.itemID) AS T1(cdate,
itemID, sumPrice);
```

Example

The following example calculates the 50-day moving average of the closing price of the stock for Zemlinsky Bros. Corporation. The ticker name for the company is ZBC.

```
SELECT MarketDay, ClosingPrice,
   MAVG(ClosingPrice,50, MarketDay) AS ZBCAverage
FROM MarketDailyClosing
WHERE Ticker = 'ZBC'
ORDER BY MarketDay;
```

The results for the query might look something like the following table.

MarketDay	ClosingPrice	ZBCAverage
12/27/1999	89 1/16	85 1/2
12/28/1999	91 1/8	86 1/16
12/29/1999	92 3/4	86 1/2
12/30/1999	94 1/2	87

MDIFF

Returns the moving difference between the specified value expression for the current row and the preceding *width* rows for each row in the partition.

Type

Teradata-specific function.

MDIFF Function Syntax

```
MDIFF ( value_expression, width, sort_spec [,...] )
```

Syntax Elements

value_expression

A numeric column or literal expression for which a moving difference is to be computed.

The expression cannot contain any ordered analytical or aggregate functions.

width

The number of previous rows to be used in the computation.

The value is always a positive integer literal.

The maximum is 4096.

sort_spec

```
sort_expression [ ASC | DESC ]
```

sort_expression

A literal or column expression or comma-separated list of literal or column expressions to be used to sort the values.

ASC

Ascending sort order.

DESC

Descending sort order.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for MDIFF are as follows:

- If operand x is character, the data type is the same as x and the format is the default format for FLOAT.
- If operand x is numeric, the data type is the same as x and the format is the same format as x.

- If operand is date, the data type is INTEGER and the format is the default format for INTEGER.

MDIFF Usage Notes

Using SUM Instead of MDIFF

The use of MDIFF is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is retained only for backward compatibility with existing applications. MDIFF(x, w, y) is equivalent to:

```
x - SUM(x) OVER (ORDER BY y
                 ROWS BETWEEN w PRECEDING AND w PRECEDING)
```

Problems With Missing Data

Ensure that rows you analyze using MDIFF have no missing data points. Computing a moving difference over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Computing the Moving Difference When No Preceding Row Exists

When the number of preceding rows to use in a moving difference computation is fewer than the specified width, the result is null.

Examples

Example

Display the difference between each quarter and the same quarter sales for last year for product code 10.

```
SELECT year_of_calendar, quarter_of_calendar,
MDIFF(sumPrice, 4, year_of_calendar, quarter_of_calendar)
FROM (SELECT a2.year_of_calendar,
a2.quarter_of_calendar, SUM(a2.Price) AS sumPrice
FROM Sales a1, SYS_CALENDAR.Calendar a2
WHERE a1.itemID=10 and a1.calendar_date=a2.calendar_date
GROUP BY a2.year_of_calendar, a2.quarter_of_calendar) AS T1
ORDER BY year_of_calendar, quarter_of_year;
```

Example

The following example computes the changing market volume week over week for the stock of company Horatio Parker Imports. The ticker name for the company is HPI.

```
SELECT MarketWeek, WeekVolume,
   MDIFF(WeekVolume,1,MarketWeek) AS HPIVolumeDiff
FROM
(SELECT MarketWeek, SUM(Volume) AS WeekVolume
FROM MarketDailyClosing
WHERE Ticker = 'HPI'
GROUP BY MarketWeek)
ORDER BY MarketWeek;
```

The result might look like the following table. Note that the first row is null for column HPIVolume Diff, indicating no previous row from which to compute a difference.

MarketWeek	WeekVolume	HPIVolumeDiff
11/29/1999	9817671	?
12/06/1999	9945671	128000
12/13/1999	10099459	153788
12/20/1999	10490732	391273
12/27/1999	11045331	554599

Related Information

- For information on the default format of data types, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on the SUM window function, see Window Aggregate Functions.

MEDIAN

For numeric values, returns the middle value or an interpolated value that would be the middle value after the values are sorted. Nulls are ignored in the calculation.

The function returns the same data type as the data type of the argument.

Type

MEDIAN is an aggregate function.

MEDIAN Function Syntax

```
MEDIAN ( value_expression )
```

Syntax Elements

value_expression

A single expression that must be a numeric or interval data type.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Example

MEDIAN, an inverse distribution function that assumes a continuous distribution model, is a specific case of PERCENTILE_CONT where the percentile value is 0.5.

```
MEDIAN  (value_expression)
```

is same as:

```
PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY value_expression)
```

Related Information

- See PERCENTILE_CONT/PERCENTILE_DISC.

MLINREG

Returns a predicted value for an expression based on a least squares moving linear regression of the previous *width* -1 (based on *sort_expression*) column values.

All rows in the results table except the first two, which are always null, display the predicted value.

Type

Teradata-specific function.

MLINREG Function Syntax

```
MLINREG ( value_expression, width, sort_expression [ ASC | DESC ] )
```

Syntax Elements

value_expression

The expression cannot contain any ordered analytical or aggregate functions.

width

The number of previous rows to be used in the computation.

The value is always a positive integer literal.

The maximum is 4096.

sort_expression

A column expression that defines the independent variable for calculating the linear regression.

For example, MLINREG(Sales, 6, Fiscal_Year_Month ASC), where Sales is the *value_expression*, 6 is the *width*, and Fiscal_Year_Month ASC is the *sort_expression*.

The data type of the column reference must be numeric or a data type that Vantage can successfully convert implicitly to numeric.

ASC

Ascending sort order.

DESC

Descending sort order.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for MLINREG are as follows:

Data Type: Same as operand x

- If operand x is character, the format is the default format for FLOAT.
- If operand x is numeric, date, or interval, the format is the same format as x.

MLINREG Usage Notes

Using ANSI-Compliant Window Functions Instead of MLINREG

Using ANSI-compliant window functions instead of MLINREG is strongly encouraged. MLINREG is a Teradata extension to the ANSI SQL:2011 standard, and is retained only for backward compatibility with existing applications.

Computing MLINREG When Preceding Rows < width - 1

When there are fewer than *width* -1 preceding rows, MLINREG computes the regression using all the preceding rows.

Example

Consider the *itemID*, *smonth*, and *sales* columns from *sales_table*:

```
SELECT itemID, smonth, sales
FROM fiscal_year_sales_table
ORDER BY itemID, smonth;
itemID  smonth    sales
------  --------  -----
A              1    100
A              2    110
A              3    120
A              4    130
A              5    140
A              6    150
A              7    170
A              8    190
A              9    210
A             10    230
A             11    250
A             12  ?
B              1     20
B              2     30
...
```

Assume that the NULL in the *sales* column is because in this example the month of December (month 12) is a future date and the value is unknown.

The following statement uses MLINREG to display the expected sales using past trends for each month for each product using the sales data for the previous six months.

```
SELECT itemID, smonth, sales, MLINREG(sales,7,smonth)
FROM fiscal_year_sales_table;
GROUP BY itemID;
itemID  smonth    sales  MLinReg(sales,7,smonth)
------  --------  -----  -----------------------
A              1    100  ?
A              2    110  ?
A              3    120           120
A              4    130           130
A              5    140           140
A              6    150           150
A              7    170           160
A              8    190           177
A              9    210           198
A             10    230           222
A             11    250           247
A             12  ?               270
B              1     20  ?
B              2     30  ?
...
```

Related Information

For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

MSUM

Computes the moving sum specified by a value expression for the current row and the preceding *n*-1 rows. This function is very similar to the MAVG function.

Type

Teradata-specific function.

MSUM Function Syntax

```
MSUM ( value_expression, width, sort_spec [,...] )
```

Syntax Elements

value_expression

The expression cannot contain any ordered analytical or aggregate functions.

width

The number of previous rows to be used in the computation.

The value is always a positive integer literal.

The maximum is 4096.

sort_spec

```
sort_expression [ ASC | DESC ]
```

sort_expression

A literal or column expression or comma-separated list of literal or column expressions to be used to sort the values.

ASC

Ascending sort order.

DESC

Descending sort order.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for MSUM are as follows:

Data Type: Same as operand x

- If operand x is character, the format is the default format for FLOAT.
- If operand x is numeric, the format is the same format as x.

MSUM Usage Notes

Using SUM Instead of MSUM

The use of MSUM is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is equivalent to the ANSI-compliant SUM window function. MSUM is retained only for backward compatibility with existing applications.

Problems With Missing Data

Ensure that data you analyze using MSUM has no missing data points. Computing a moving average over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Computing MSUM When Number of Rows < width

For data having fewer than *width* rows, MSUM computes the sum using all the preceding rows. MSUM returns the current sum rather than nulls when the number of rows in the sample is fewer than *width*.

Possible Result Overflow with SELECT Sum

When using this function, the result can create an overflow when the data type and format are not in sync. For a column defined as:

```
Salary Decimal(15,2) Format ‘$ZZZ,ZZ9.99’
```

The following query:

```
SELECT SUM (Salary) FROM Employee;
```

causes an overflow because the decimal operand and the format are not in sync.

To avoid possible overflows, explicitly specify the format for decimal sum to specify a format large enough to accommodate the decimal sum resultant data type.

```
SELECT Sum(Salary) (format ‘$Z,ZZZ,ZZZ,ZZ9.99) FROM Employee;
```

PERCENT_RANK

Returns the relative rank of rows for a *value_expression*.

Type

ANSI SQL:2011 window function.

Computation

The assigned rank of a row is defined as 1 (one) plus the number of rows that precede the row and are not peers of it.

PERCENT_RANK is expressed as an approximate numeric ratio between 0.0 and 1.0.

PERCENT_RANK has this value …	FOR the result row assigned this rank …
0.0	1.
1.0	highest in the result.

PERCENT_RANK Function Syntax

```
PERCENT_RANK() OVER (
  [ PARTITION BY column_reference [,...] ]
  ORDER BY value_spec [,...]
  [ RESET WHEN condition ]
)
```

Syntax Elements

OVER

Specifies how values are grouped, ordered, and considered when computing the cumulative, group, or moving function.

Values are grouped according to the PARTITION BY BEGIN and RESET WHEN clauses END, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.

PARTITION BY *column_reference* [,...]

The group or groups over which the function operates.

If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a partition.

PARTITION BY clause is also called the window partition clause.

ORDER BY

The order in which the values in a group or partition are sorted.

value_spec

```
value_expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

RESET WHEN

The group, or groups, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition.

If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a partition.

condition

A conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

For more information, see “RESET WHEN Condition Rules” and the “QUALIFY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

ANSI Compliance

This is ANSI SQL:2011 compliant.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

Result Type and Attributes

For PERCENT_RANK() OVER (PARTITION BY *x* ORDER BY *y direction*), the data type, format, and title are as follows:

Data Type	Format	Title
REAL	the default format for DECIMAL(7,6).	Percent_Rank(y direction)

Examples

Example: Relative Rank

Determine the relative rank, called the percent_rank, of Christmas sales.

The following query:

```
SELECT sales_amt,
PERCENT_RANK() OVER (ORDER BY sales_amt)
FROM xsales;
```

might return the following results. Note that the relative rank is returned in ascending order, the default when no sort order is specified and that the currency is not reported explicitly.

sales_amt	Percent_Rank
100.00	0.000000
120.00	0.125000
130.00	0.250000
140.00	0.375000
143.00	0.500000
147.00	0.625000
150.00	0.750000
155.00	0.875000
160.00	1.000000

Example: Rank and Relative Rank

Determine the rank and the relative rank of Christmas sales.

```
SELECT sales_amt,
RANK() OVER (ORDER BY sales_amt),
PERCENT_RANK () OVER (ORDER BY sales_amt)
FROM xsales;
```

sales_amt	Rank	Percent_Rank
100.00	1	0.000000
120.00	2	0.125000
130.00	3	0.250000
140.00	4	0.375000

sales_amt	Rank	Percent_Rank
143.00	5	0.500000
147.00	6	0.625000
150.00	7	0.750000
155.00	8	0.875000
160.00	9	1.000000

Example: PERCENT_RANK and CUM_DIST

The following SQL statement illustrates the difference between PERCENT_RANK and cumulative distribution.

```
SELECT sales_amt,
 PERCENT_RANK() OVER (ORDER BY sales_amt),
 CUME_DIST() OVER (ORDER BY sales_amt)
 FROM xsales;
```

sales_amt	PERCENT_Rank	CUME_DIST
100.	.000000	0.125000
120.	.142857	0.250000
130	.285714	.375000
140.	.428571	.500000
147.	.571429	.625000
150.	.714286	.750000
155.	.857143	.875000
160.	1.000000	1.000000

PERCENTILE_CONT/PERCENTILE_DISC

Returns an interpolated value that falls within its *value_expression* with respect to its sort specification.

The function returns the same data type as the data type of the argument.

Nulls are ignored in the calculation.

Type

PERCENTILE_CONT and PERCENTILE_DISC are aggregate functions.

PERCENTILE_CONT/PERCENTILE_DISC Function Syntax

```
{ PERCENTILE_CONT | PERCENTILE_DISC } ( value_expression_1 )
  WITHIN GROUP ( ORDER BY value_spec [,...] )
```

Syntax Elements

value_expression_1
A numeric value between 0 and 1 inclusive.

WITHIN GROUP
The order in which the values in a group or partition are sorted.

ORDER BY
The order in which the values in a group or partition are sorted.

value_spec

```
value_expression_2 [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

value_expression_2
A single expression that must be a numeric value.

ASC
Ascending sort order.

DESC
Descending sort order.

NULLS FIRST
NULL results are to be listed first.

NULLS LAST
NULL results are to be listed last.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

Both functions are inverse distribution functions that assume a continuous distribution.

- PERCENTILE_CONT returns a computed result after doing linear interpolation.
- PERCENTILE_DISC simply returns a value from the set of values.

Example

Using this table:

Area	Address	Price
Downtown	72 Easy Street	509000
Downtown	29 Right Way	402000
Downtown	45 Diamond Lane	203000
Downtown	76 Blind Alley	201000
Downtown	15 Tern Pike	199000
Downtown	444 Kanga Road	102000
Uptown	15 Peak Street	456000
Uptown	27 Primrose Path	349000
Uptown	44 Shady Lane	341000
Uptown	34 Design Road	244000
Uptown	2331 Highway 64	244000
Uptown	77 Sunset Strip	102000

the following SQL statement returns a computed result after doing linear interpolation, as shown in the table immediately below.

```
SELECT area,
       AVG(price),
       PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY price),
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY price)
FROM market
GROUP BY area;
```

Area	Average Price	PERC_DISC	PERC_CONT
Downtown	269333	201000	202000
Uptown	289333	244000	292500

QUANTILE

Computes the quantile scores for the values in a group.

Quantile

A quantile is a generic interval of user-defined width. For example, percentiles divide data among 100 evenly spaced intervals, deciles among 10 evenly spaced intervals, quartiles among 4, and so on. A quantile score indicates the fraction of rows having a *sort_expression* value lower than the current value. For example, a percentile score of 98 means that 98 percent of the rows in the list have a *sort_expression* value lower than the current value.

Type

Teradata-specific function.

QUANTILE Function Syntax

```
QUANTILE ( quantile_literal, sort_spec [,...] )
```

Syntax Elements

quantile_literal

A positive integer literal used to define the number of quantile partitions to be used.

Quantile values range from 0 through (Q-1), where Q is the number of quantile partitions specified by *quantile_literal*.

sort_spec

```
sort_expression [ ASC | DESC ]
```

sort_expression

A literal or column expression or comma-separated list of literal or column expressions to be used to sort the values.

For each row in the group, QUANTILE returns an integer value that represents the quantile of the *sort_expression* value for that row relative to the *sort_expression* value for all the rows in the group.

ASC

Ascending sort order.

DESC

Descending sort order.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for QUANTILE(Q, list) are as follows.

Data Type	Format	Title
INTEGER	the default format for the INTEGER data type	Quantile(Q, list)

QUANTILE Usage Notes

Using ANSI Window Functions Instead of QUANTILE

The use of QUANTILE is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard and is retained only for backward compatibility with existing applications.

To compute QUANTILE(q, s) using ANSI window functions, use the following:

```
   (RANK() OVER (ORDER BY s) - 1) * q / COUNT(*) OVER()
```

Examples

Example

Display each item and its total sales in the ninth (top) decile according to the total sales.

```
SELECT itemID, sumPrice
FROM (SELECT a1.itemID, SUM(price)
FROM Sales a1
GROUP BY a1.itemID) AS T1(itemID, sumPrice)
QUALIFY QUANTILE(10,sumPrice)=9;
```

Example

The following example groups all items into deciles by profitability.

```
SELECT Item, Profit, QUANTILE(10, Profit) AS Decile
FROM
   (SELECT Item, Sum(Sales) – (Count(Sales) * ItemCost) AS Profit
   FROM DailySales, Items
   WHERE DailySales.Item = Items.Item
   GROUP BY Item) AS Item;
```

The result might look like the following table.

Item	Profit	Decile
High Tops	97112	9
Low Tops	74699	7
Running	69712	6
Casual	28912	3
Xtrain	100129	9

Example

Because QUANTILE uses equal-width histograms to partition the specified data, it does not partition the data equally using equal-height histograms. In other words, do not expect equal row counts per specified quantile. Expect empty quantile histograms when, for example, duplicate values for *sort_expression* are found in the data.

For example, consider the following simple SELECT statement.

```
SELECT itemNo, quantity, QUANTILE(10,quantity) FROM inventory;
```

The report might look like this.

itemNo	quantity	Quantile(10, quantity)
13	1	0
9	1	0
7	1	0
2	1	0
5	1	0
3	1	0
1	1	0
6	1	0
4	1	0
10	1	0
8	1	0
11	1	0
12	9	9

Because the quantile sort is on quantity, and there are only two quantity scores in the inventory table, there are no scores in the report for deciles 1 through 8.

Related Information

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

RANK (ANSI)

Returns an ordered ranking of rows based on the *value_expression* in the ORDER BY clause.

All rows having the same *value_expression* value are assigned the same rank.

If *n* rows have the same *value_expression* values, then they are assigned the same rank, call it rank *r*. The next distinct value receives rank *r* +*n*. And so on.

Less formally, RANK sorts a result set and identifies the numeric rank of each row in the result. RANK returns an integer that represents the rank of each row in the result.

To use this function with time series data, see *Teradata Vantage™ - Time Series Tables and Operations*, B035-1208.

Type

ANSI SQL:2011 window function.

RANK Function Syntax (ANSI)

```
RANK() OVER (
  [ PARTITION BY column_reference [,...] ]
  ORDER BY value_spec [,...]
  [ RESET WHEN condition ]
  [ WITH TIES { LOW | HIGH | AVG | DENSE } ]
)
```

Syntax Elements

OVER

Specifies how values are grouped, ordered, and considered when computing the cumulative, group, or moving function.

Values are grouped according to the PARTITION BY BEGIN and RESET WHEN clauses END, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.

PARTITION BY *column_reference* [,...]

The group or groups over which the function operates.

If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a partition.

PARTITION BY clause is also called the window partition clause.

ORDER BY

The order in which the values in a group or partition are sorted.

value_spec

```
value_expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

RESET WHEN

The group, or groups, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition.

If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a partition.

condition

A conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

For more information, see “RESET WHEN Condition Rules” and the “QUALIFY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

WITH TIES

Specifies the rank for all ties:

Option	Description
LOW	All ties get lowest rank. Returns integer data type.
HIGH	All ties get highest rank. Returns integer data type.
AVG	All ties get average rank. Returns decimal data type.
DENSE	All ties are ranked as DENSE_RANK ranks them. Returns integer data type.

ASC

Ascending sort order.

DESC

Descending sort order.

NULLS FIRST

NULL results are to be listed first.

NULLS LAST

NULL results are to be listed last.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Result Type and Attributes

For RANK() OVER (PARTITION BY *x* ORDER BY *y direction*), the data type, format, and title are as follows.

Data Type	Format	Title
INTEGER	the default format for the INTEGER data type	Rank(y direction)

Examples

Example: Ranking Salespeople Based on Sales

This example ranks salespersons by sales region based on their sales.

```
SELECT sales_person, sales_region, sales_amount,
   RANK() OVER (PARTITION BY sales_region ORDER BY sales_amount DESC)
FROM sales_table;
```

sales_person	sales_region	sales_amount	Rank(sales_amount)
Garabaldi	East	100	1
Baker	East	99	2
Fine	East	89	3
Adams	East	75	4
Edwards	West	100	1
Connors	West	99	2
Davis	West	99	2

The rank column in the preceding table lists salespersons in declining sales order according to the column specified in the PARTITION BY clause (sales_region) and that the rank of their sales (sales_amount) is reset when the sales_region changes.

Example: Finding Differences Between RANK(ANSI) and DENSE_ RANK(ANSI)

The following SQL statement illustrates the difference between RANK(ANSI) and DENSE_RANK(ANSI), returning the RANK and DENSE_RANK for sales_person by sales_region and sales_amount.

```
SELECT sales_person, sales_region, sales_amount,
 RANK() OVER
    (PARTITION BY sales_region ORDER BY sales_amount DESC) as "Rank",
 DENSE_RANK() OVER
    (PARTITION BY sales_region ORDER BY sales_amount DESC) as "DenseRank"
 FROM sales_table;
```

sales_person	sales_region	sales_amount	Rank	DenseRank
Garabaldi	East	100	1	1
Baker	East	100	1	1
Fine	East	89	3	2
Adams	East	75	4	3
Edwards	West	100	1	1
Connors	West	99	2	2
Davis	West	99	2	2
Russell	West	50	4	3

Related Information

- For more information, see “RESET WHEN Condition Rules” and the “QUALIFY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- For an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

RANK (Teradata)

Returns the rank (1 … *n*) of all the rows in the group by the value of *sort_expression* list, with the same *sort_expression* values receiving the same rank.

A rank *r* implies the existence of exactly *r* -1 rows with *sort_expression* value preceding it. All rows having the same *sort_expression* value are assigned the same rank.

For example, if *n* rows have the same *sort_expression* values, then they are assigned the same rank, call it rank *r*. The next distinct value receives rank *r* +*n*.

Less formally, RANK sorts a result set and identifies the numeric rank of each row in the result. The only argument for RANK is the sort column or columns, and the function returns an integer that represents the rank of each row in the result.

Type

Teradata-specific function.

RANK Function Syntax (Teradata)

```
RANK ( sort_spec [,...] )
```

Syntax Elements

sort_spec

```
sort_expression [ ASC | DESC ]
```

sort_expression

A literal or column expression or comma-separated list of literal or column expressions to be used to sort the values.

The expression cannot contain any ordered analytical or aggregate functions.

ASC

Ascending sort order.

DESC

Descending sort order.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The data type, format, and title for RANK(x) are as follows.

Data Type	Format	Title
INTEGER	the default format for the INTEGER data type	Rank(x)

RANK Usage Notes (Teradata)

Using ANSI RANK Instead of Teradata RANK

The use of Teradata RANK is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is equivalent to the ANSI-compliant RANK window function. Teradata RANK is retained only for backward compatibility with existing applications.

Computing Top and Bottom Values

You can use RANK to compute top and bottom values as shown in the following examples.

Top(*n*, column) is computed as QUALIFY RANK(column DESC) <=*n.*

Bottom(*n*, column) is computed as QUALIFY RANK(column ASC) <=*n*.

Examples

Example

Display each item, its total sales, and its sales rank for the top 100 selling items.

```
SELECT itemID, sumPrice, RANK(sumPrice)
FROM
   (SELECT a1.itemID, SUM(a1.Price)
   FROM Sales a1
   GROUP BY a1.itemID AS T1(itemID, sumPrice)
   QUALIFY RANK(sumPrice) <=100;
```

Example

Sort employees alphabetically and identify their level of seniority in the company.

```
SELECT EmployeeName, (HireDate - CURRENT_DATE) AS ServiceDays,
RANK(ServiceDays) AS Seniority
FROM Employee
ORDER BY EmployeeName;
```

The result might look like the following table.

EmployeeName	Service Days	Seniority
Ferneyhough	9931	2
Lucier	9409	4
Revueltas	9408	5
Ung	9931	2
Wagner	10248	1

Example

Sort items by category and report them in order of descending revenue rank.

```
SELECT Category, Item, Revenue, RANK(Revenue) AS ItemRank
FROM ItemCategory,
   (SELECT Item, SUM(sales) AS Revenue
   FROM DailySales
   GROUP BY Item) AS ItemSales
WHERE ItemCategory.Item = ItemSales.Item
ORDER BY Category, ItemRank DESC;
```

The result might look like the following table.

Category	Item	Revenue	ItemRank
Hot Cereal	Regular Oatmeal	39112.00	4
Hot Cereal	Instant Oatmeal	44918.00	3
Hot Cereal	Regular COW	59813.00	2
Hot Cereal	Instant COW	75411.00	1

Related Information

- For information on the default format of data types, see “Data Type Formats and Format Phrases” in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information on the RANK window function, see RANK (ANSI).

ROW_NUMBER

Returns the sequential row number, where the first row is number one, of the row within its window partition according to the window ordering of the window.

Type

ANSI SQL:2011 window function.

ROW_NUMBER Function Syntax

```
ROW_NUMBER() OVER (
  [ PARTITION BY column_reference [,...] ]
  ORDER BY value_spec [,...]
  [ RESET WHEN condition ]
)
```

Syntax Elements

OVER

Specifies how values are grouped, ordered, and considered when computing the cumulative, group, or moving function.

Values are grouped according to the PARTITION BY BEGIN and RESET WHEN clauses END, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.

PARTITION BY *column_reference* [,...]

The group or groups over which the function operates.

ORDER BY

The order in which the values in a group or partition are sorted.

value_spec

```
value_expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

RESET WHEN

The group, or groups, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition.

If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a partition.

condition

A conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

For more information, see “RESET WHEN Condition Rules” and the “QUALIFY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

ASC

Ascending sort order.

DESC

Descending sort order.

NULLS FIRST

NULL results are to be listed first.

NULLS LAST

NULL results are to be listed last.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

ROW_NUMBER Usage Notes

Window Aggregate Equivalent

```
  ROW_NUMBER() OVER (PARTITION BY  column
 ORDER BY  value
)
```

is equivalent to

```
  COUNT(*) OVER (PARTITION BY  column
 ORDER BY  value
```

```
ROWS UNBOUNDED PRECEDING).
```

Example

To order salespersons based on sales within a sales region, the following SQL query might yield the following results.

```
SELECT ROW_NUMBER() OVER (PARTITION BY sales_region
                          ORDER BY sales_amount DESC),
sales_person, sales_region, sales_amount
FROM sales_table;

Row_Number()  sales_person  sales_region  sales_amount
------------  ------------  ------------  ------------
           1  Baker         East                   100
           2  Edwards       East                    99
           3  Davis         East                    89
           4  Adams         East                    75
           1  Garabaldi     West                   100
           2  Connors       West                    99
           3  Fine          West                    99
```

Related Information

- For more information, see “RESET WHEN Condition Rules” and the “QUALIFY Clause” in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.
- For more information on COUNT, see Window Aggregate Functions.

Regular Expression Functions

The following sections describe regular expression functions you can use to search and manipulate strings.

These functions use Perl 5 syntax and semantics. For more information about regular expression syntax and semantics, visit the Perl Compatible Regular Expressions (PCRE) open source website at: http://www.pcre.org/.

REGEXP_SUBSTR

Extracts a substring from *source_string* that matches a regular expression specified by *regexp_string*.

REGEXP_SUBSTR supports 2, 3, 4, or 5 parameters.

REGEXP_SUBSTR is an embedded services system function.

REGEXP_SUBSTR Function Syntax

```
[TD_SYSFNLIB.] REGEXP_SUBSTR ( source_string, regexp_string
  [, position_arg, occurrence_arg, match_arg ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character argument.

If *source_string* is NULL, NULL is returned.

regexp_string

A character argument.

If *regexp_string* is NULL, NULL is returned.

position_arg

A numeric argument.

position_arg specifies the position in source_string from which to start searching (default is 1).

If *position_arg* is greater than the input string length, NULL is returned.

If *position_arg* is NULL, the value NULL is used. If *position_arg* is not specified, the default (1) is used.

occurrence_arg

A numeric argument.

Specifies the number of the occurrence to be returned (default is 1). For example, if *occurrence_arg* is 2, the function matches the first occurrence in *source_string* and starts searching from the character following the first occurrence in *source_string* for the second occurrence in *source_string*.

If *occurrence_arg* is greater than the number of matches found, NULL is returned.

If *occurrence_arg* is NULL, a NULL result is returned. If *occurrence_arg* is omitted, the default value (1) is used.

match_arg

A character argument.

Valid values are:

- 'i' = case-insensitive matching.
- 'c' = case sensitive matching.
- 'n' = the period character (match any character) can match the newline character.
- 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option, the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*.
- 'l' = if *source_string* exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail.
- 'x' = ignore whitespace.

The argument can contain more than one character. If a character in the argument is not valid, then that character is ignored.

If *match_arg* is not specified, is NULL, or is empty:

- The match is case-sensitive.
- A period does not match the newline character.
- *source_string* is treated as a single line.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR, or CLOB.
- *regexp_string* = CHAR or VARCHAR
- *position_arg* = NUMBER
- *occurrence_arg* = NUMBER
- *match_arg* = VARCHAR

The *source_string* maximum size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum source size is 32000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum source size is 64000 bytes.
- For parameters that are Latin or Unicode CLOBs, the maximum source size is 16 MB.

The *regexp_string* maximum pattern string size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum *regexp_string* size is 32000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum *regexp_string* size is 32000 bytes.
- For parameters that are Latin CLOBs, the maximum *regexp_string* size is 30000 bytes.
- For parameters that are Unicode CLOBs, the maximum *regexp_string* size is 30000 bytes.

The maximum return string size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum return string size is 16000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum return string size is 16000 bytes.
- For parameters that are Latin or Unicode CLOBs, the maximum return string size is 16 MB.

REGEXP_SUBSTR returns an error if this size is exceeded unless *match_arg* = 'l', in which case, it returns NULL.

The *x* match option ignores whitespace characters in the pattern/regexp_string. By default, whitespace characters match themselves.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

REGEXP_SUBSTR is a scalar function whose return value data type depends on the data type associated with *source_string* input parameter that is passed into the function.

A *source_string* of:

- CHAR, VARCHAR returns VARCHAR in the same character set as *source_string*.
- CLOB returns CLOB in the same character set as *source_string*.

Examples

Example

The following query:

```
SELECT REGEXP_SUBSTR('mint chocolate chip', 'ch(i|o)p', 1, 1, 'c');
```

returns 'chip'.

Example

The following query:

```
SELECT REGEXP_SUBSTR('mint chocolate chip chop', ' ch(i|o)p', 1, 2, 'i');
```

returns 'chop' because it is the second occurrence of the match.

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For information on activating and invoking embedded services functions, see Embedded Services System Functions.

REGEXP_REPLACE

Replaces portions of source_string that match *regexp_string* with the *replace_string*.

REGEXP_REPLACE supports 2, 3, 4, 5, or 6 parameters.

REGEXP_REPLACE is an embedded services system function.

REGEXP_REPLACE Function Syntax

```
[TD_SYSFNLIB.] REGEXP_REPLACE ( source_string, regexp_string
  [, replace_string, position_arg, occurrence_arg, match_arg ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character argument.

If *source_string* is NULL, NULL is returned.

regexp_string

A character argument.

If *regexp_string* is NULL, NULL is returned.

replace_string

A character argument.

If a *replace_string* is not specified, is NULL or is an empty string, the matches are removed from the result.

The maximum backreference number in a *replace_string* is 9 (for example, \9). Any backreference in the *replace_string* that is higher than 9 is considered a backreference.

position_arg

A numeric argument.

position_arg specifies the position in source_string from which to start searching (default is 1).

If *position_arg* is greater than the input string length, NULL is returned.

If *position_arg* is NULL, the value NULL is used. If *position_arg* is not specified, the default (1) is used.

occurrence_arg

A numeric argument.

Specifies the occurrence to replace the match with *replace_string*.

- If a value of 0 is specified, all occurrences are replaced.
- If the value is greater than 1, the search begins for the second occurrence beginning with the first character following the first occurrence of the *regexp_string*, and so on.

If *occurrence_arg* is greater than the number of matches found, nothing is replaced and *source_string* is returned.

If *occurrence_arg* is NULL, a NULL result is returned. If *occurrence_arg* is omitted, 0 is the default value.

match_arg

A character argument.

The argument can contain more than one character. If a character in the argument is not valid, then that character is ignored.

If *match_arg* is not specified, is NULL, or is empty:

- The match is case-sensitive.
- A period does not match the newline character.
- *source_string* is treated as a single line.

- 'i' = case-insensitive matching.
- 'c' = case-sensitive matching.
- 'n' = the period character (match any character) can match the newline character.
- 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option, the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*.
- 'l' = For a CLOB data type, if *source_string* exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail. You can only specify this option for a CLOB data type.
- 'x' = ignore whitespace.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR, or CLOB
- *regexp_string* = CHAR or VARCHAR
- *replace_string* = CHAR, VARCHAR CLOB
- *position_arg* = NUMBER
- *occurrence_arg* = NUMBER
- *match_arg* = VARCHAR

The *source_string* maximum size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum source size is 32000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum source size is 64000 bytes.

- For parameters that are Latin or Unicode CLOBs, the maximum source size is 16 MB.

The *regexp_string* maximum pattern string size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum *regexp_string* size is 16000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum *regexp_string* size is 16000 bytes.
- For parameters that are Latin or Unicode CLOBs, the maximum *regexp_string* size is 30000 bytes.

The maximum replace string size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum replace string size is 16000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum replace string size is 16000 bytes.
- For parameters that are Latin or Unicode CLOBs, the maximum replace string size is 30000 bytes.

The maximum return string size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum return string size is 16000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum return string size is 16000 bytes.
- For parameters that are Latin or Unicode CLOBs, the maximum return string size is 16 MB.

The function returns an error if this size is exceeded unless *match_arg* = 'l', in which case, it returns the original string.

If *position_arg* is omitted, the default value (1) is used.

The maximum backreference number in a replace string is 9 (for example, \9). Any backreference in the replace string that is higher than 9 is not considered a backreference.

The *x* match option ignores whitespace characters in the pattern/regexp_string. By default, whitespace characters match themselves.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

REGEXP_REPLACE is a scalar function whose return value data type depends on the data type associated with source_string input parameter that is passed into the function.

A source_string of:

- CHAR, VARCHAR returns VARCHAR in the same character set as source_string.
- CLOB returns CLOB in the same character set as source_string.

REGEXP_REPLACE Usage Notes

Limitation: NULL inside Input Strings

REGEXP_REPLACE has a limitation in handling ASCII Chr(0), which is NULL. If you concatenate two strings using ASCII Chr(0), REGEXP_REPLACE cannot handle that input string. For example, consider the following code:

```
sel TD_SYSFNLIB.RegExp_Replace ('a'||chr(0)||'bc', '[b]', 'X', 1, 0) AS
regex_rep_input_string;
```

The function is supposed to replace the b in the input string `'a'||chr(0)||'bc'` with X. The result should be a `Xc`. However, because of the limitation, the result is a.

Examples

Example

The following query:

```
SELECT REGEXP_REPLACE('Hello World World', '(world)$', 'My', 1, 1, 'i');
```

returns the result "Hello World My".

Example

The following query:

```
SELECT REGEXP_REPLACE('Friday is the best day of the week.', 'of the week',
'EVER', 1, 1, 'c');
```

returns the result 'Friday is the best day EVER'.

Example

The following query:

```
SELECT REGEXP_REPLACE('Hello Santa says ho ho', 'ho', 'HO!', 1, 2, 'c');
```

returns the result 'Hello Santa says ho HO!'.

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For information on activating and invoking embedded services functions, see Embedded Services System Functions.

REGEXP_INSTR

Searches *source_string* for a match to *regexp_string*.

REGEXP_INSTR supports 2, 3, 4, 5, or 6 parameters.

REGEXP_INSTR is an embedded services system function.

REGEXP_INSTR Function Syntax

```
[TD_SYSFNLIB.] REGEXP_INSTR ( source_string, regexp_string
  [, position_arg, occurrence_arg, return_opt, match_arg ] )
```

Syntax Elements

TD_SYSFNLIB

Name of the database where the function is located.

source_string

A character argument.

If *source_string* is NULL, NULL is returned.

regexp_string

A character argument.

If *regexp_string* is NULL, NULL is returned.

position_arg

A numeric argument.

position_arg specifies the position in source_string from which to start searching (default is 1).

If *position_arg* is greater than the input string length, zero is returned.

If *position_arg* is NULL, NULL is used. If *position_arg* is not specified, the default (1) is used.

occurrence_arg

A numeric argument.

Specifies the number of the occurrence to be returned. For example, if *occurrence_arg* is 2, the function matches the first occurrence in *source_string* and starts searching from the character following the first occurrence in *source_string* for the second occurrence in *source_string*.

If *occurrence_arg* is greater than the number of matches found, 0 is returned.

If *occurrence_arg* is NULL, a NULL result is returned. If *occurrence_arg* is omitted, the default value (1) is used.

return_opt

A numeric argument.

Valid values are:

- 0 = function returns the beginning position of the match (default).
- 1 = function returns the end position (character following the occurrence) of the match.

If this syntax element is NULL, NULL is returned. If the syntax element is omitted, the default value (0) value is used.

match_arg

A character argument.

Valid values are:

- 'i' = case-insensitive matching.
- 'c' = case sensitive matching.
- 'n' = the period character (match any character) can match the newline character.
- 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option, the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*.
- 'l' = if source_string exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail.

- 'x' = ignore whitespace.

If *match_arg* is not specified, is NULL, or is empty:

- The match is case sensitive.
- A period does not match the newline character.
- *source_string* is treated as a single line.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR, or CLOB
- *regexp_string* = CHAR or VARCHAR
- *position_arg* = NUMBER
- *occurrence_arg* = NUMBER
- *return_opt* = NUMBER
- *match_arg* = VARCHAR

The *source_string* maximum size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum source size is 32000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum source size is 64000 bytes.
- For parameters that are Latin or Unicode CLOBs, the maximum source size is 16 MB.

The *regexp_string* maximum pattern string size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum *regexp_string* size is 32000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum *regexp_string* size is 32000 bytes.
- For parameters that are Latin CLOBs, the maximum *regexp_string* size is 30000 bytes.
- For parameters that are Unicode CLOBs, the maximum *regexp_string* size is 60000 bytes.

The function returns an error if this size is exceeded unless *match_arg* = 'l' is specified. If this is specified, a NULL is returned instead of an error.

The *x* match option ignores whitespace characters in the pattern/regexp_string. By default, whitespace characters match themselves.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

REGEXP_INSTR is a scalar function whose return value data type is:

- INTEGER if source_string is a CHAR or VARCHAR
- BIGINT if source_string is a CLOB

Examples

Example

The following query:

```
SELECT REGEXP_INSTR('Hello Santa says ho ho','Hello Santa says ho ho', 1, 1,
1, 'c');
```

returns the result 23.

Example

The following query:

```
SELECT REGEXP_INSTR('Hello Santa says ho ho', 'Hello Santa says ho ho', 1, 1,
0, 'c');
```

returns the result 1.

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For information on activating and invoking embedded services functions, see Embedded Services System Functions.

REGEXP_SIMILAR

Compares *source_string* to *regexp_string* and returns integer value.

REGEXP_SIMILAR supports 2 or 3 parameters.

REGEXP_SIMILAR is an embedded services system function.

REGEXP_SIMILAR Function Syntax

```
[TD_SYSFNLIB.] REGEXP_SIMILAR ( source_string, regexp_string [, match_arg ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character argument.

If *source_string* is NULL, NULL is returned.

regexp_string

A character argument.

If *regexp_string* is NULL, NULL is returned.

match_arg

A character argument.

Valid values are:

- 'i' = case-insensitive matching.
- 'c' = case sensitive matching.
- 'n' = the period character (match any character) can match the newline character.
- 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option, the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*.
- 'l' = if *source_string* exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail.
- 'x' = ignore whitespace.

The argument can contain more than one character. If a character in the argument is not valid, then that character is ignored.

If *match_arg* is not specified, is NULL, or is empty:

- The match is case-sensitive.
- A period does not match the newline character.

- *source_string* is treated as a single line.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR, CLOB
- *regexp_string* = CHAR, VARCHAR
- *match_arg* = VARCHAR

The *source_string* maximum size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum source size is 32000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum source size is 64000 bytes.
- For parameters that are Latin or Unicode CLOBs, the maximum source size is 16 MB.

The *regexp_string* maximum pattern string size is:

- For parameters that are Latin CHAR or VARCHAR, the maximum *regexp_string* size is 32000 bytes.
- For parameters that are Unicode CHAR or VARCHAR, the maximum *regexp_string* size is 32000 bytes.
- For parameters that are Latin CLOBs, the maximum *regexp_string* size is 30000 bytes.
- For parameters that are Unicode CLOBs, the maximum *regexp_string* size is 60000 bytes.

The function returns an error if this size is exceeded unless *match_arg* = 'l' is specified. If this is specified, a NULL is returned instead of an error.

The *x* match option ignores whitespace characters in the pattern/regexp_string. By default, whitespace characters match themselves.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

REGEXP_SIMILAR is a scalar function whose return value data type in an integer value:

- 1 (true) if the entire string matches *regexp_arg*
- 0 (false) if the entire string does not match *regexp_arg*

Example

The following query:

```
SELECT name FROM customers WHERE REGEXP_SIMILAR(name, '(Mike B(i|y)rd)|
(Michael B(i|y)rd)', 'c') = 1;
```

returns the names from the customers table that match:

- 'Mike Bird'
- 'Mike Byrd'
- 'Michael Bird'
- 'Michael Byrd'

The matching is case sensitive.

Related Information

- "Compatible Types" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For information on activating and invoking embedded services functions, see Embedded Services System Functions.

REGEXP_SPLIT_TO_TABLE

Splits *source_string* into a table of strings using *regexp_string* as the delimiter.

REGEXP_SPLIT_TO_TABLE is an embedded services system function.

REGEXP_SPLIT_TO_TABLE Function Syntax

```
[TD_SYSFNLIB.] REGEXP_SPLIT_TO_TABLE ( inkey, source_string, regexp_string,
match_arg )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

inkey

A numeric or character argument that uniquely identifies the *source_string* in the output result set.

If the *inkey* is NULL, an empty string is returned.

source_string

A character argument.

If *source_string* is NULL, 0 rows are returned. Zero rows are also returned if the *source_string* is the empty string.

regexp_string

A character argument.

If *regexp_string* is NULL, the original *source_string* is returned.

match_arg

A character argument.

Valid values are:

- 'i' = case-insensitive matching.
- 'c' = case sensitive matching.
- 'n' = the period character (match any character) can match the newline character.
- 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option, the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*.
- 'l' = if *source_string* exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail. Although the maximum source_string size is 16 MB, the resulting token can only be VARCHAR, and has a maximum return token size of 64000 bytes.

If *match_arg* is not specified, an error is returned.

If there is no match, the original *source_string* is returned.

outkey

The value of *inkey*.

token_ndx

The ordinal position of the token in the input string.

token

A character argument.

The *token* from the input string in the same character set as *instring*.

Note:

REGEXP_SPLIT_TO_TABLE does not support the output type of CLOB in the token section.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *inkey* = NUMERIC, VARCHAR
- *source_string* = CHAR, VARCHAR
- *regexp_string* = CHAR, VARCHAR
- *match_arg* = VARCHAR

The function returns an error if this size is exceeded or if *match_arg* = 'l' is specified.

If the *inkey* is null, an empty string is returned.

The *x* match option ignores whitespace characters in the pattern/regexp_string. By default, whitespace characters match themselves.

If Vantage passes constants as the second and third parameter in an OREPLACE call, the character type of the first argument is passed as Unicode, and calls oreplace_unicode() with the return type VARCHAR in Unicode charset.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result row type is:

- *outkey* = NUMERIC, VARCHAR
- *token_ndx* = INTEGER
- *token* = VARCHAR

Example

An example of a SELECT statement:

```
SELECT * from table(regexp_split_to_table
('phonemaker','Apple&Microsoft&Google','&','c')
returns (outkey varchar(30), token_ndx integer, token varchar(100))) as t1;
```

returns a table with the following rows:

```
outkey                            token_ndx          token
------------------------------ ----------------  -----------------------
p h o n e  m a k e r              1                  A p p l e
p h o n e  m a k e r              2                  M i c r o s o f t
p h o n e  m a k e r              3                  G o o g l e
```

String Operators and Functions

String Operators and Functions

The following sections describe string operators and functions.

About String Functions

SQL provides a concatenation operator and string functions to translate, concatenate, and perform other operations on strings.

The functions documented in the following sections are designed primarily to work with strings of characters. Because many of them can also process byte and numeric literal and literal data strings, the term *string* is frequently used here to refer to all three of these data type families.

Data Types on Which String Functions can Operate

The following table lists all the data types that can be processed as strings. Note that not all types are acceptable to all functions. See the individual functions for the types they can process.

Data Type Grouping		
Character	Byte	Numeric
• CHARACTER • VARCHAR • CLOB	• BYTE • VARBYTE • BLOB	• BYTEINT • DECIMAL • FLOAT • INTEGER • NUMERIC • SMALLINT

ANSI Equivalence of Teradata SQL String Functions

Several of the Teradata SQL string functions are extensions to the ANSI SQL:2011 standard.

To maintain ANSI compatibility, use the ANSI equivalent functions instead of Teradata SQL string functions, when available.

Change this Teradata string function …	To this ANSI string function in new applications …
INDEX	POSITION

Change this Teradata string function ...	To this ANSI string function in new applications ...
MINDEX†	
SUBSTR	SUBSTRING
MSUBSTR†	
† These functions are no longer documented because their use is deprecated and they will no longer be supported after support for KANJI1 is dropped.	

The following Teradata functions have no ANSI equivalents:

- CHAR2HEXINT
- SOUNDEX
- TRANSLATE_CHK
- UPPER
- VARGRAPHIC

Additional Functions That Operate on Strings

SQL provides other string functions and operators that are not discussed here.

FOR more information on ...	SEE ...
attribute functions that return descriptive information about strings, such as: • BYTE • CHARACTER_LENGTH/ CHAR_LENGTH • OCTET_LENGTH	Attribute Functions.
comparison operators	Comparison Operators and Functions.
the LIKE predicate	Logical Predicates.

Effects of Server Character Sets on Character String Functions

String functions that operate on character data follow the rules listed below.

Uppercase Character Conversion for LATIN

For the LATIN server character set, the method of converting to uppercase characters is based on ISO 8859 Latin1.

Logical Characters vs. Physical Characters

For UNICODE, GRAPHIC and KANJISJIS server character sets, the functions operate on a logical character basis, except for the functions that are sensitive to the ANSI mode vs. Teradata mode switch.

Although the storage space for KANJISJIS is allocated on a physical basis and is not ANSI compatible, all string operations on this type operate on a character basis as dictated by ANSI.

Untranslatable KANJI1 Characters

Note:

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see *KANJI1 Character Set* in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Character string functions do not work on all characters in the KANJI1 server character set when the session character set is UTF8 or UTF16, because the KANJI1 server character set is ambiguous in regard to multibyte characters and some single-byte characters.

Recommendation: Unless the KANJI1 server character set is required, use the UNICODE server character set with the UTF8 and UTF16 session character sets for best results.

The following single-byte characters in KanjiEBCDIC to KANJI1 translations are mapped to the following Unicode character names.

Hexadecimal Value	Character	Unicode Character Name
0x10	¢	CENT SIGN
0x11	£	POUND SIGN
0x12	¬	NOT SIGN
0x13	\	REVERSE SOLIDUS
0x14	~	TILDE

However, with a KanjiSJIS character set, these hexadecimal values map to control characters.

Implicit Server Character Set Translation

For functions that operate on more than one argument, if the arguments have different server character sets, implicit translation rules take effect.

Concatenation Operator

Concatenates string expressions.

Concatenation Operator Syntax

```
string_expression_1 { || | ¦¦ } string_expression_2
  [ { || | ¦¦ } string_expression_n ] [...]
```

Note:

The bold or colored vertical bar in the preceding syntax indicates a choice. One choice is two vertical bars; the other is two broken vertical bars.

Syntax Elements

string_expression_1
string_expression_2
string_expression_n

A byte, numeric, or character string or string expression.

Argument Types and Rules

Use the concatenation operator on strings and string expressions of type:

- Byte

  If any argument is a byte type, all other arguments must also be byte types.
- Numeric

  A numeric argument is converted to a character string using the format for the numeric value. For details about implicit numeric to character data type conversion, see “Implicit Numeric-to-Character Conversion”
- Character

  When the arguments are both character types, but have different server character sets, then implicit string conversion occurs. For details, see "Implicit Character-to-Character Translation" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

- UDTs that have implicit casts to a predefined character type.

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

  Implicit type conversion of UDTs for system operators and functions, including the concatenation operator, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

  For more information on implicit type conversion of UDTs, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ANSI Compliance

Solid and broken VERTICAL LINE character pairs (||) are ANSI SQL:2011 compliant forms of the concatenation operator.

Result Type and Attributes

The result of a concatenation operation is a string formed by concatenating the arguments in a left-to-right direction.

Here are the default result type and attributes for *arg1* || *arg2*:

- If the arguments are byte strings, the result is a byte string.
- If the arguments are numeric, character strings, or UDTs that are implicitly cast to character strings, the result is a character string.

If either argument is null, the result is null.

The data types and attributes of the arguments determine whether the result type of a concatenation operation is a fixed length or varying length string. Result types appear in the following table, where *n* is the sum of the lengths of all arguments:

IF this argument …	Is this data type or attribute …	THEN the result is this data type or attribute …
either	VARBYTE	VARBYTE(*n*)
	VARCHAR	VARCHAR(*n*)
	numeric	
	UDT that is implicitly cast to VARCHAR	
	CLOB	CLOB(*n*)
	BLOB	BLOB(*n*)
both	BYTE	BYTE(*n*)

IF this argument …	Is this data type or attribute …	THEN the result is this data type or attribute …
	CHARACTER (with same server character set)	CHARACTER(*n*)
	UDT that is implicitly cast to CHARACTER (with the same server character set)	
	CHARACTER (with different server character sets)	VARCHAR(*n*)
	UDT that is implicitly cast to CHARACTER (with different server character sets)	
	numeric	

When either argument is a character string that specifies the CASESPECIFIC attribute, the result also specifies the CASESPECIFIC attribute.

Concatenation Operator Usage Notes

Concatenating Character Strings Having Different Server Character Sets

There are special considerations for the concatenation of character strings that specify different server character sets in the CHARACTER SET attribute.

Implicit translation rules apply.

If the strings are fixed strings, then the result is varying with length equal to the sum of the lengths of the strings being concatenated.

This is true regardless of whether the string lengths are defined in terms of bytes or characters. So, a fixed *n* -byte KANJISJIS character string concatenated with a fixed *m* -character UNICODE string produces a VARCHAR(m+n) CHARACTER SET UNICODE result.

Consider the following table definition:

```
CREATE TABLE tab1
   (cunicode  CHARACTER(4)  CHARACTER SET UNICODE
   ,clatin    CHARACTER(3)  CHARACTER SET LATIN
   ,csjis     CHARACTER(3)  CHARACTER SET KANJISJIS);
```

The following values are inserted into table tab1:

```
INSERT tab1 ('abc', 'abc', 'abc');
```

The following table illustrates these concatenation properties.

Concatenation	Result	Type of Result
cunicode || clatin	'abcΔ abc'	VARCHAR(7) CHARACTER SET UNICODE
clatin || csjis	'abcabc'	VARCHAR(6) CHARACTER SET UNICODE
cunicode || csjis	'abcΔ abc'	VARCHAR(7) CHARACTER SET UNICODE

With the exception of KanjiEBCDIC, concatenation of KANJI1 character strings acts as described above. Under KanjiEBCDIC, any adjacent shift-out (<) and shift-in (>) characters within the resulting expression are removed. In this case, the result string is padded as necessary with trailing <single-byte space> characters.

Examples

Example: Using Concatenation to Create More Readable Results

Literals, spaces, and the TITLE phrase can be included in the operation definition to format the result heading and improve readability.

For example, the following definition returns side titles, evenly spaced result strings, and a blank heading.

```
SELECT ('Sex ' || sex ||', Marital Status ' || mstat)(TITLE ' ')
FROM Employee ;

Sex M, Marital Status S
Sex F, Marital Status M
Sex M, Marital Status M
Sex F, Marital Status M
Sex F, Marital Status M
Sex M, Marital Status M
Sex F, Marital Status W
   ...
```

Example: Concatenating First Name With Last Name

Consider a table called names that contains last and first names columns, defined as VARCHAR, as listed here:

```
lname         fname
------------  ------------
Ryan          Loretta
Villegas      Arnando
```

```
    Kanieski      Carol
    Brown         Alan
```

Use string concatenation and a space separator to combine first and last names:

```
    SELECT fname ||' '|| lname
    FROM names
    ORDER BY lname ;
```

The result is:

```
    ((fname||' ')||lname)
    ---------------------
    Alan Brown
    Carol Kanieski
    Loretta Ryan
    Arnando Villegas
```

Example: Concatenating Last Name With First Name

Change the SELECT and the separator to obtain last and first names:

```
    SELECT lname||', '||fname
    FROM names
    ORDER BY lname;
```

The result is:

```
    ((lname||', ')||fname)
    ----------------------
    Brown, Alan
    Kanieski, Carol
    Ryan, Loretta
    Villegas, Arnando
```

Example: Concatenating Byte Strings

This example shows how to concatenate byte strings. Consider the following table definition:

```
CREATE TABLE tsttbla
   (column_1 BYTE(2)
   ,column_2 VARBYTE(10)
   ,column_3 BLOB(128K) );
```

The following values are inserted into table tsttbla:

```
    INSERT tsttbla ('4142'XB, '7A7B7C'XB, '1A1B1C2B2C'XB);
```

The following SELECT statement concatenates column_2 and column_1 and column_3:

```
SELECT (column_2 || column_1 || column_3) (FORMAT 'X(20)')
FROM tsttbla ;
```

The result is:

```
((column_2||column_1)||column_3)
--------------------------------
7A7B7C41421A1B1C2B2C
```

The resulting data type is BLOB.

Examples for Japanese Character Sets

The following tables show the results of concatenating string expressions under each of the Kanji character sets supported by Vantage.

These examples assume that the string expressions follow the rules defined in the SQL data definition rules in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Example: KanjiEBCDIC

```
    string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
< ABC >	< DEF >G	< ABCDEF >G
< ABC >	<>	< ABC >
< ABC >a	< DEF >	< ABC >a< DEF >

Example: KanjiEUC

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
ABC m	DEF g	ABC mDEF g
ss3 A ss2 B m	ss3 C	ss3 A ss2 B m ss3 C

Example: KanjiShift-JIS

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
mnABC X	B	mnABC X B
mnABC X	g	mnABC X g

Related Information

For details on implicit translation rules, see "Implicit Character-to-Character Translation" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

ASCII

Returns the decimal representation of the first character in *string_expr* as a NUMBER value. The decimal representation will reflect the character set of the input string.

ASCII is an embedded services system function.

ASCII Function Syntax

```
[TD_SYSFNLIB.] ASCII ( string_expr )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

string_expr

A character string or string expression.

If *string_expr* is NULL, NULL is returned.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- CHAR
- VARCHAR
- CLOB

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is NUMBER. ASCII returns the decimal representation in the character set of the input argument. For example, if a Unicode string is passed to the function, the decimal representation of the Unicode character is returned.

Example

The following query returns the result 121.

```
SELECT ASCII('y');
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

CHAR2HEXINT

Returns the hexadecimal representation for a character string.

Use CHAR2HEXINT on character strings or character string expressions.

CHAR2HEXINT is not supported for CLOBs.

CHAR2HEXINT Function Syntax

```
CHAR2HEXINT ( character_string_expr )
```

Syntax Elements

character_string_expr

A character string or character string expression for which the hexadecimal representation is to be returned.

Result Type and Attributes

Here are the default attributes for CHAR2HEXINT(*character_string_expression*):

Data Type	Heading
CHARACTER	Char2HexInt(*character_string_expression*)

The length of the result is twice the length of *character_string_expression*.

The server character set of the result is LATIN.

If the *character_string_expression* argument is *null*, the result is *null*.

CHAR2HEXINT Usage Notes

CHAR2HEXINT and Literal Strings

You can apply CHAR2HEXINT to a string literal to determine its hexadecimal equivalent.

Character literals are treated as VARCHAR(*n*) CHARACTER SET UNICODE, where *n* is the length of the literal.

The following statement and results illustrate how CHAR2HEXINT operates on literal strings:

```
SELECT CHAR2HEXINT('123');

Char2HexInt('123')
```

```
   ----------------------
   003100320033
```

UDT Arguments

By default, Vantage performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and a predefined character type.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

Implicit type conversion of UDTs for system operators and functions, including CHAR2HEXINT, is a Teradata extension to the ANSI SQL standard.

Examples

Example

Assume that the system was enabled with Japanese language support during system initialization (sysinit).

```
   CREATE TABLE tab1
               (clatin   CHAR(3)  CHARACTER SET LATIN
               ,cunicode CHAR(3)  CHARACTER SET UNICODE
               ,csjis    CHAR(3)  CHARACTER SET KANJISJIS
               ,cgraphic CHAR(3)  CHARACTER SET GRAPHIC
               ,ckanji1  CHAR(3)  CHARACTER SET KANJI1);

   INSERT INTO tab1('abc','abc','abc',_GRAPHIC 'ABC
','abc');
```

The bold uppercase LATIN characters in the example represent full width LATIN characters.

CHAR2HEXINT returns the following results for the character strings inserted into tab1.

This function ...	Returns this result ...
CHAR2HEXINT(clatin)	616263
CHAR2HEXINT(cunicode)	006100620063'
CHAR2HEXINT(csjis)	616263
CHAR2HEXINT(cgraphic)	FF41FF42FF43
CHAR2HEXINT(ckanji1)	616263

Example

To find the internal hexadecimal representation of all table names, submit the following SELECT statement using CHAR2HEXINT.

```
SELECT CHAR2HEXINT(TRIM(t.tablename))(FORMAT 'X(30)')
(TITLE 'Internal Hex Representation of TableName')
,t.tablename (TITLE 'TableName')
FROM dbc.tables T
WHERE t.tablekind = 'T'
ORDER BY t.tablename;
```

Partial output from this SELECT statement is similar to the following report:

```
Internal Hex Representation of TableName   TableName
----------------------------------------   ----------------
416363657373526967687473                    AccessRights
4163634C6F6752756C6554626C                  AccLogRuleTbl
4163634C6F6754626C                          AccLogTbl
4163636F756E7473                            Accounts
4163637467                                  Acctg
416C6C                                      All
436F70496E666F54626C                        CopInfoTbl
```

Related Information

- For information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For details on how to disable the CHAR2HEXINT extension, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For information on implicit type conversion of UDTs, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

CHR

Returns the Latin ASCII character given a numeric code value.

CHR is an embedded services system function.

CHR Function Syntax

```
[TD_SYSFNLIB.] CHR ( numeric_expr )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

numeric_expr

numeric_expr must be zero or greater. If numeric_expr is greater than 255, an operation of *numeric_expr* mod 256 is executed to return a value between 0 and 255.

If *numeric_expr* is NULL, NULL is returned.

Argument Types and Rules

Expressions passed to this function must have a data type of NUMBER.

You can also pass arguments with data types such as BYTEINT, SMALLINT, INTEGER, or BIGINT that can be converted to NUMBER using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is CHAR(1) CHARACTER SET LATIN.

Example

The following query returns the result 'B'.

```
SELECT CHR(66);
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

CONCAT

Concatenates string expressions.

CONCAT Function Syntax

```
CONCAT ( string_expression_1, string_expression_2 [, string_expression_n ][...] )
```

Syntax Elements

string_expression_1
string_expression_2
string_expression_n

A byte, numeric, character string or string expression.

Argument Type and Rules

Use the concatenation operator on strings and string expressions of type byte, numeric, character, or UDTs that have implicit casts to a predefined character type.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

The result of a concatenation operation is a string formed by concatenating the arguments in a left-to-right direction.

Example

The result of this statement is Hello World!

```
SELECT CONCAT('Hello', 'World', '!')
```

This statement concats the first name with the last name.

```
SELECT CONCAT(fname, lname) FROM Names;
```

Related Topic

For more information, see Concatenation Operator.

CSV

CSV (Comma-Separated Value Data Unloading) returns input row column values in text format separated by a user-specified delimiter character.

CSV is an embedded services system function.

CSV Function Syntax

```
[TD_SYSFNLIB.] CSV (
  NEW VARIANT TYPE ( value [,...] ),
  delimit_string_value,
  quote_string_value
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

NEW VARIANT_TYPE *value*

A numeric or character expresssion.

NEW VARIANT_TYPE can support up to 8 row column values.

Each row column value can have a maximum of 128 columns of any supported data type.

delimit_string_value

A character expression.

A comma (,) is the default delimiter character.

quote_string_value

A character expression.

If you specify a quotation mark character, for example '"', columns defined as string data types are returned within quotation marks.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- NEW VARIANT_TYPE *value* = BYTEINT, SMALLINT, BIGINT, INTEGER, DECIMAL, FLOAT, DATE, TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, INTERVAL, CHAR, VARCHAR

  Note:

  value cannot be CLOB or GRAPHIC.

- *delimit_string_value* = VARCHAR
- *quote_string_value* = VARCHAR

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

CSV is a table function whose return value data type is VARCHAR in either the LATIN or UNICODE character set.

Examples

Example

The following query:

```
SELECT * FROM TABLE(CSV(NEW VARIANT_TYPE(dt.c1, dt.c2, dt.c3), ',', '"')
RETURNS (op varchar(64000) character set LATIN)) as t1;
```

CSV returns a string where column values are separated by commas and columns with string data types are enclosed in quotation marks.

```
SELECT * FROM TABLE (CSV(NEW VARIANT_TYPE(dt.c1, dt.c2, dt.c3), ',', '')
RETURNS (op varchar(32000) character set UNICODE)) as t1;
```

CSV returns a string where column values are separated by comma and columns with string data types are not enclosed by any characters.

Example

The following query:

```
SELECT * FROM TABLE (CSV(NEW VARIANT_TYPE(dt.c1, dt.c2, dt.c3), ',', '')
RETURNS (op varchar(32000) character set UNICODE)) as t1;
```

CSV returns a string where column values are separated by comma and columns with string data types are not enclosed by any characters.

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

CSVLD

CSVLD (Comma-Separated Value Data Loading) takes in a comma-separated string produced through the CSV table function, parses the string, and returns VARCHAR columns.

CSVLD is an embedded services system function.

CSVLD Function Syntax

```
[TD_SYSFNLIB.] CSVLD (
  data_string_value,
  delim_string_value,
  quote_string_value
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

data_string_value

A string argument.

delim_string_value

A character expression.

A comma (,) is the default delimiter character.

quote_string_value

A character expression.

The quotation mark character, for example “”, is used to indicate that the delimiter character is a part of the value, and not a field separator.

The value inside the *quote_string_value* is returned as output exclusive of the *quote_string_value* given.

Argument Types and Rules

Expressions passed to this function must have a VARCHAR data type.

You can also pass arguments with data types that can be converted to the above type using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

CSVLD is a table function can return up to 1024 VARCHAR columns in either the LATIN or UNICODE character set.

The number of output columns specified in this function must match the comma separated values in the input string.

Example

The following query:

```
SELECT * FROM TABLE (CSVLD(load_date.data, ',', '"')
RETURNS (p1 varchar(100), p2 varchar(100))) as T1;
```

CSVLD parses the comma-separated strings and returns two VARCHAR columns.

Related Information

- "Compatible Types" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

EDITDISTANCE

Returns the minimum number of edit operations (insertions, deletions, substitutions and transpositions) required to transform *string1* into *string2*.

EDITDISTANCE is an embedded services system function.

EDITDISTANCE Function Syntax

```
[TD_SYSFNLIB.] EXITDISTANCE ( string1, string2 [, ci, cd, cs, ct ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

string1

A character string or string expression.

string2

A character string or string expression.

ci

This syntax element cannot be a negative value.

If not specified, a default value of 1 is used.

cd

This syntax element cannot be a negative value.

If not specified, a default value of 1 is used.

cs

This syntax element cannot be a negative value.

If not specified, a default value of 1 is used.

ct

This syntax element cannot be a negative value.

If not specified, a default value of 1 is used.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *string1* = CHAR, VARCHAR, or CLOB
- *string2* = CHAR, VARCHAR, or CLOB
- *ci* = INTEGER
- *cd* = INTEGER
- *cs* = INTEGER
- *ct* = INTEGER

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is INTEGER.

Usage Notes

EDITDISTANCE measures the similarity between two strings. A low number of deletions, insertions, substitutions or transpositions implies a high similarity. The insertions, deletions, substitutions, and transpositions are based on the Damerau-Levenshtein Distance algorithm with modifications for costed operations.

If either *string1* or *string2* is NULL, the function returns NULL.

Examples

Example

The following query returns a result of 9.

```
SELECT EDITDISTANCE('Jim D. Swain', 'John Smith');
```

The following query returns a result of 0 since the strings are the same.

```
SELECT EDITDISTANCE('John Smith', 'John Smith');
```

The following query returns a result of 9.

```
SELECT EDITDISTANCE('Jim D. Swain', 'John Smith', 2, 1, 1, 2);
```

The following query returns a result of 11.

```
SELECT EDITDISTANCE('John Smith', 'Jim D. Swain', 2, 1, 1, 2);
```

Example

The following query returns a result of 0 since the strings are the same.

```
SELECT EDITDISTANCE('John Smith', 'John Smith');
```

Example

The following query returns a result of 9.

```
SELECT EDITDISTANCE('Jim D. Swain', 'John Smith', 2, 1, 1, 2);
```

Example

The following query returns a result of 11.

```
SELECT EDITDISTANCE('John Smith', 'Jim D. Swain', 2, 1, 1, 2);
```

Related Information

For more information on activating and invoking embedded services functions, see Embedded Services System Functions.

INDEX

Returns the position in *string_expression_1* where *string_expression_2* starts.

INDEX Function Syntax

```
INDEX ( string_expression_1, string_expression_2 )
```

Syntax Elements

string_expression_1

A full string to be searched.

string_expression_2

A substring to be searched for its position within the full string.

Argument Types and Rules

INDEX operates on the following types of arguments:

- Character
- Byte

  If one string expression is of type BYTE, then both string expressions must be of type BYTE.
- Numeric

  If any string expression is numeric, then it is converted implicitly to CHARACTER type.
- UDTs that have implicit casts that cast between the UDT and any of the following predefined types:
  - Numeric
  - Character

- DATE
- Byte

  To define an implicit cast for a UDT, use CREATE CAST and specify AS ASSIGNMENT. For details on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

  Implicit type conversion of UDTs for system operators and functions, including INDEX, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

INDEX does not support CLOBs or BLOBs.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Use POSITION instead of INDEX for ANSI SQL:2011 compliance.

Result Type and Attributes

Here are the default result type and attributes for INDEX(arg1, arg2).

Data Type	Heading
INTEGER	Index(arg1, arg2)

INDEX Usage Notes

Expected Values

The following rules apply to the value that INDEX returns:

- If *string_expression_2* is not found in *string_expression_1*, then the result is zero.
- If *string_expression_2* is null, then the result is null.
- If the arguments are character types, INDEX returns a logical character position, not a byte position, except when the server character set of the arguments is KANJI1 and the session client character set is KanjiEBCDIC.

Rules for Character Type Arguments

If the arguments are character types, matching is in terms of logical characters. Single byte characters are matched against single byte characters, and multibyte characters are matched against multibyte characters. For a match to occur, representation of the logical character must be identical in both expressions.

If the server character sets of the arguments are not the same, INDEX performs an implicit character translation.

The CASESPECIFIC attribute affects whether characters are considered to be a match.

IF the session mode is …	THEN the default case specification for character columns and literals is …
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. The exception is character data of type GRAPHIC, which is always CASESPECIFIC.

To override the default case specification, you can apply the CASESPECIFIC or NOT CASESPECIFIC phrase to a character column in CREATE TABLE or ALTER TABLE.

Or, you can apply the CASESPECIFIC or NOT CASESPECIFIC phrase to the INDEX character string arguments.

IF …	THEN …
either argument has a CASESPECIFIC attribute (either by default or specified explicitly)	simple Latin letters are considered to be matching only if they are the same letters and the same case.
both arguments have a NOT CASESPECIFIC attribute (either by default or specified explicitly)	before the operation begins, some characters are converted to uppercase. If the character is a lowercase simple Latin letter, the character is converted to uppercase before the operation begins. If the character is a non-Latin single byte character, a multibyte character, or a byte indicating a transition between single-byte and multibyte character data, the character is not converted to uppercase.

Using the rules for character type arguments, if you want INDEX to match letters only if they are the same letters in the same case, specify the CASESPECIFIC phrase with at least one of the arguments. For example:

```
SELECT Name
FROM Employee
WHERE INDEX(Name, 'X' (CASESPECIFIC)) = 1;
```

If you want INDEX to match letters without considering the case, specify the NOT CASESPECIFIC phrase with both of the arguments.

Rules for KANJI1 Server Character Set

Note:

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see *KANJI1 Character Set* in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

When the server character set is KANJI1 and the client character set is KanjiEBCDIC, the offset count includes Shift-Out/Shift-In characters, but they are not matched. They are treated only as an indication of a transition from a single byte character and an multibyte character.

The nonzero position of the result is reported as follows:

IF the character set is …	THEN the result is the …
KanjiEBCDIC	position of the first byte of the logical character offset (including Shift-Out /Shift-In in the offset count) within string_expression_1.
other than KanjiEBCDIC	logical character offset within string_expression_1.

Relationship Between INDEX and POSITION

INDEX and POSITION behave identically, except on character type arguments when the client character set is KanjiEBCDIC, the server character set is KANJI1, and an argument contains a multibyte character.

Examples

Example: Using Simple INDEX Expressions

The following table shows examples of simple INDEX expressions and their results.

Expression	Result
INDEX('catalog','log')	5
INDEX('catalog','dog')	0
INDEX('41424344'XB,'43'XB)	3

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for string_1 and the server character set for *string_2* differ. In these cases, both arguments are converted to UNICODE (if needed) and the characters are matched logically.

IF string_1 is …		AND string_2 is …		THEN the result is …
Character Set	Data	Character Set	Data	
UNICODE	92 年 abc	LATIN	abc	4
UNICODE	abc	UNICODE	c	3
KANJISJIS	92年 04	UNICODE	0	4

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for both arguments is KANJI1 and the client character set is KanjiEBCDIC.

Note that for KanjiEBCDIC, results are returned in terms of physical units, making INDEX DB2-compliant in that environment.

IF string_1 contains …	AND string_2 contains …	THEN the result is …
MN<AB >	<B >	6
MN<AB >	<A >	4
MN<AB >P	P	9
MX N<AB >P	<B >	7

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for both arguments is KANJI1 and the client character set is KanjiEUC.

IF string_1 contains …	AND string_2 contains …	THEN the result is …
a b ss3 A	ss3 A	3
a b ss2 B	ss2 B	3
CS1_DATA	A	6
a b ss2 D ss3 E ss2 F	ss2 F	5
a b C ss2 D ss3 E ss2 F	ss2 F	6
CS1_D mATA	A	7

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for both arguments is KANJI1 and the client character set is KanjiShift-JIS.

IF string_1 contains …	AND string_2 contains …	THEN the result is …
mnABC X	B	4
mnABC X	X	6

In this example, INDEX is applied to ' ' (the SPACE character) in the value strings in the Name column of the Employee table.

```
SELECT name
FROM employee
WHERE INDEX(name, ' ') > 6 ;
```

INDEX examines the Name field and returns all names where a space appears in a character position beyond the sixth (character position seven or higher).

The following example displays a list of projects in which the word Batch appears in the project description, and lists the starting position of the word.

```
SELECT proj_id, INDEX(description, 'Batch')
FROM project
WHERE INDEX(description, 'Batch') > 0 ;
```

The system returns the following report.

```
proj_id       Index (description, 'Batch')
------------- ----------------------------
OE2-0003                                 5
AP2-0003                                13
OE1-0003                                 5
AP1-0003                                13
AR1-0003                                10
AR2-0003                                10
```

A somewhat more complex construction employing concatenation, SUBSTRING, and INDEX might be more instructive. Suppose the employee table contains the following values.

```
empno      name
---------- -----------
10021      Smith T
10007      Aguilar J
10018      Russell S
10011      Chin M
10019      Newman P
```

You can transpose the form of the names from the name column selected from the employee table and change the punctuation in the report using the following query:

```
   SELECT empno,
   SUBSTRING(name FROM INDEX(name,' ')+1 FOR 1)||
'. '||
   SUBSTRING(name FROM 1 FOR INDEX(name, ' ')-1)
   (TITLE 'Emp Name')
   FROM employee ;
```

The system returns the following report.

```
   empno      Emp Name
   ---------- --------------
   10021      T. Smith
   10007      J. Aguilar
   10018      S. Russell
   10011      M. Chin
   10019      P. Newman
```

Example

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for *string_1* and the server character set for *string_2* differ. In these cases, both arguments are converted to UNICODE (if needed) and the characters are matched logically.

IF string_1 is …		AND string_2 is …		THEN the result is …
Character Set	Data	Character Set	Data	
UNICODE	92年 abc	LATIN	abc	4
UNICODE	abc	UNICODE	c	3
KANJISJIS	92年 04	UNICODE	0	4

Example: Using INDEX with KANJI1 and KanjiEBCDIC

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for both arguments is KANJI1 and the client character set is KanjiEBCDIC.

Note that for KanjiEBCDIC, results are returned in terms of physical units, making INDEX DB2-compliant in that environment.

IF string_1 contains …	AND string_2 contains …	THEN the result is …
MN<AB>	<B>	6
MN<AB>	<A>	4
MN<AB>P	P	9
MX N<AB>P	<B>	7

Example: Using INDEX with KANJI1 and KanjiEUC

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for both arguments is KANJI1 and the client character set is KanjiEUC.

IF string_1 contains …	AND string_2 contains …	THEN the result is …
a b ss3 A	ss3 A	3
a b ss2 B	ss2 B	3
CS1_DATA	A	6
a b ss2 D ss3 E ss2 F	ss2 F	5
a b C ss2 D ss3 E ss2 F	ss2 F	6
CS1_D mATA	A	7

Example: Using INDEX with KANJI1 and KanjiShift-JIS

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for both arguments is KANJI1 and the client character set is KanjiShift-JIS.

IF string_1 contains …	AND string_2 contains …	THEN the result is …
mnABC X	B	4
mnABC X	X	6

Example: Applying INDEX to the SPACE Character

In this example, INDEX is applied to ' ' (the SPACE character) in the value strings in the Name column of the Employee table.

```
SELECT name
FROM employee
WHERE INDEX(name, ' ') > 6 ;
```

INDEX examines the Name field and returns all names where a space appears in a character position beyond the sixth (character position seven or higher).

Example: Using "Batch" in the Project Description

The following example displays a list of projects in which the word Batch appears in the project description, and lists the starting position of the word.

```
SELECT proj_id, INDEX(description, 'Batch')
FROM project
WHERE INDEX(description, 'Batch') > 0 ;
```

The system returns the following report.

```
proj_id       Index (description, 'Batch')
------------- ----------------------------
OE2-0003                                 5
AP2-0003                                13
OE1-0003                                 5
AP1-0003                                13
AR1-0003                                10
AR2-0003                                10
```

Example: Using Concatenation, SUBSTRING, and INDEX

A somewhat more complex construction employing concatenation, SUBSTRING, and INDEX might be more instructive. Suppose the employee table contains the following values.

```
empno      name
---------- -----------
10021      Smith T
10007      Aguilar J
10018      Russell S
10011      Chin M
10019      Newman P
```

You can transpose the form of the names from the name column selected from the employee table and change the punctuation in the report using the following query:

```
   SELECT empno,
   SUBSTRING(name FROM INDEX(name,' ')+1 FOR 1)||
'. '||
   SUBSTRING(name FROM 1 FOR INDEX(name, ' ')-1)
   (TITLE 'Emp Name')
   FROM employee ;
```

The system returns the following report.

```
   empno      Emp Name
   ---------- --------------
   10021      T. Smith
   10007      J. Aguilar
   10018      S. Russell
   10011      M. Chin
   10019      P. Newman
```

Related Information

- For information on implicit type conversion, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For an example of when the two functions return different results for the same data, see How POSITION and INDEX Differ.
- For details, see Rules for KANJI1 Server Character Set.
- For a description of implicit character translation rules, see "Implicit Character-to-Character Translation" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

INITCAP

Modifies a string argument and returns the string with the first character in each word in uppercase and all other characters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

INITCAP is an embedded services system function.

INITCAP Function Syntax

```
[TD_SYSFNLIB.] INITCAP ( string )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

string

A character string or string expression.

If *string* is NULL, NULL is returned.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

CHAR, VARCHAR, or CLOB

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type and character set are the same as those of the input string. For example, if the input string has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Example

The following query returns the result 'Hello World'.

```
SELECT INITCAP('hello WORLD');
```

Related Information

- "Compatible Types" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

- To activate and invoke embedded services functions, see Embedded Services System Functions.

INSTR

Searches the *source_string* argument for occurrences of *search_string*.

INSTR is an embedded services system function.

INSTR Function Syntax

```
[TD_SYSFNLIB.] INSTR (
  source_string,
  search_string
  [, position [, occurrence] ]
)
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character string or string expression.

search_string

A string of characters that the function searches for in *source_string*.

position

Specifies that the search will begin at this position in *source_string*.

If *position* is not specified, the search starts at the beginning of *source_string*.

If *position* is negative, the function counts and searches backwards from the end of *source_string. position* cannot have a value of zero.

occurrence

Specifies which occurrence of *search_string* to find in *source_string*.

If *occurrence* is not specified, the function searches for the first occurrence.

If *occurrence* is greater than 1, the function searches for additional occurrences beginning with the second character in the previous occurrence. *occurrence* cannot be zero or a negative value.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR, or CLOB
- *search_string* = CHAR, VARCHAR, or CLOB
- *position* = INTEGER, BIGINT, or NUMBER
- *occurrence* = INTEGER, BIGINT, or NUMBER

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

If a match is found, the function returns the position (starting from 1) in the *source_string* for the match; otherwise the function returns 0. If any of the input arguments are NULL, the function returns NULL.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is NUMBER.

Usage Notes

This function is CASESPECIFIC.

REGEXP_INSTR is an extended version of INSTR that can perform case-insensitive string matching.

Examples

Example

The following query returns the result 20 indicating the position of 'ch' in 'chip'. This is the second occurrence of 'ch' with the search starting from the second character of the source string.

```
SELECT INSTR('choose a chocolate chip cookie','ch',2,2);
```

Example

The following query returns the result 2, which indicates the position of the first occurrence of 'N' in the source string with the search starting from the beginning of the string.

```
SELECT INSTR('INSTR FUNCTION','N');
```

Related Information

For information on activating and invoking embedded services functions, see Embedded Services System Functions.

LEFT

Truncates in input string to a specified number of characters.

The LEFT function can be called with the 'LEFT' or 'TD_LEFT' alias names.

The arguments include two parameters:

- First Parameter

  The input string that the substring is created from. CHAR, VARCHAR, and CLOB are supported, but other types must be explicitly cast. The input can be null or an empty string.
- Second Parameter

  A positive integer specifying the number of characters desired from the left side of the string. If the number of character exceeds the number of characters in the original string, the original string is returned.

LEFT Function Syntax

```
[TD_SYSFNLIB.] LEFT ( source_string, length )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character string or string expression.

If *source_string* is NULL, NULL is returned.

length

An integer specifying the length of the returned string.

Usage Notes

When submitting queries using function LEFT on a client that uses ODBC, disable ODBC parsing. Otherwise, the query may be altered by ODBC to a SUBSTR function call.

Character Sets Supported

The following character sets are supported with the LEFT function:

- Unicode
- Latin
- Kanji SJIS
- Graphic

Result Type

A substring is returned. The return type is set to the input type with the exception of CHAR. A CHAR input has a result type of VARCHAR.

The result character set is the same as the *source_string* character set for Unicode and Latin. The result character set is Unicode for all other supported character sets.

Example

```
SELECT LEFT('Test String',6);
Result- 'Test S'
```

LENGTH

Returns the number of characters in the expression.

LENGTH is an embedded services system function.

LENGTH Function Syntax

```
[TD_SYSFNLIB.] LENGTH ( expr )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

expr

A character string or string expression.

If *expr* is NULL, NULL is returned.

A character string, string expression, or numeric expression.

If *expr* is NULL, NULL is returned.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

CHAR, VARCHAR, or CLOB

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is NUMBER.

Example

The following query returns the result 7.

```
SELECT LENGTH('astring');
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

LOCATE

Returns the position of the first occurrence of *string_expr1* within *string_expr2*. The search for the first occurrence of *string_expr1* begins with the first character position in *string_expr2* unless the optional argument, *n1*, is specified.

LOCATE Function Syntax

```
LOCATE ( string_expr1, string_expr2 [, n1 ] )
```

Syntax Elements

string_expr1

Substring to be searched for its position within the full string.

string_expr2

The full string to be searched.

n1

The search begins with the character position indicated by the value of *n1*.

Argument Types and Rules

For information, see POSITION.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

For information, see POSITION.

Examples

Examples of LOCATE function usage:

```
SELECT LOCATE ('world', 'Hello world!');
```

```
SELECT LOCATE ('world', 'Hello world!', 4);
```

LOWER

Returns a character string identical to *character_string_expression*, except that all uppercase letters are replaced with their lowercase equivalents.

LOWER is valid only for character strings or character string expressions. The function does not accept CLOBs and non-character arguments.

LOWER Function Syntax

```
LOWER ( character_string_expression )
```

Syntax Elements

character_string_expression

A character string or character string expression for which all uppercase characters are to be replaced with their lowercase equivalents.

Argument Types

LOWER is valid only for character strings or character string expressions. The function does not accept CLOBs and non-character arguments.

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including LOWER, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

Result Type and Attributes

Here are the default result type and attributes for LOWER(*arg*).

Data Type	Heading
Same type as *arg*	Lower(*arg*)

The LOWER function returns the result in the same character set as the input argument. The exception is when the input is KANJI1 data; LOWER returns the result in the LATIN server character set.

Note:

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see *KANJI1 Character Set* in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Usage Notes

See *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125 for the internal mappings that Vantage uses for the LOWER function.

Teradata SQL has the type attribute NOT CASESPECIFIC that allows case blind comparisons, but the type attributes CASESPECIFIC and NOT CASESPECIFIC are Teradata extensions to the ANSI standard.

For ANSI portability, use the UPPER function for case blind comparisons with ANSI-compliant syntax.

UDT Arguments

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including LOWER, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

Example

The use of LOWER to return and store values is shown in the following example.

```
SELECT LOWER (last_name)
FROM names;
INSERT INTO names
SELECT LOWER(last_name),LOWER(first_name)
FROM newnames;
```

The identical result is achieved with a USING phrase.

```
USING (last_name CHAR(20),first_name CHAR(20))
INSERT INTO names (LOWER(:last_name), LOWER(:first_name));
```

Related Information

- For information on implicit type conversion of UDTs, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143 .
- For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For details, see UPPER/UCASE.

LPAD

Returns the *source_string* padded to the left with the characters in *fill_string* so that the resulting string is *length* characters.

LPAD is an embedded services system function.

LPAD Function Syntax

```
[TD_SYSFNLIB.] LPAD ( source_string, length [, fill_string ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character string or string expression.

If the length of *source_string* is greater than *length*, *source_string* is truncated to *length* characters.

length

The number of characters in the resulting string.

fill_string

The string of characters used to pad the *source_string*.

The sequence of characters in *fill_string* is replicated as necessary.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *source_string* = VARCHAR or CLOB
- *length* = INTEGER, BIGINT, or NUMBER
- *fill_string* = CHAR, VARCHAR, or CLOB

If any of the input arguments are NULL, the function returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type and character set are the same as those of the *source_string* argument. For example, if the *source_string* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Examples

Example

The following query returns the result 'yzybuilding'.

```
SELECT LPAD('building', 11, 'yz');
```

The following query returns the result 'build'.

```
SELECT LPAD('building', 5, 'yz');
```

The following query returns the result '   building'. The space character is used by default to pad the source string.

```
SELECT LPAD('building', 11);
```

Example

The following query returns the result 'build'.

```
SELECT LPAD('building', 5, 'yz');
```

Example

The following query returns the result '   building'. The space character is used by default to pad the source string.

```
SELECT LPAD('building', 11);
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

LTRIM

Returns the argument *expr1*, with its left-most characters removed up to the first character that is not in the argument *expr2*.

LTRIM is an embedded services system function.

LTRIM Function Syntax

```
[TD_SYSFNLIB.] LTRIM ( expr1 [, expr2 ] )
```

Syntax Elements

TD_SYSFNLIB.
: Name of the database where the function is located.

expr1
: A character string or string expression. If *expr1* is NULL, NULL is returned

expr2

A string of characters or a numeric expression that will be removed from *string1*. If *expr2* is specified, it must be the same datatype as *expr1*. If *expr2* is not specified, the default is to use a single space character.

Argument Types and Rules

Expressions passed to this function can be one of the following data types:

- CHAR
- VARCHAR
- CLOB
- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- FLOAT/REAL/DOUBLE PRECISION
- DECIMAL/NUMERIC
- NUMBER

If any of the input arguments are NULL, the function returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type and character set are the same as those of the *expr1* argument. For example, if the *expr1* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Examples

Example

The following query returns the result 'XxyLAST WORD'. LTRIM removes the individual occurrences of 'x' and 'y', and it stops removing characters when it encounters 'X' because 'X' is not in the *expr2* argument.

```
SELECT LTRIM('xyxXxyLAST WORD','xy');
```

The following query returns the result 'LEFT TRIM'. LTRIM removes the spaces from the left, and it stops removing characters when it encounters 'L' because 'L' is not in the *expr2* argument. Since the *expr2* argument is not explicitly specified, the default of a single space is used.

```
SELECT LTRIM('  LEFT TRIM');
```

Example

The following query returns the result 'LEFT TRIM'. LTRIM removes the spaces from the left, and it stops removing characters when it encounters 'L' because 'L' is not in the *string2* argument. Since the *string2* argument is not explicitly specified, the default of a single space is used.

```
SELECT LTRIM('  LEFT TRIM');
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

NGRAM

Returns the number of n-gram matches between *string1* and *string2*.

A high number of matching n-gram patterns implies a high similarity between the two strings.

NGRAM is an embedded services system function.

NGRAM Function Syntax

```
[TD_SYSFNLIB.] NGRAM ( string1, string2, length [, position ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

string1

A character string or string expression.

If *string1* is NULL, NULL is returned.

string2

A character string or string expression.

If *string2* is NULL, NULL is returned.

length

The value *ninn-gram*, which is the comparison length.

position

Specifies that the n-gram is a positional n-gram match.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *string1* = CHAR, VARCHAR, or CLOB
- *string2* = CHAR, VARCHAR, or CLOB
- *length* = INTEGER
- *position* = INTEGER

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

If the data type of *string1* is CHAR or VARCHAR, the result data type is INTEGER.

If the data type of *string1* is CLOB, the result data type is BIGINT.

Usage Notes

For positional n-gram matching, the position as well as the pattern must match when measuring similarity. The *position* value indicates how far away positionally the match may be between the 2 strings as follows:

- If *position* is set to a value of zero, the match must be at the same position in the 2 strings.
- If *position* is set to a value of *x* , the match must be within *x* positions in the 2 strings. For example, if *position* = 2, then the match must be within 2 positions in the 2 strings.

As an example, for a string of 'abc', the 1-grams (length =1) are 'a', 'b', and 'c'. The 2-grams (length =2) are 'ab' and 'bc'. The 3-gram (length = 3) is 'abc'. By definition, there are no 4-grams or greater.

The function returns zero in the following cases:

- If the *length* argument is greater than the length of either *string1* or *string2*.
- If the *length* argument is <= 0 or if either *string1* or *string2* is an empty string.

Patterns beyond the length of 255 are ignored.

Examples

Example

The following query returns a result of 2. The 3-grams 'mit' and 'ith' match. Note that 'Smi' and 'smi' do not match because of the difference in case.

```
SELECT NGRAM('John Smith','Allen smith 1',3);
```

The following query returns a result of zero. There are no 3-grams in the first string expression of '' since the length of the string is less than 3.

```
SELECT NGRAM ('','str1 empty',3);
```

The following query returns a result of zero. There are no 0-grams in the strings.

```
SELECT NGRAM ('test with zero length', 'test with zero length',0);
```

The following query returns a result of 3. The 1-grams 'a', 'b', and 'c' match.

```
SELECT NGRAM ('abc','yyabc',1);
```

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match.

```
SELECT NGRAM ('abc','yyabc',2);
```

The following query returns a result of zero. The 2-grams 'ab' and 'bc' match, but they are not within 1 position of each other.

```
SELECT NGRAM ('abc','yyabc',2, 1);
```

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match, and they are within 2 positions of each other.

```
SELECT NGRAM ('abc','yyabc',2, 2);
```

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match, and they are at the same position in each string.

```
SELECT NGRAM ('abc','abc',2, 0);
```

The following query returns a result of zero. There are no 5-grams since the length of either input string is less than 5.

```
SELECT NGRAM ('abc','abc',5,0);
```

Example

The following query returns a result of zero. There are no 3-grams in the first string expression of '' since the length of the string is less than 3.

```
SELECT NGRAM ('','str1 empty',3);
```

Example

The following query returns a result of zero. There are no 0-grams in the strings.

```
SELECT NGRAM ('test with zero length', 'test with zero length',0);
```

Example

The following query returns a result of 3. The 1-grams 'a', 'b', and 'c' match.

```
SELECT NGRAM ('abc','yyabc',1);
```

Example

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match.

```
SELECT NGRAM ('abc','yyabc',2);
```

Example

The following query returns a result of zero. The 2-grams 'ab' and 'bc' match, but they are not within 1 position of each other.

```
SELECT NGRAM ('abc','yyabc',2, 1);
```

Example

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match, and they are within 2 positions of each other.

```
SELECT NGRAM ('abc','yyabc',2, 2);
```

Example

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match, and they are at the same position in each string.

```
SELECT NGRAM ('abc','abc',2, 0);
```

Example

The following query returns a result of zero. There are no 5-grams since the length of either input string is less than 5.

```
SELECT NGRAM ('abc','abc',5,0);
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

NVP

Extracts the value of a name-value pair where the name in the pair matches the name and the number of the occurrence specified.

NVP is an embedded services system function.

NVP Function Syntax

```
[TD_SYSFNLIB.] NVP (
  instring,
  name_to_search
  [, name_delimiters ]
  [, value_delimiters ]
  [, occurrence ]
)
```

Syntax Elements

TD_SYSFNLIB

Name of the database where the function is located.

instring

The name-value pairs separated by multibyte delimiters.

name_to_search

The name whose instring value NVP returns.

name_delimiters

The multibyte delimiters used to separate name-value pairs.

Delimiters can contain any characters. They are separated from each other in the string by spaces. If a space is used as part of a delimiter, it must be escaped using a backslash (\). The maximum length of any delimiter is 10, and the maximum size of this parameter is 32.

This parameter is optional and if not specified, the default value is & (ampersand).

value_delimiters

The multibyte delimiters used to associate a name to its value in a name-value pair.

Delimiters can contain any characters. They are separated from each other in the string by spaces. If a space is used as part of a delimiter, it must be escaped using a backslash (\). The maximum length of any delimiter is 10, and the maximum size of this parameter is 32

This parameter is optional and if not specified, the default value is = (equal sign).

occurrence

The number of occurrences of name_to_search that NVP searches for.

This parameter is optional and if not specified, the default value is 1.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- instring = VARCHAR or CLOB
- name_to_search = VARCHAR
- name_delimiters = VARCHAR
- value_delimiters = VARCHAR
- occurrence = INTEGER

The character set of *instring*, *name_to_search*, *name_delimiters*, and *value_delimiters* can be LATIN or UNICODE. If the parameter character sets are mixed, then all the parameters are converted to UNICODE.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

If a delimiter is part of a longer delimiter, the longer delimiter has precedence in the matching process.

Adjacent delimiters are treated as a single delimiter.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is VARCHAR with the same character set as that of instring.

Examples

Example: Querying for Entree

The following query:

```
SELECT NVP ('entree:orange chicken#entree2:honey salmon', 'entree','#', ':', 1);
```

returns 'orange chicken'.

Example: Querying for Second Occurrence of 'store'

The following query:

```
SELECT NVP('store = whole foods&&store: ?Bristol farms','store', '&&', '\ =\
:\ ?', 2);
```

returns 'Bristol farms'.

In this example, *occurrence* = 2 instructs NVP to search for the second occurrence of 'store'.

Example: Querying for Entree (Default Value)

The following query:

```
SELECT NVP('entree=orange chicken&entree2=honey salmon', 'entree', 1)
```

returns 'orange chicken'.

In this example, *name_delimiters* is & (default value) and *value_delimiters* is = (default value).

Example: Querying for Entree with 1 Occurrence

The following query:

```
SELECT NVP('entree=orange chicken&entree2=honey salmon', 'entree');
```

returns 'orange chicken'.

In this example, *name_delimiters* is & (default value), *value_delimiters* is = (default value), and occurrence is 1 (default value).

Related Information

- To activate and invoke embedded services functions, see Embedded Services System Functions.
- For details about UDF implicit type conversion rules, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

OREPLACE

Replaces every occurrence of *search_string* in the *source_string* with the *replace_string*. Use this function either to replace or remove portions of a string.

OREPLACE is an embedded services system function.

OREPLACE Function Syntax

```
[TD_SYSFNLIB.] OREPLACE ( source_string, search_string [, replace_string ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character string or string expression.

If *source_string* is NULL, NULL is returned.

search_string

A string of characters that the function searches for in *source_string*.

If *search_string* is NULL, NULL is returned.

replace_string

A string of characters that replaces the characters specified by *search_string*.

If *replace_string* is NULL or is an empty string, or is omitted, all occurrences of *search_string* are removed from the *source_string*.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- CHAR
- VARCHAR
- CLOB

If Vantage passes constants as the second and third parameter in an OREPLACE call, the character type of the first argument is passed as Unicode, and calls oreplace_unicode() with the return type VARCHAR in Unicode charset.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type and character set depend on those of the *source_string* argument.

- If the *source_string* is CHAR, the result data type is VARCHAR.
- If the *source_string* is VARCHAR, the result data type is VARCHAR.
- If the *source_string* is CLOB, the result data type is CLOB.

For example, if the *source_string* argument has a data type of CHAR CHARACTER SET UNICODE, then the result data type will be VARCHAR CHARACTER SET UNICODE.

The maximum length of a VARCHAR result value is 16000 characters for LATIN, and 8000 characters for UNICODE.

An error is returned if the result string is larger than the maximum result string size.

Usage Notes

OREPLACE provides a superset of the functionality provided by the OTRANSLATE function. OTRANSLATE provides single character, 1-to-1 substitution while OREPLACE allows you to substitute 1 string for another, as well as to remove character strings.

Examples

Example

The following query returns the string 'TD14.0 is the current version'. The string '13.1' in the source string was replaced by the string '14.0'.

```
SELECT OREPLACE('TD13.1 is the current version', '13.1', '14.0');
```

The following query returns the string 'This chair is a brown chair'. Both occurrences of the search string 'bag' in the source string were replaced by the string 'chair'.

```
SELECT OREPLACE('This bag is a brown bag', 'bag', 'chair');
```

The following query returns the string 'TD13.1 is the current version'. The source string is returned unchanged since the search string is NULL. The result would be the same if the search string is an empty string or a string that has no matches in the source string.

```
SELECT OREPLACE('TD13.1 is the current version', NULL, '14.0');
```

The following query returns the string 'We removed the extra word'. The occurrence of the search string 'superfluous' was removed from the source string.

```
SELECT OREPLACE('We removed the superfluous extra word',
   'superfluous', NULL);
```

The result set from the following query will have an ADDRESS column that is the concatenation of the ADDRESS1 and ADDRESS2 columns from the CUSTOMER table, with every occurrence of 'st.' replaced with ' street'.

```
SELECT OREPLACE(ADDRESS1||ADDRESS2, ' st.', ' street') AS ADDRESS
   from CUSTOMER;
```

Example

The following query returns the string 'This chair is a brown chair'. Both occurrences of the search string 'bag' in the source string were replaced by the string 'chair'.

```
SELECT OREPLACE('This bag is a brown bag', 'bag', 'chair');
```

Example

The following query returns the string 'TD13.1 is the current version'. The source string is returned unchanged since the search string is NULL. The result would be the same if the search string is an empty string or a string that has no matches in the source string.

```
SELECT OREPLACE('TD13.1 is the current version', NULL, '14.0');
```

Example

The following query returns the string 'We removed the extra word'. The occurrence of the search string 'superfluous' was removed from the source string.

```
SELECT OREPLACE('We removed the superfluous extra word',
   'superfluous', NULL);
```

Example

The result set from the following query will have an ADDRESS column that is the concatenation of the ADDRESS1 and ADDRESS2 columns from the CUSTOMER table, with every occurrence of 'st.' replaced with ' street'.

```
SELECT OREPLACE(ADDRESS1||ADDRESS2, ' st.', ' street') AS ADDRESS
   from CUSTOMER;
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

OTRANSLATE

Returns *source_string* with every occurrence of each character in *from_string* replaced with the corresponding character in *to_string*.

OTRANSLATE is an embedded services system function.

OTRANSLATE Function Syntax

```
[TD_SYSFNLIB.] OTRANSLATE ( source_string, from_string, to_string )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character string or string expression.

If *source_string* is NULL, NULL is returned.

from_string

A string of characters that will be replaced in *source_string*.

If *from_string* is NULL, the function returns *source_string*.

to_string

A string of characters that replaces the characters specified by *from_string*.

If *to_string* is NULL or empty, the function removes the characters specified in *from_string*.

Argument Types and Rules

Expressions passed to this function must have one of the following data types: CHAR or VARCHAR.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is VARCHAR. The character set is the same as that of the *source_string* argument.

Usage Notes

If the first character in *from_string* occurs in the *source_string*, all occurrences of it are replaced by the first character in *to_string*. This repeats for all characters in *from_string* and for all characters in *to_string*. The replacement is performed character-by-character, that is, the replacement of the second character is done on the string resulting from the replacement of the first character.

If *from_string* contains more characters than *to_string*, the extra characters are removed from the *source_string*.

If *from_string* contains fewer characters than *to_string*, the extra characters in *to_string* have no effect.

If the same character occurs more than once in *from_string*, only the replacement character from the *to_string* corresponding to the first occurrence is used.

Examples

Example: Returning the Current Database Version

The following query returns the string 'TD14.0 is the current database version'. The occurrence in *source_string* of the character in *from_string* ('3') is replaced by the character in *to_string* ('4').

```
SELECT OTRANSLATE('TD13.0 is the current database version',
   '3', '4');
```

Example: Removing Extra Characters from the Query Results

In the following query, the characters 'T' and 'h' are replaced with 'S' and 'p', resulting in the string 'Spin and Spick'. Next, the extra character 'k' in the *from_string* (where there is no corresponding character in the *to_string*) is removed from the *source_string*. The resulting string is 'Spin and Spic'.

```
SELECT OTRANSLATE('Thin and Thick', 'Thk', 'Sp');
```

Example: Replacing and Returning Query Characters

In the following query, the characters 'T' and 'h' are replaced with 'S' and 'p'. The character 'T' occurs twice in the *from_string*, but only the replacement character in the *to_string* that corresponds to the first occurrence of 'T' is used. That is, only 'S' is used to replace 'T'. Next, the character 'k' is replaced with 'x', and the extra characters 'y' and 'z' in the *to_string* are ignored. The resulting string is 'Spin and Spicx'.

```
SELECT OTRANSLATE('Thin and Thick', 'ThTk', 'Sptxyz');
```

Example: Removing Characters without Replacing Them

The following query removes the *from_string* from *source_string* because *to_string* is empty.

```
SELECT OTRANSLATE('diet', 't', '');

 *** Query completed. One row found. One column returned.
 *** Total elapsed time was 1 second.

OTRANSLATE('diet','t','')
---------------------------------------------------------------------------
die
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

POSITION

Returns the position in *string_expression_2* where *string_expression_1* starts.

POSITION Function Syntax

```
POSITION ( string_expression_1 IN string_expression_2 )
```

Syntax Elements

string_expression_1
A substring to be searched for its position within the full string.

string_expression_2
A full string to be searched.

Argument Types and Rules

POSITION operates on the following types of arguments:

- Character, except for CLOB
- Byte, except for BLOB

If one string expression is of type BYTE, then both expressions must be of type BYTE.

- Numeric

  Numeric string expressions are converted implicitly to CHARACTER type.
- UDTs that have implicit casts that cast between the UDT and any of the following predefined types:
  - Numeric
  - Character
  - DATE
  - Byte

    To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

    Implicit type conversion of UDTs for system operators and functions, including POSITION, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Use POSITION instead of INDEX for ANSI SQL:2011 conformance. POSITION and INDEX behave identically except when the client character set is KanjiEBCDIC and the server character for an argument is KANJI1 and contains multibyte characters.

Use POSITION in place of MINDEX. (MINDEX no longer appears in this book because its use is deprecated and it will not be supported after support for KANJI1 is dropped.)

Result Type and Attributes

Here are the default result type and attributes for POSITION(*arg1* IN *arg2*):

Data Type	Heading
INTEGER	Position(*arg1* in *arg2*)

POSITION Usage Notes

Expected Values

POSITION returns a value according to the following rules.

IF …	THEN the result is …
either argument is null	null.
string_expression_1 has length zero	one.
string_expression_1 is a substring within *string_ expression_2*	the position in *string_expression_2* where *string_expression_1* starts.
none of the preceding is true	zero.

If the arguments are character types, then regardless of the server character set, the value for POSITION represents the position of a logical character, not a byte position.

How POSITION and INDEX Differ

INDEX and POSITION behave identically except when the session client character set is KanjiEBCDIC, the server character set is KANJI1, and the parent string contains a multibyte character.

This is the only case for which the results of these two functions differ when performed on the same data.

Suppose we create the following table.

```
CREATE TABLE iptest (
 column_1 VARCHAR(30) CHARACTER SET Kanji1
 column_2 VARCHAR(30) CHARACTER SET Kanji1);
```

We then insert the following set of values for the columns.

column_1	column_2
MN<AC>	<C>
MN<AC>P	<A>
MN<AB>P	P
MN<AB>P	<B>

The client session character set is KanjiEBCDIC5026_0I. Now we perform a query that demonstrates how INDEX and POSITION return different results in this condition.

```
SELECT column_1, column_2, INDEX(column_1,column_2)
FROM iptest;
```

The result of this query looks like the following:

```
   column_1     column_2       Index(column_1,column_2)
   -----------  -----------    ------------------------
   MN<AC
>       <C
>                                  6
   MN<AC
>P      <A
>                                  4
   MN<AB
>P      P                                    9
   MN<AB
>P      <B
>                                  6
```

With the same session characteristics in place, perform the semantically identical query on the table using POSITION instead of INDEX.

```
   SELECT column_1, column_2, POSITION(column_2 IN column_1)
   FROM iptest;
```

The result of this query looks like the following:

```
   column_1     column_2     Position(column_2 in column_1)
   -----------  -----------  ------------------------------
   MN<AC
>       <C
>                                     4
   MN<AC
>P      <A
>                                     3
   MN<AB
>P      P                                       5
   MN<AB
>P      <B
>                                     4
```

The different results are accounted for by the following differences in how INDEX and POSITION operate in this particular case.

- INDEX counts Shift-Out and Shift-In characters; POSITION does not.
- INDEX counts bytes; POSITION counts logical characters. As a result, an A , for example, counts as two bytes (two physical characters) for INDEX, but only one logical character for POSITION.

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

REVERSE

Reverses the input string.

CHAR, VARCHAR, and CLOB are supported while other types must be explicitly cast. The input can be NULL or an empty string.

The result character set is the same as the *source_string* character set for Unicode and Latin. The result character set is Unicode for all other supported character sets.

REVERSE Function Syntax

```
[TD_SYSFNLIB.] REVERSE ( source_string )
```

Syntax Elements

TD_SYSFNLIB

Name of the database where the function is located.

source_string

A character string or string expression.

Usage Notes

When submitting queries using function REVERSE on a client that uses ODBC, disable ODBC parsing. Otherwise, the query may be altered by ODBC to a SUBSTR function call.

Character Sets Supported

The following character sets are supported with the REVERSE function:

- Unicode
- Latin
- Kanji SJIS
- Graphic

Result Type

The reverse of the input is returned. The return type is set to the input type with the exception of CHAR. A CHAR input has a result type of VARCHAR.

Example

```
SELECT REVERSE('Test String');
Result- 'gnirtS tseT'
```

RIGHT

Starting from the end of the input string, a substring is created with the number of characters specified by the second parameter.

The RIGHT function can be called with the 'RIGHT' or 'TD_RIGHT' alias names.

The arguments include two parameters:

- First Parameter

  The input string that the substring will be created from. CHAR, VARCHAR, and CLOB are supported while other types must be explicitly casted. The input can be null or an empty string.
- Second Parameter

  A positive integer specifying the number of characters desired from the right side of the string. If the number of character exceeds the number of characters in the original string, then the original string will be returned.

RIGHT Function Syntax

```
[TD_SYSFNLIB.] RIGHT ( source_string, length )
```

Syntax Elements

TD_SYSFNLIB

Name of the database where the function is located.

source_string

A character string or string expression.

If the length of *source_string* is greater than *length*, *source_string* is truncated to *length* characters.

length

An integer specifying the length of the returned string.

Usage Notes

When submitting queries using function RIGHT on a client that uses ODBC, disable ODBC parsing. Otherwise, the query may be altered by ODBC to a SUBSTR function call.

Character Sets Supported

The following character sets are supported with the RIGHT function:

- Unicode
- Latin
- Kanji SJIS
- Graphic

Result Type

A substring is returned. The return type is set to the input type with the exception of CHAR. A CHAR input has a result type of VARCHAR.

The result character set is the same as the *source_string* character set for Unicode and Latin. The result character set is Unicode for all other supported character sets.

Example

```
SELECT RIGHT('Test String',6);
Result- 'String'
```

RPAD

Returns the *source_string* padded to the right with the characters in *fill_string* so that the resulting string is *length* characters.

RPAD is an embedded services system function. For information on activating and invoking embedded services functions, see Embedded Services System Functions.

RPAD Function Syntax

```
[TD_SYSFNLIB.] RPAD ( source_string, length [, fill_string ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

source_string

A character string or string expression.

If *source_string* is NULL, NULL is returned.

If the length of *source_string* is greater than *length*, *source_string* is truncated to *length* characters.

length

The number of characters in the resulting string.

fill_string

The string of characters used to pad the *source_string*.

The sequence of characters in *fill_string* is replicated as necessary.

If *fill_string* is not specified, *source_string* will be padded to the right with space characters.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *source_string* = VARCHAR or CLOB
- *length* = INTEGER, BIGINT, or NUMBER
- *fill_string* = CHAR, VARCHAR, or CLOB

If any of the input arguments are NULL, the function returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type and character set are the same as those of the *source_string* argument. For example, if the *source_string* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is ARCHAR CHARACTER SET UNICODE.

Examples

Example

The following query returns the result 'buildingyzy'.

```
SELECT RPAD('building', 11, 'yz');
```

The following query returns the result 'build'.

```
SELECT RPAD('building', 5, 'yz');
```

The following query returns the result 'building   '. The space character is used by default to pad the source string.

```
SELECT RPAD('building', 11);
```

Example

The following query returns the result 'build'.

```
SELECT RPAD('building', 5, 'yz');
```

Example

The following query returns the result 'building   '. The space character is used by default to pad the source string.

```
SELECT RPAD('building', 11);
```

Related Information

For details, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

RTRIM

Returns the argument *expr1*, with its right-most characters removed up to the first character that is not in the argument *expr2*.

RTRIM is an embedded services system function.

RTRIM Function Syntax

```
[TD_SYSFNLIB.] RTRIM ( expr1 [, expr2 ] )
```

Syntax Elements

TD_SYSFNLIB.
Name of the database where the function is located.

expr1
A character string or string expression. If *expr1* is NULL, NULL is returned.

expr2
A string of characters or a numeric expression that will be removed from *string1*. If *expr2* is specified, it must be the same datatype as *expr1*. If *expr2* is not specified, the default is to use a single space character.

Argument Types and Rules

Expressions passed to this function can be one of the following data types:

- CHAR
- VARCHAR
- CLOB
- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- FLOAT/REAL/DOUBLE PRECISION

- DECIMAL/NUMERIC
- NUMBER

If any of the input arguments are NULL, the function returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type and character set are the same as those of the *expr1* argument. For example, if the *expr2* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Examples

Example

The following query returns the result 'TURNERyxX'. RTRIM removes the individual occurrences of 'x' and 'y', and it stops removing characters when it encounters 'X' because 'X' is not in the *expr2* argument.

```
SELECT RTRIM('TURNERyxXxy,'xy');
```

```
SELECT RTRIM('  RIGHTT TRIM   ');
```

Example

The following query returns the result ' RIGHTT TRIM' because the leading spaces are not removed. It stops removing characters when it encounters 'M' because 'M' is not in the *expr2* argument. Since the *expr2* argument is not explicitly specified, the default of a single space is used.

```
    SELECT RTRIM('  RIGHTT TRIM   ');
```

Related Information

- “Compatible Types” in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

SOUNDEX

Returns a character string that represents the Soundex code for *string_expression*.

Soundex

Soundex is a system that codes surnames having the same or similar sounds, but variant spellings. The Soundex system was first used by the National Archives in 1880 to index the United States census.

Soundex codes begin with the first letter of the surname followed by a three-digit code. Zeros are added to names that do not have enough letters.

SOUNDEX Function Syntax

```
SOUNDEX ( string_expression )
```

Syntax Elements

string_expression

A character string or expression that contains a surname to be evaluated in simple Latin characters.

A simple Latin character is one that does not have diacritical marks such as tilde (~) or acute accent (´).

There are 26 uppercase simple Latin characters and 26 lowercase simple Latin characters.

SOUNDEX is case insensitive.

Embedded or trailing pad characters within *character_string* return an error to the requestor.

Argument Types

Use SOUNDEX on character strings or character string expressions that use the LATIN or UNICODE server character set.

SOUNDEX does not accept CLOB types.

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

Implicit type conversion of UDTs for system operators and functions, including SOUNDEX, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

If the *string_expression* argument is *null*, the result is *null*.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

SOUNDEX Usage Notes

Soundex Coding Guide

The following process outlines the Soundex coding guide:

1. Retain the first letter of the name.
2. Drop all occurrences of the following letters:

   A, E, I, O, U, Y, H, W

   in other positions.
3. Assign the following number to the remaining letters after the first letter:

   1 = B, F, P, V

   2 = C, G, J, K, Q, S X, Z

   3 = D, T

   4 = L

   5 = M, N

   6 = R
4. If two or more letters with the same code are adjacent in the original name or adjacent except for any intervening H or W, omit all but the first.
5. Convert the form “letter, digit, digit, digit,” by adding trailing zeros if less than three digits.
6. Drop the rightmost digits if more than three digits.
7. Names with adjacent letters having the same equivalent number are coded as one letter with a single number

   Surname prefixes are generally not used.

Examples

Example

The following SELECT statement returns the result that follows.

```
SELECT SOUNDEX ('ashcraft');

Soundex('ashcraft')
-------------------
a261
```

The surname “ashcraft” initially evaluates to “a2h2613,” but the following Soundex rules convert the result to a261.

- “h” is dropped because it occurs in the third position. Soundex drops all occurrences of the following characters in any position other than the first.

  A, E, I, O, U, Y, H, W
- “2” is dropped because it represents the second occurrence of one of the following characters:

  C, G, J, K, Q, S X, Z

  If two or more characters with the same code are adjacent in the original name, or adjacent except for any intervening H or W, Soundex omits all but the code for the first occurrence of the character in the returned code.
- “3” is dropped because Soundex drops the rightmost digits if *character_string* evaluates to more than three digits following the initial simple Latin character.

Example

This example and Example use the following table data:

```
SELECT family_name FROM family;

family_name
-----------
John
Joan
Joey
joanne
michael
Bob
```

Here are the results of the SOUNDEX function on the data in the family_name column:

```
SELECT SOUNDEX(TRIM(family.family_name));

Soundex(TRIM(BOTH FROM family_name))
------------------------------------
J500
J500
B100
J000
m240
j500
```

Example

Find all family names in Family that sound like “Joan”.

```
SELECT family_name
FROM family
WHERE SOUNDEX(TRIM(family.family_name)) = SOUNDEX('Joan');

family_name
-----------
John
Joan
Joanne
```

Examples of Non Valid Usage

The following SOUNDEX examples are not valid for the reasons given in the table.

Statement	Why the Statement is Not Valid
SELECT SOUNDEX(12345);	12345 is a numeric string, not a character string.
SELECT SOUNDEX('ábç');	The characters á and ç are not simple Latin characters.

Related Information

- For information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- For details on how to disable the SOUNDEX extension, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For information on implicit type conversion of UDTs, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

STRING_CS

Returns a heuristically derived integer value that you can use to help determine which KANJI1-compatible client character set was used to encode *string_expression*.

Note:

The result is not guaranteed correct, but should work for most strings likely to be encountered.

STRING_CS Function Syntax

```
STRING_CS ( string_expression )
```

Syntax Elements

string_expression

A CHAR or VARCHAR character string or expression.

Argument Types

Use STRING_CS on character strings or character string expressions that use the KANJI1 server character set. (Non-KANJI1 character strings will be coerced to KANJI1, but the results are unlikely to be useful.)

STRING_CS does not accept CLOB or UDT types.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Value

STRING_CS returns a heuristically derived INTEGER value that you can use to help determine the client character set that was used to encode the KANJI1 character string or expression. The result value can also help determine which client character set to use to interpret the character data.

IF the result value is …	THEN the heuristic found that *string_expression* …
-1	most likely uses a single-byte client character set encoding, but it may also contain a mix of encodings.
0	does not contain anything distinguishable from any particular character set, so any character set that you use to interpret *string_expression* provides the same result. Not all translations use the same interpretation for the characters represented by 0x5C and 0x7E, however. If *string_expression* contains: 0x5C and you want it to be interpreted as REVERSE SOLIDUS, use a single-byte character set. 0x7E and you want it to be interpreted as TILDE, use a single-byte character set. 0x5C and you want it to be interpreted as YEN SIGN, or 0x7E and you want it to be interpreted as OVERLINE, use any of the following: • KANJISJIS_0S • KANJIEBCDIC5026_0I • KANJIEBCDIC5035_0I • KATAKANAEBCDIC • KANJIEUC_0U
1	uses the encoding of one of the following: • KANJIEBCDIC5026_0I • KANJIEBCDIC5035_0I • KATAKANAEBCDIC
2	uses the encoding of KANJIEUC_0U.
3	uses the encoding of KANJISJIS_0S.

Usage Notes

STRING_CS helps determine which encoding to use when using the TRANSLATE function to translate a string from the KANJI1 server character set to the UNICODE server character set.

IF the result value is …	THEN substitute the following value for *source_TO_target* in TRANSLATE(*string_expression* USING *source_to_target*) …
-1	KANJI1_SBC_TO_UNICODE.
0	KANJI1_SBC_TO_UNICODE.
1	KANJI1_KANJIEBCDIC_TO_UNICODE.
2	KANJI1_KANJIEUC_TO_UNICODE.
3	KANJI1_KANJISJIS_TO_UNICODE.

Examples

Example: Using STRING_CS to Determine the Client Character Set

Consider the following table definition:

```
CREATE TABLE SysNames
   (SysID INTEGER
   ,SysName VARCHAR(30) CHARACTER SET KANJI1);
```

Suppose the session character set is KANJIEBCDIC5026_0I. The following statement inserts the mixed single-byte/multibyte character string '<TEST >Q' into the SysName column of the SysNames table:

```
INSERT SysNames (101, '0E42E342C542E242E30FD8'XC);
```

Using STRING_CS to determine the client character set that was used to encode the string produces the results that follow:

```
SELECT STRING_CS(SysName) FROM SysNames WHERE SysID = 101;
String_CS(SysName)
------------------
                 1
```

Example: Using STRING_CS to Translate a KANJI1 String to UNICODE

Consider the SysNames table from the preceding example, “Example: Using STRING_CS to Determine the Client Character Set.”

The following statement uses STRING_CS to determine which encoding to use to translate strings in the SysName column from the KANJI1 server character set to the UNICODE server character set:

```
SELECT CASE STRING_CS(SysName)
   WHEN 0 THEN TRANSLATE(SysName USING KANJI1_SBC_TO_UNICODE)
   WHEN 1 THEN TRANSLATE(SysName USING KANJI1_KANJIEBCDIC_TO_UNICODE)
   WHEN 2 THEN TRANSLATE(SysName USING KANJI1_KANJIEUC_TO_UNICODE)
   WHEN 3 THEN TRANSLATE(SysName USING KANJI1_KANJISJIS_TO_UNICODE)
   ELSE TRANSLATE(SysName USING KANJI1_SBC_TO_UNICODE)
   END
FROM SysNames;
```

Related Information

- "Compatible Types" in *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To activate and invoke embedded services functions, see Embedded Services System Functions.

STRTOK

Splits *instring* into tokens based on the specified list of delimiter characters and returns the *n*th token, where *n* is specified by the *tokennum* argument.

STRTOK is an embedded services system function.

STRTOK Function Syntax

```
[TD_SYSFNLIB.] STRTOK ( instring [, delimiter ] [, tokennum ] )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

instring

A character string or string expression.

If *instring* is NULL, NULL is returned.

delimiter

A list of delimiter characters. Each character in the string is considered a delimiter character.

If not specified, the value defaults to a space character.

tokennum

The ordinal token to return.

If not specified, the default value is 1.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *instring* = VARCHAR(32000) or CLOB
- *delimiter* = VARCHAR(64)
- *tokennum* = INTEGER

- *token size* = The maximum limit of any token returned is 256.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result data type is VARCHAR and the character set is the same as that of the *instring* argument. For example, if the *instring* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Usage Notes

STRTOK has similar semantics as the standard C/C++ library STRTOK function.

If *instring* contains fewer tokens than *tokennum*, the function returns NULL.

If either *instring* or *delimiter* is NULL, the function returns NULL.

Example: Using STRTOK

Consider the following table and inserted data.

```
   CREATE TABLE t (id INTEGER, str VARCHAR(256));
   INSERT INTO t VALUES (1,'Teradata-Warehouse 13.10 - Combine 2 powerful forms of
business intelligence (BI).');
   INSERT INTO t VALUES (2,'http://www.teradata.com/');
```

The following query returns the third token from the t.str string value. There are only two tokens in the second string, so the function returns NULL for that row.

```
SELECT id, STRTOK(t.str, ' -/', 3) FROM t;
```

The output from the query is:

```
         id  STRTOK(str,' -/',3)
-----------  ------------------------------------------------------------
          1  13.10
          2  NULL
```

STRTOK_SPLIT_TO_TABLE

Splits strings into a table of tokens based on the provided delimiter(s) string.

Note:

STRTOK_SPLIT_TO_TABLE has similar semantics to the standard C/C++ library STRTOK function.

STRTOK_SPLIT_TO_TABLE is an embedded services system function.

STRTOK_SPLIT_TO_TABLE Function Syntax

```
[TD_SYSFNLIB.] STRTOK_SPLIT_TO_TABLE ( inkey, instring, delimiters )
  RETURNS ( outkey, tokennum, token )
```

Syntax Elements

TD_SYSFNLIB.

Name of the database where the function is located.

inkey

A numeric or character expresssion.

instring

If *instring* is NULL, NULL is returned.

delimiters

A character string or string expression.

If *delimiters* is NULL, NULL is returned.

If not specified, the value defaults to a space character.

outkey

A numeric or character expresssion.

tokennum

A character argument.

The *token* from the input string in the same character set as *instring*.

token

A character expression.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *inkey* = NUMERIC or VARCHAR
- *instring* = VARCHAR(32000) or CLOB
- *delimiter* = VARCHAR(64)

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type

The result row type is:

- *outkey* = NUMERIC OR VARCHAR(64)
- *tokennum* = INTEGER
- *token* = VARCHAR(256)

Example

If:

```
CREATE TABLE t (id integer, str varchar(256)character set unicode);
```

and:

```
INSERT INTO t VALUES (1,'Teradata-Warehouse 13.10 - Combine 2 powerful forms of
business intelligence (BI).');
```

and:

```
insert into t values (2,'http://www.teradata.com/');
```

then:

```
SELECT d.* FROM TABLE (strtok_split_to_table(t.id, t.str, ' -/')
RETURNS (outkey integer, tokennum integer, token varchar(20)character set
unicode) ) as d order by 1,2;
outkey   tokennum  token
-----------  -----------  --------------------------
          1            1  Teradata
          1            2  Warehouse
          1            3  13.10
          1            4  Combine
          1            5  2
          1            6  powerful
          1            7  forms
          1            8  of
          1            9  business
          1           10  intelligence
          1           11  (BI).
          2            1  http:
          2            2  www.teradata.com
```

SUBSTRING

Extracts a substring from a named string based on position.

SUBSTRING Function Syntax

ANSI

```
SUBSTRING ( string_expression FROM n1 [ FOR n2 ] )
```

Syntax Elements

string_expression

A string expression from which the substring is to be extracted.

n1

The starting position of the substring to extract from *string_expression*.

The length of the substring to extract from *string_expression*.

If *n1* < 0, the function returns an error.

n2

The length of the substring to extract from *string_expression*.

If *n2* < 0, the function returns an error.

If you omit *n2*, then you extract the entire right hand portion of the named string or string expression, beginning at the position named by *n1*.

If *string_expression* is a BYTE or CHAR type and you omit *n2*, trailing binary zeros or pad characters are trimmed.

Teradata

```
{ SUBSTRING | SUBSTR } ( string_expression, n1 [, n2 ] )
```

Syntax Elements

string_expression

A string expression from which the substring is to be extracted.

n1

The starting position of the substring to extract from *string_expression*.

n2

The length of the substring to extract from *string_expression*.

If *n2* < 0, the function returns an error.

If *string_expression* is a BYTE or CHAR type and you omit *n2*, trailing binary zeros or pad characters are trimmed.

Argument Types and Rules

SUBSTRING and SUBSTR operate on the following types of arguments:

- Character
- Byte
- Numeric

If the string_expression argument is numeric, it is implicitly converted to CHARACTER type.

- UDTs that have implicit casts to any of the following predefined types:
    - Character
    - Numeric
    - Byte
    - DATE

    To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

    Implicit type conversion of UDTs for system operators and functions, including SUBSTRING and SUBSTR, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

Result Type and Attributes

Here are the default result type and attributes for SUBSTR(string , n1 , n2) and SUBSTRING(string FROM n1 FOR n2):

If the string argument is a:

- BLOB, the result type is BLOB(n).
- Byte string other than BLOB, the result type is VARBYTE(n).
- CLOB, the result type is CLOB(n).
- Numeric or character string other than CLOB, the result type is VARCHAR(n).

In ANSI mode, the value of n for the resulting BLOB(n), VARBYTE(n), CLOB(n), or VARCHAR(n) is the same as the original string. In Teradata mode, the value of n for the result type depends on the number of characters or bytes in the resulting string. To get the data type of the resulting string, use the TYPE function.

SUBSTRING Usage Notes

Result Value

SUBSTRING/SUBSTR extracts n2 characters or bytes from string_expression starting at position n1.

To get the number of characters or bytes in the resulting string, use the BYTE function for byte strings and the CHARACTER_LENGTH function for character strings.

If either of the following conditions are true, SUBSTRING/SUBSTR returns a zero length string:

- (n1 > string_length) AND (0 ≤ n2)

- (n1 < 1) AND (0 ≤ n2) AND ((n2 + n1 - 1) ≤ 0)

IF n2 is …	THEN …			
specified	IF …	THEN …		
	n2 < 0	SUBSTRING/SUBSTR returns an error.		
	0 ≤ n2 and n1 > string_length	SUBSTRING/SUBSTR returns a string containing zero characters.		
	0 ≤ n2 and n1 < 1	SUBSTRING/SUBSTR sets n4 = n2 + n1 - 1 and sets n3 = 1.		
			IF …	THEN SUBSTRING/ SUBSTR returns …
			n4 ≤ 0	a string containing zero characters.
			n4 > string_length	the source string.
			0 < n4 <= string_length	a string that starts at n3 and extends for n4 characters.
	0 < n2 AND 1 ≤ n1 ≤ string_length		IF …	THEN SUBSTRING/ SUBSTR returns a string …
			(n1 + n2 - 1) > string_length	that starts at n1 and ends with the last character of the source string.
			0 < (n1 + n2 - 1) ≤ string_length	that starts at n1 and extends for n2 characters.
not specified	IF …	THEN SUBSTRING/SUBSTR returns …		
	n1 < 1	the source string. If the source string is a CHAR type, trailing pad characters are trimmed.		
	n1 > string_length	a string containing zero characters.		
	1 ≤ n1 ≤ string_length	a string that starts at n1 and ends with the last character of the source string. If the source string is a CHAR type, trailing pad characters are trimmed.		

Usage Rules for SUBSTRING and SUBSTR

SUBSTRING is the ANSI SQL:2011 syntax. Teradata syntax using SUBSTR is supported for backward compatibility. Use SUBSTRING in place of SUBSTR for ANSI compliance.

Use SUBSTRING in place of MSUBSTR. (MSUBSTR no longer appears in this book because its use is deprecated and it will not be supported after support for KANJI1 is dropped.)

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see *KANJI1 Character Set* in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Difference Between SUBSTRING and SUBSTR

SUBSTRING and SUBSTR perform identically except when they operate on character strings in Teradata mode where the server character set is KANJI1 and the client character set is KanjiEBCDIC.

In this case, SUBSTR interprets n1 and n2 as physical units, making the DB2-compliant SUBSTR operate on a byte-by-byte basis. Shift-Out and Shift-In bytes are significant because the result might be formatted incorrectly. For example, the result string might not contain either the opening Shift-Out character or the closing Shift-In character.

Otherwise, if string_expression is character data, then SUBSTRING expects mixed single byte and multibyte character strings and operates on logical characters that are valid for the character set of the session. In this case, n1 is a positive integer pointing to the first character of the result and n2 is in terms of logical characters.

Examples

Example: Searching for Car Serial IDs

Suppose sn is a CHARACTER(15) field of Serial IDs for Automobiles and positions 3 to 5 represent the country of origin as three letters.

For example:

```
12JAP3764-35421
37USA9873-26189
11KOR1221-13145
```

To search for serial IDs of cars made in the USA:

```
SELECT make, sn
FROM autos
WHERE SUBSTRING (sn FROM 3 FOR 3) = 'USA';
```

Example: Accessing Serial ID Characters

If we want the last five characters of the serial ID, which represent manufacturing sequence number, another substring can be accessed.

```
SELECT make, SUBSTRING (sn FROM 11) AS sequence
FROM autos
WHERE SUBSTRING (sn FROM 3 FOR 3) = 'USA';
```

Example: Limiting Returned Characters

Suppose nameaddress is a VARCHAR(120) field, and the application used positions 1 to 30 for name, starting address at position 31. To return address only, but limit the number of characters returned to 50 use:

```
...
SUBSTRING (nameaddress FROM 31 FOR 50)
```

This returns an address of up to 50 characters.

Example: Using a SELECT Statement to Request Substrings

The following example shows a SELECT statement requesting substrings from a character field in positions 1 through 4 for every row:

```
SELECT SUBSTRING (jobtitle FROM 1 FOR 4)
FROM employee ;
```

The result is as follows.

```
Substring(jobtitle From 1 For 4)
--------------------------------
Tech
Cont
Sale
Secr
```

```
Test
 ...
```

Example: Using the CREATE TABLE cstr Table

Consider the following table:

```
CREATE TABLE cstr
  (c1 CHAR(3) CHARACTER SET LATIN
  ,c2 CHAR(10) CHARACTER SET KANJI1);
```

INSERT cstr ('abc', '92年abc

INSERT cstr ('abc', '92abcFunction	Result
SELECT SUBSTR(c2, 2, 3) FROM cstr;	'2年 a'
SELECT SUBSTR(c1, 2, 2) FROM cstr;	'bc'

Example: Differences Between SUBSTR and SUBSTRING

Consider the following table:

```
CREATE TABLE ctable1
  (c1 VARCHAR(11) CHARACTER SET KANJI1);
```

The following table shows the difference between SUBSTR and SUBSTRING in Teradata mode for KANJI1 strings from KanjiEBCDIC client character set.

IF c1 contains …	THEN this query …	Returns …
MN<ABC >P	SELECT SUBSTR(c1,2) FROM ctable1;	N<ABC>P
	SELECT SUBSTR(c1,3,8) FROM ctable1;	<ABC>
	SELECT SUBSTR(c1,4) FROM ctable1;	ABC >P **Note:** The client application might not be able to properly interpret the resulting multibyte characters because the shift out (<) is missing.
	SELECT SUBSTRING(c1 FROM 2) FROM ctable1;	N<ABC>P

IF c1 contains …	THEN this query …	Returns …
	SELECT SUBSTRING(c1 FROM 3 FOR 8) FROM ctable1;	<ABC>P
	SELECT SUBSTRING(c1 FROM 4) FROM ctable1;	<BC>P

Example: Using the KanjiEUC Client Character Set with the ctable1 Table

The following table shows examples for the KanjiEUC client character set, where ctable1 is the table defined in "Example: Effect of the Order of SELECT Statements on Data Type" in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

IF c1 contains …	THEN this query …	Returns …
A ss2 B CD	SELECT SUBSTR(c1,2) FROM ctable1;	ss2 B CD
ss3 A ss2 B ss3 C ss2 D	SELECT SUBSTR(c1,2,2) FROM ctable1;	ss2 B ss3 C

Example: Using Examples for the KanjiShift-JIS Client Character Set

The following table shows examples for KanjiShift-JIS client character set, where ctable1 is the table defined in "Example: Effect of the Order of SELECT Statements on Data Type" in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

IF c1 contains …	THEN this query …	Returns …
mnABC X	SELECT SUBSTR(c1, 6, 1) FROM ctable1;	X
	SELECT SUBSTR(c1,4) FROM ctable1;	BC X

Example: Applying the SUBSTRING Function to a CLOB Column

The following statement applies the SUBSTRING function to a CLOB column in table full_text and stores the result in a CLOB column in table sub_text.

```
INSERT sub_text (text)
SELECT SUBSTRING (text FROM 9 FOR 128000)
FROM full_text;
```

Related Information

- For CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

TRANSLATE

Converts a character string or character string expression from one server character set to another server character set.

TRANSLATE Function Syntax

```
TRANSLATE (
  character_string_expression
  USING source_repertoire_name
  [ _encoding ]
  TO target_repertoire_name
  [ _suffix ]
  [ WITH ERROR ]
)
```

Syntax Elements

character_string_expression

A character string or character string expression for which the hexadecimal representation is to be returned.

If *character_string_expression* is not a character type, an error is returned.

source_repertoire_name

The source character set of the string to translate.

A value of LOCALE can be specified for *source_repertoire_name* to translate a character string from LATIN or KANJI1 to UNICODE using a source repertoire determined by the language support mode of the system and the client character set of the session.

_encoding

A literal for translating from KANJI1 to UNICODE that indicates a specific encoding of KANJI1.

The *_encoding* option is not allowed if LOCALE is specified for *source_repertoire_name* or *target_repertoire_name*.

If the translation is from these character sets:

- KatakanaEBCDIC
- KanjiEBCDIC5026_0I
- KanjiEBCDIC5038_0I

Use the following value for _encoding: _KanjiEBCDIC

KanjiEUC_0U, use the following value for _encoding: _KanjiEUC

KanjiShiftJIS_0S, use the following value for _encoding: _KANJISJIS

ASCII or EBCDIC, use the following value for _encoding: _SBC

target_repertoire_name

The target character set of the string to translate.

A value of LOCALE can be specified for *target_repertoire_name* to translate a character string from UNICODE to LATIN or KANJI1 using a target repertoire determined by the language support mode of the system and the client character set of the session.

_suffix

Specifies that the translation maps some source characters to semantically different characters.

For example, a translation that specifies the _Halfwidth suffix maps any character with a halfwidth variant to that variant, and all fullwidth variants to their non-fullwidth counterparts.

The *_suffix* option also indicates the form of character data translated from UNICODE to the KANJI1 server character set, for example, _KanjiEUC.

Valid values are:

- _KanjiEBCDIC
- _KanjiEUC
- _KANJISJIS
- _SBC
- _PadSpace
- _PadGraphic
- _Fullwidth
- _Halfwidth
- _FoldSpace
- _VarGraphic

The *_suffix* option is not allowed if LOCALE is specified for *source_repertoire_name* or *target_repertoire_name*.

WITH ERROR

Specifies that the translation replaces offending characters in the string with a designated error character, instead of reporting an error.

Argument Types

Use TRANSLATE on character strings or character string expressions.

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including TRANSLATE, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

Result Type and Attributes

The default attributes for TRANSLATE (*string* USING *source* _TO_*target*) are as follows:

- If the argument is CHAR or VARCHAR, the result is VARCHAR(*n*) or CHARACTER SET *target*
- If the argument is CLOB, the result is CLOB(*n*) or CHARACTER SET *target*

where source_TO_target determines the character set value of target, according to the supported translations in “Supported Translations Between Character Sets”.

If the *string* USING *source* _TO_*target* argument is *null*, the result is *null*.

TRANSLATE Usage Notes

Supported Translations for CLOB Strings

The following translations are supported for CLOB strings:

- LATIN_TO_UNICODE
- UNICODE_TO_LATIN

Supported Translations Between Character Sets

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see *KANJI1 Character Set* in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

The following table lists the supported values that you can use for *source_repertoire_name* _TO_*target_repertoire_name* to translate between server character sets.

Value of *source* _TO_*target*	Source Character Set	Target Character Set
GRAPHIC_TO_KANJISJIS	GRAPHIC	KANJISJIS
GRAPHIC_TO_LATIN	GRAPHIC	LATIN
GRAPHIC_TO_UNICODE	GRAPHIC	UNICODE
GRAPHIC_TO_UNICODE_PadSpace	GRAPHIC	UNICODE
KANJI1_KanjiEBCDIC_TO_UNICODE	KANJI1	UNICODE
KANJI1_KanjiEUC_TO_UNICODE	KANJI1	UNICODE
KANJI1_KANJISJIS_TO_UNICODE	KANJI1	UNICODE
KANJI1_SBC_TO_UNICODE	KANJI1	UNICODE
KANJISJIS_TO_GRAPHIC	KANJISJIS	GRAPHIC
KANJISJIS_TO_LATIN	KANJISJIS	LATIN
KANJISJIS_TO_UNICODE	KANJISJIS	UNICODE
LATIN_TO_GRAPHIC	LATIN	GRAPHIC
LATIN_TO_KANJISJIS	LATIN	KANJISJIS
LATIN_TO_UNICODE	LATIN	UNICODE
LOCALE_TO_UNICODE	KANJI1	UNICODE
	LATIN	
UNICODE_TO_GRAPHIC	UNICODE	GRAPHIC
UNICODE_TO_GRAPHIC_PadGraphic	UNICODE	GRAPHIC
UNICODE_TO_GRAPHIC_VarGraphic	UNICODE	GRAPHIC
UNICODE_TO_KANJI1_KanjiEBCDIC	UNICODE	KANJI1
UNICODE_TO_KANJI1_KanjiEUC	UNICODE	KANJI1
UNICODE_TO_KANJI1_KANJISJIS	UNICODE	KANJI1
UNICODE_TO_KANJI1_SBC	UNICODE	KANJI1
UNICODE_TO_KANJISJIS	UNICODE	KANJISJIS
UNICODE_TO_LATIN	UNICODE	LATIN

Value of *source* _TO_*target*	Source Character Set	Target Character Set
UNICODE_TO_LOCALE	UNICODE	KANJI1
		LATIN
UNICODE_TO_UNICODE_FoldSpace	UNICODE	UNICODE
UNICODE_TO_UNICODE_Fullwidth	UNICODE	UNICODE
UNICODE_TO_UNICODE_Halfwidth	UNICODE	UNICODE
UNICODE_TO_UNICODE_NFC	UNICODE	UNICODE
UNICODE_TO_UNICODE_NFD	UNICODE	UNICODE
UNICODE_TO_UNICODE_NFKC	UNICODE	UNICODE
UNICODE_TO_UNICODE_NFKD	UNICODE	UNICODE

If the value specified for *source_repertoire_name* _TO_*target_repertoire_name* is UNICODE_TO_LOCALE or LOCALE_TO_UNICODE, the repertoire that the translation uses for LOCALE is determined by the language support mode for the system and the client character set for the session.

IF the language support mode is …	AND the session character set is …		THEN the repertoire that the translation uses for LOCALE is …
standard	any		LATIN
Japanese	• ASCII • LATIN1252_0A • LATIN1_0A • LATIN9_0A	• EBCDIC • EBCDIC037_0E • EBCDIC273_0E • EBCDIC277_0E	KANJI1_SBC
	• any other client character set with a name that has a suffix of _0A or _0E • a single-byte, extended site-defined client character set		
	• KANJIEBCDIC5026_0I • KANJIEBCDIC5035_0I • KATAKANAEBCDIC • any other client character set with a name that has a suffix of _0I		KANJI1_ KANJIEBCDIC
	• UTF8 • UTF16 • KanjiShiftJIS_0S • any other client character set with a name that has a suffix of _0S • a multibyte extended site-defined client character set		KANJI1_ KANJISJIS

IF the language support mode is …	AND the session character set is …	THEN the repertoire that the translation uses for LOCALE is …
	• KanjiEUC_0U • any other client character set with a name that has a suffix of _0U	KANJI1_ KanjiEUC

Source Characters That Generate Errors

The following table lists the characters that generate errors for specific *source_repertoire_name* _TO_*target_repertoire_name* translations. For supported translations that do not appear in the table, only the error character generates errors.

Value of *source* _TO_*target*	Source Characters That Generate Errors
• LATIN_TO_GRAPHIC • KANJISJIS_TO_GRAPHIC • UNICODE_TO_GRAPHIC	non-GRAPHIC
• LATIN_TO_KANJISJIS • KANJI1_KANJISJIS_TO_UNICODE • GRAPHIC_TO_KANJISJIS • UNICODE_TO_KANJI1_KANJISJIS • UNICODE_TO_KANJISJIS • LOCALE_TO_UNICODE or UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is KANJI1_KANJISJIS	non-KANJISJIS
• KANJI1_KanjiEBCDIC_TO_UNICODE • UNICODE_TO_KANJI1_KanjiEBCDIC • LOCALE_TO_UNICODE or UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is KANJI1_KanjiEBCDIC	non-KanjiEBCDIC KANJI1 is very permissive, so there may be characters outside the defined region of the encoding as well as illegal form-of-use errors.
• KANJI1_KanjiEUC_TO_UNICODE • UNICODE_TO_KANJI1_KanjiEUC • LOCALE_TO_UNICODE or UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is KANJI1_KanjiEUC	non-KanjiEUC
• KANJISJIS_TO_LATIN • GRAPHIC_TO_LATIN • UNICODE_TO_LATIN • UNICODE_TO_KANJI1_SBC	non-LATIN

Value of *source* _TO_*target*	Source Characters That Generate Errors
• UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is LATIN or KANJI1_SBC	

Error Characters Assigned by the WITH ERROR Option

The error characters substituted for offending characters that cannot be translated to a designated target character set are defined in the following table.

Target Character Set	Error Character
LATIN	0x1A
KANJI1	0x1A
KANJISJIS	0x1A
UNICODE	U+FFFD
GRAPHIC	U+FFFD

Suffixes

The _*suffix* variable is used for translations that map source characters to semantically different characters. They indicate the nature of the semantic transformation.

The translations perform minor, yet essential, semantic changes to the data, such as halfwidth/fullwidth conversions, and Space folding modification.

The _*suffix* variable also indicates the form of character data translated from UNICODE to the KANJI1 server character set in one of the four possible encodings, for example Unicode_TO_Kanji1_KanjiEBCDIC.

This form of translation is also useful for migrating object names. For information, see Migration.

Translations Between Fullwidth and Halfwidth Character Data

UNICODE has an area known as the compatibility zone. Among other things, this zone includes halfwidth and fullwidth variants of characters that exist elsewhere in the standard.

Translations between fullwidth and halfwidth are provided by the following *source_repertoire_name* _TO_*target_repertoire_name* values.

source _TO_*target*	Meaning
UNICODE_TO_ UNICODE_Halfwidth	Maps the fullwidth characters of Unicode to the halfwidth characters of Unicode. Other characters remain unchanged by the translation. See *UNICODE to UNICODE_Halfwidth*, B035-1201.
UNICODE_TO_ UNICODE_Fullwidth	Maps the halfwidth characters of Unicode to the fullwidth characters of Unicode. At the same time, it maps any character defined by the standard as a halfwidth variant to its non-halfwidth counterpart outside the compatibility zone. Other characters remain unchanged by the translation. See *UNICODE to UNICODE_Halfwidth*, B035-1201.
UNICODE_TO_ GRAPHIC_ VarGraphic	This translation is an ANSI equivalent to the VARGRAPHIC function. See *UNICODE to Vargraphic*, B0035-1057.

Note:

The mapping and translation files are readable, but are intended to be used by software. In most cases, items not in a mapping file are mapped to themselves.

Also note that these translations are useful for maintaining more information as a step in translating GRAPHIC to LATIN and vice versa.

Space Folding

Space folding is performed via UNICODE_TO_UNICODE_FoldSpace. All characters defined as space are converted to U+0020.

All other characters are left unchanged.

UNICODE Normalization Form Translations

Teradata supports translation using the 4 UNICODE normalization forms: NFC, NFD, NFKC, and NFKD, which correspond to the ANSI NORMALIZE function. You can perform these translations using:

- UNICODE_TO_UNICODE_NFC
- UNICODE_TO_UNICODE_NFD
- UNICODE_TO_UNICODE_NFKC
- UNICODE_TO_UNICODE_NFKD

Because normalization functions can cause errors due to not preserving BMP characters, you should use the TRANSLATE_CHK function to verify a clean translation.

Pad Character Translation

The following translations do not translate the pad character.

source **_TO_*****target***	**Pad Character Translation**
GRAPHIC_TO_UNICODE	A GRAPHIC string that includes an Ideographic Space is translated to a UNICODE string with an Ideographic Space.
UNICODE_TO_GRAPHIC	A UNICODE string with a Space character generates an error when translated to GRAPHIC.

If you require pad character translation, use one of the following translations.

source **_TO_*****target***	**Pad Character Translation**
GRAPHIC_TO_UNICODE_PadSpace	Converts all occurrences of Ideographic Space (U+3000) to Space (U+0020).
UNICODE_TO_ GRAPHIC_PadGraphic	Converts all occurrences of Space to Ideographic Space.

Other characters are not affected. Note that the position of a character does not affect the translation, so not only trailing pad characters are modified.

Migration

During the migration process, any GRAPHIC data in the old form must be translated to the new canonical form. Note that this involves converting the pad characters from Null (U+0000) to Ideographic Space (U+3000).

Implicit Character Data Type Conversion

TRANSLATE performs implicit conversion if the *string* server character set does not match the type implied by *source_repertoire_name.*

An implicit conversion generates an error if a character from *character_string_expression* has no corresponding character in the *source_repertoire_name* type. This holds regardless of whether you specify the WITH ERROR option.

For example, the following function first translates the string from UNICODE to LATIN, because Vantage treats literals as UNICODE, and then translates the string from LATIN to KANJISJIS. However, the translation generates an error because the last character is not in the LATIN repertoire.

```
   ...
   TRANSLATE('abc ' USING LATIN_TO_KanjiSJIS WITH ERROR)   ...
```

To circumvent the problem if error character substitution is acceptable, specify two levels of translation, as used in the following example.

```
   ...
   TRANSLATE((TRANSLATE(_UNICODE 'abc' USING UNICODE_TO_LATIN WITH
ERROR)) USING LATIN_TO_KanjiSJIS WITH ERROR)
```

Examples

Function	Result	Type of the Result
TRANSLATE('abc' USING UNICODE_TO_LATIN)	'abc'	VARCHAR(3) CHARACTER SET LATIN
TRANSLATE('abc' USING UNICODE_TO_ UNICODE_Fullwidth)	'abc '	VARCHAR(3) CHARACTER SET UNICODE
TRANSLATE('abc ' USING UNICODE_TO_LATIN WITH ERROR) where ε represents the designated error character for LATIN (0x1A).	'abcε '	VARCHAR(4) CHARACTER SET LATIN

Related Information

- For details about *target_repertoire_name* expression, see Supported Translations Between Character Sets.
- For details about errors, see Error Characters Assigned by the WITH ERROR Option.
- For details on the mappings that Vantage uses for the TRANSLATE function, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.
- For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For information on implicit type conversion of UDTs, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details, see Supported Translations Between Character Sets.
- For details on mappings or on which characters are converted to U+0020, see *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

TRANSLATE_CHK

Determines if a TRANSLATE conversion can be performed without producing errors; returns an integer test result. Use TRANSLATE_CHK to filter untranslatable strings. You can choose to select translatable strings only, or untranslatable strings only, depending on how you form your SELECT statement.

Result	Meaning
0	The string can be translated without error.
NULL	The string result is null.
anything else	The position of the first character in the string causing a translation error. The value is a logical position for arguments of type LATIN, UNICODE, KANJISJIS, and GRAPHIC. The value is a physical position for arguments of type KANJI1.

TRANSLATE_CHK Function Syntax

```
TRANSLATE_CHK (
  character_string_expression
  USING source_repertoire_name
  [ _encoding ]
  TO target_repertoire_name
  [ _suffix ]
)
```

Syntax Elements

character_string_expression

A character string or character string expression for which the hexadecimal representation is to be returned.

If *character_string_expression* is not a character type, an error is returned.

source_repertoire_name

The source character set of the string to translate.

A value of LOCALE can be specified for *source_repertoire_name* to translate a character string from LATIN or KANJI1 to UNICODE using a source repertoire determined by the language support mode of the system and the client character set of the session.

_encoding

A literal for translating from KANJI1 to UNICODE that indicates a specific encoding of KANJI1.

The *_encoding* option is not allowed if LOCALE is specified for *source_repertoire_name* or *target_repertoire_name*.

If the translation is from these character sets:

- KatakanaEBCDIC
- KanjiEBCDIC5026_0I

- KanjiEBCDIC5038_0I

Use the following value for _encoding: _KanjiEBCDIC

KanjiEUC_0U, use the following value for _encoding: _KanjiEUC

KanjiShiftJIS_0S, use the following value for _encoding: _KANJISJIS

ASCII or EBCDIC, use the following value for _encoding: _SBC

target_repertoire_name

The target character set of the string to translate.

A value of LOCALE can be specified for *target_repertoire_name* to translate a character string from UNICODE to LATIN or KANJI1 using a target repertoire determined by the language support mode of the system and the client character set of the session.

_suffix

Specifies that the translation maps some source characters to semantically different characters.

For example, a translation that specifies the _Halfwidth suffix maps any character with a halfwidth variant to that variant, and all fullwidth variants to their non-fullwidth counterparts.

The *_suffix* option also indicates the form of character data translated from UNICODE to the KANJI1 server character set, for example, _KanjiEUC.

Valid values are:

- _KanjiEBCDIC
- _KanjiEUC
- _KANJISJIS
- _SBC
- _PadSpace
- _PadGraphic
- _Fullwidth
- _Halfwidth
- _FoldSpace
- _VarGraphic

The *_suffix* option is not allowed if LOCALE is specified for *source_repertoire_name* or *target_repertoire_name*.

Argument Types

Use TRANSLATE_CHK on character strings and character string expressions.

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

Implicit type conversion of UDTs for system operators and functions, including TRANSLATE_CHK, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

Default attributes for TRANSLATE_CHK (*string* USING *source* _TO_*target*) are:

Data Type	Heading
INTEGER	Translate_Chk(*string* using *source* _to_*target*)

If the *string* USING *source* _TO_*target* argument is *null*, the result is *null*.

TRANSLATE_CHK Usage Notes

Checking UNICODE Normalization Form Translations

When using TRANSLATE_CHK to verify UNICODE normalization form translations, any valid Unicode string can be translated to any of the normalization forms. A successful result (0) is expected unless the compatibility ideographs U+FA6C, U+FACF, U+FAD0, U+FAD1, U+FAD5, U+FAD6, or U+FAD7 are present, because these characters normalize outside the BMP, that is, outside the range U+10000 to U+10FFF.

For all normalization forms, these characters normalize as follows:

Ideograph	Normalized Form
U+FA6C	U+242EE
U+FACF,	U+2284A
U+FAD0	U+22844
U+FAD1	U+233D5
U+FAD5	U+25249

Ideograph	Normalized Form
U+FAD6	U+25CD0
U+FAD7	U+27ED3

Examples

Example

Function	Result
TRANSLATE_CHK(‘abc’ USING UNICODE_TO_LATIN)	0
TRANSLATE_CHK(‘abc ’ USING UNICODE_TO_LATIN)	4

Consider the following table definition:

```
CREATE TABLE table_1
  (cunicode CHARACTER(64) CHARACTER SET UNICODE);
```

To find all values in cunicode that can be translated to LATIN, use the following statement:

```
SELECT cunicode
FROM table_1
WHERE TRANSLATE_CHK(cunicode USING Unicode_TO_Latin) = 0;
```

Consider the following table definitions:

```
CREATE TABLE table_1
  (ckanji1 VARCHAR(20) CHARACTER SET KANJI1);

CREATE TABLE table_2
 (cunicode CHARACTER(20) CHARACTER SET UNICODE);
```

Assume table_1 is populated from the KanjiEUC client character set.

To translate the data in ckanji1 in table_1 to UNICODE, and populate table_2 with translations that have no errors, use the following statement:

```
INSERT INTO table_2
SELECT TRANSLATE(ckanji1 USING Kanji1_KanjiEUC_TO_Unicode)
```

```
FROM table_1
WHERE TRANSLATE_CHK(ckanji1 USING Kanji_KanjiEUC_TO_Unicode) = 0;
```

After converting column ckanji1 in table_1 to column cunicode in table_2, you want to find all the fields in table_1 that could not be translated.

```
SELECT ckanji1
FROM table_1
WHERE TRANSLATE_CHK(ckanji1 USING Kanji1_KanjiEUC_TO_Unicode) <> 0;
```

Example

Consider the following table definition:

```
CREATE TABLE table_1
  (cunicode CHARACTER(64) CHARACTER SET UNICODE);
```

To find all values in cunicode that can be translated to LATIN, use the following statement:

```
SELECT cunicode
FROM table_1
WHERE TRANSLATE_CHK(cunicode USING Unicode_TO_Latin) = 0;
```

Example

Consider the following table definitions:

```
CREATE TABLE table_1
  (ckanji1 VARCHAR(20) CHARACTER SET KANJI1);

CREATE TABLE table_2
 (cunicode CHARACTER(20) CHARACTER SET UNICODE);
```

Assume table_1 is populated from the KanjiEUC client character set.

To translate the data in ckanji1 in table_1 to UNICODE, and populate table_2 with translations that have no errors, use the following statement:

```
INSERT INTO table_2
SELECT TRANSLATE(ckanji1 USING Kanji1_KanjiEUC_TO_Unicode)
FROM table_1
WHERE TRANSLATE_CHK(ckanji1 USING Kanji_KanjiEUC_TO_Unicode) = 0;
```

Example

After converting column ckanji1 in table_1 to column cunicode in table_2, you want to find all the fields in table_1 that could not be translated.

```
SELECT ckanji1
FROM table_1
WHERE TRANSLATE_CHK(ckanji1 USING Kanji1_KanjiEUC_TO_Unicode) <> 0;
```

Related Information

- For information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.
- For information on implicit type conversion of UDTs, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

TRIM

Takes a character or byte *string_expression* argument, trims the specified pad characters or bytes, and returns the trimmed string.

TRIM Function Syntax

```
TRIM (
  [ { BOTH | TRAILING | LEADING } [ trim_expression ] FROM ]
  [ character_set ]
  string_expression
)
```

Syntax Elements

BOTH
TRAILING
LEADING

Specifies how to trim the specified trim character or byte from *string_expression*.

- BOTH means trim both trailing and leading characters or bytes.
- TRAILING means trim only trailing characters or bytes.
- LEADING mens trim only leading characters or bytes.

If you omit this option, the default is BOTH, and the default trim character is a null byte for byte types and a pad character for character types.

trim_expression

The specific character or byte to trim from the head, tail, or both, of *string_expression*.

The expression must evaluate to a single character.

You cannot specify *trim_expression* without also specifying BOTH, TRAILING, or LEADING.

You cannot specify a *trim_expression* of type KANJI1, nor can you apply a *trim_expression* to a *string_expression* of type KANJI1.

character_set

The name of the server character set to associate with the string expression.

Valid values are:

- _Latin, which is the LATIN server character set
- _Unicode, which is the UNICODE server character set
- _KanjiSJIS, which is the KANJISJIS server character set
- _Graphic, which is the GRAPHIC server character set

string_expression

A byte or character string or string expression to be trimmed.

Argument Types and Rules

The *trim_expression* argument must evaluate to a single byte that has a byte data type or single character that has a character data type.

TRIM operates on the following types of *string_expression* arguments:

- Character, except for CLOB
- Byte, except for BLOB
- Numeric

  If a numeric expression is used as the *string_expression* argument, it is converted implicitly to CHARACTER type.
- UDTs that have implicit casts to any of the following predefined types:
  - Character
  - Numeric
  - Byte
  - DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

Implicit type conversion of UDTs for system operators and functions, including TRIM, is a Teradata extension to the ANSI SQL standard.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

Here are the default result type and attributes for TRIM(*string_expression*):

- If *string_expression* is a byte string, the result type is VARBYTE.
- If the *string_expression* is a numeric expression or character string, the result type is VARCHAR.

It is possible for the length of the result to be zero.

The server character set of the result is the same as the argument.

If the *string_expression* argument is null, the result is null.

TRIM Usage Notes

Concatenation With TRIM

The TRIM function is typically used with the concatenation operator to remove trailing pad characters or trailing bytes containing binary 00 from the concatenated string.

If the TRIM function is specified for character data types, leading, trailing, or leading and trailing pad characters are suppressed in the concatenated string, according to which syntax is used.

Examples

Example

If the Names table includes the columns first_name and last_name, which contain the following information:

```
first_name (CHAR(12)) has a value of 'Mary        '
last_name (CHAR(12)) has a value of  'Jones       '
```

then this statement:

```
SELECT TRIM (BOTH FROM last_name) || ', ' || TRIM(BOTH FROM first_name)
FROM names ;
```

returns the following string (note that the seven trailing blanks at the end of string Jones, and the eight trailing blanks at the end of string Mary are not included in the result):

```
'Jones, Mary'
```

If the TRIM function is removed, the statement:

```
SELECT last_name || ', ' || first_name
FROM names;
```

returns trailing blanks in the string:

```
'Jones       , Mary        '
```

Example

Assume column a is BYTE(4) and column b is VARBYTE(10).

If these columns contained the following values:

```
a             b
------------  ---------
78790000      43440000
68690000      3200
12550000      332200
```

then this function:

```
SELECT TRIM (TRAILING FROM a) || TRIM (TRAILING FROM b) FROM ...
```

returns:

```
78794344
686932
12553322
```

Example

The following statement trims trailing SEMICOLON characters from the specified string.

```
SELECT TRIM( TRAILING ';' FROM textfield) FROM texttable;
```

Example: Using TRIM Functions

The following table illustrates several more complicated TRIM functions.

Function	Result
SELECT TRIM(LEADING 'a' FROM 'aaabcd');	'bcd'

CREATE TABLE t2
  (i1 INTEGER, c1 CHAR(6), c2 CHAR(1));
INSERT t2 (1, 'aaabcd', 'a');
SELECT TRIM(LEADING c2 FROM c1) FROM t2;
	'bcd'

CREATE TABLE t3
  (i1 INTEGER, c1 CHAR(6) CHAR SET UNICODE);
INSERT t3 (1, _Unicode '006100610061006200630064'XC);
SELECT TRIM(LEADING _Unicode '0061'XC FROM t3.c1);
	'bcd'

SELECT TRIM(_Unicode 'ΔΔ abc 年 ΔΔΔ ');	'abc 年 '
SELECT TRIM(_Unicode 'ΔΔ abc 年 ΔΔ Δ ');	'abc 年 ΔΔ ' Δ (GRAPHIC pad) is not removed.

CREATE TABLE t1
  (c1 CHARACTER(6) CHARACTER SET GRAPHIC);
INSERT t1 (_Graphic 'abc 年 ΔΔ ');
SELECT TRIM(c1) from t1;
	'abc 年 '
	Δ (GRAPHIC pad) is removed because the operand of the TRIM function is of type GRAPHIC.

Related Information

For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

UPPER/UCASE

Returns a character string identical to *character_string_expression*, except that all lowercase letters are replaced by their uppercase equivalents.

UPPER/UCASE does not convert multibyte characters to uppercase in the KANJI1 server character set.

UPPER/UCASE Function Syntax

```
{ UPPER | UCASE } ( character_string_expression )
```

Syntax Elements

character_string_expression

A character string or character string expression for which all lowercase characters are to be replaced by their uppercase equivalents.

Argument Types

UPPER/UCASE is valid only for character strings and character string expressions. The function does not accept CLOBs and non-character arguments.

By default, Vantage performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Implicit type conversion of UDTs for system operators and functions, including UPPER/UCASE, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

ANSI Compliance

This statement is ANSI SQL:2011 compliant.

Result Type and Attributes

Here are the default result type and attributes for UPPER(*arg*)/UCASE(*arg*).

Data Type	Heading
Same type as *arg*	UPPER(*arg*)/UCASE(*arg*)

Usage Notes

The UPPER/UCASE function allows users who want ANSI portability to have case blind comparisons with ANSI-compliant syntax.

This function is treated the same as the following obsolete form:

```
 expression
(UPPERCASE)
```

You can also replace characters with lowercase equivalents. For more information, see LOWER.

Examples

Example: Using a Table Definition with CASESPECIFIC Attributes

Consider the following table definition where the character columns have CASESPECIFIC attributes:

```
CREATE TABLE employee
  (last_name CHAR(32) CASESPECIFIC
  ,city      CHAR(32) CASESPECIFIC
  ,emp_id    CHAR(9)  CASESPECIFIC
  ,emp_ssn   CHAR(9)  CASESPECIFIC);
```

To compare on a case blind basis:

```
SELECT emp_id
FROM employee
WHERE UPPER(emp_id) = UPPER(emp_ssn);
```

To compare with a string literal:

```
SELECT emp_id
FROM employee
WHERE UPPER(city) = 'MINNEAPOLIS';
```

Teradata SQL also has the data type attribute NOT CASESPECIFIC, which allows case blind comparisons. Note that the data type attributes CASESPECIFIC and NOT CASESPECIFIC are Teradata extensions to the ANSI standard.

Example: Using UPPER to Store Values

The use of UPPER to store values is shown in the following examples:

```
INSERT INTO names
SELECT UPPER(last_name),UPPER(first_name)
   FROM newnames;
```

or

```
USING (last_name CHAR(20),first_name CHAR(20))
INSERT INTO names
(UPPER(:last_name), UPPER(:first_name));
```

Example: Converting Single Byte Characters to Uppercase

This example shows that in the KANJI1 server character set, only single byte characters are converted to uppercase.

SELECT UPPER('abcd 年');

The result is 'ABCD 年'.

Related Information

- For information on implicit type conversion, see "Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Teradata Vantage™ - Database Utilities*, B035-1102.

VARGRAPHIC

Returns the VARGRAPHIC representation of the character data in *character_string_expression*.

VARGRAPHIC Function Syntax

```
VARGRAPHIC ( character_string_expression )
```

Syntax Elements

character_string_expression

A character string or character string expression for which the VARGRAPHIC representation is to be returned.

Argument Types

VARGRAPHIC operates on the following types of arguments:

- Character, except for CLOB
- Numeric

If the argument is numeric, it is implicitly converted to a character type.

- UDTs that have implicit casts to any of the following predefined types:
    - Character
    - Numeric
    - DATE

    To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause.

    Implicit type conversion of UDTs for system operators and functions, including VARGRAPHIC, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE.

ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

Result Type and Attributes

Here are the default result type and attributes for VARGRAPHIC(*arg*):

Data Type	Heading
VARCHAR(*n*) CHARACTER SET GRAPHIC	Vargraphic(*arg*)

VARGRAPHIC Usage Notes

Rules

VARGRAPHIC reports an error if the session character set is UTF8 or a single-byte character set, such as ASCII. If the argument is of type KANJI1, the only valid session character set is KanjiEBCDIC.

All characters in the string are converted into one or more graphics that are valid for the character set of the current session. For more information, see VARGRAPHIC Function Conversion Tables.

The argument cannot be of type GRAPHIC.

A result that exceeds the maximum length of a VARCHAR CHARACTER SET GRAPHIC data type generates an error.

VARGRAPHIC cannot appear as the first argument in a user-defined method invocation.

Specific rules apply to the server character set of *character_string_expression*.

IF the string specifies this server character set …	THEN VARGRAPHIC operates as follows …
KANJI1	Shift-Out/Shift-In characters in the *character_string_expression* do not appear in the result string. They are required only to indicate the transition between single byte characters and multibyte characters. Improperly placed Shift-Out/Shift-Ins are replaced by the illegal character for the character set of the session. The SPACE CHARACTER translates to the IDEOGRAPHIC SPACE CHARACTER.
UNICODE	• Characters with fullwidth representation in the UNICODE compatibility zone translate to that fullwidth representation. • Halfwidth characters from the compatibility zone translate to the corresponding characters outside the compatibility zone. • The SPACE CHARACTER translates to the IDEOGRAPHIC SPACE CHARACTER. • The control characters U+0000 - U+001F and character U+007F are converted to the VARGRAPHIC error character. • Other characters are left untranslated.
anything else	The result is as if string were first converted to UNICODE and then translated according to the rules listed for UNICODE above.

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible. For more information, see *KANJI1 Character Set* in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

VARGRAPHIC Function Conversion Tables

The following table shows the translation of a single byte character to its double byte equivalent by the VARGRAPHIC function. Values in columns 2, 3, and 4 are hexadecimal.

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
00		FEFE	FEFE
01		FEFE	FEFE
02		FEFE	FEFE
03		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	**JIS X 0201 Printable Character**	**KanjiEBCDIC 5026/5035**	**Katakana EBCDIC**
04		FEFE	FEFE
05		FEFE	FEFE
06		FEFE	FEFE
07		FEFE	FEFE
08		FEFE	FEFE
09		FEFE	FEFE
0A		FEFE	FEFE
0B		FEFE	FEFE
0C		FEFE	FEFE
0D		FEFE	FEFE
0E	For KanjiEBCDIC, the SO /SI is not placed in the output of vargraphic function. In particular, a single SO character will not generate any output, or strictly speaking will generate a string with 0 length.	N/A	N/A
0F	For KanjiEBCDIC, the SO/SI is not placed in the output of vargraphic function. However, if the SI character appears in the input without matching SO, we will generate FEFE for that SI.	FEFE	FEFE
10		FEFE	FEFE
11	£ (Pound Sterling sign)	424A	424A
12	¬ (Logical NOT)	425F	FEFE
13	\	43E0	FEFE
14	~	43A1	FEFE
15		FEFE	FEFE
16		FEFE	FEFE
17		FEFE	FEFE
18		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
19		FEFE	FEFE
1A		FEFE	FEFE
1B		FEFE	FEFE
1C		FEFE	FEFE
1D		FEFE	FEFE
1E		FEFE	FEFE
1F		FEFE	FEFE
20		4040	4040
21	!	425A	425A
22	"	4472	4472
23	#	427B	427B
24	$	42E0	42E0
25	%	426C	426C
26	&	4250	4250
27	'	4471	4471
28	(	424D	424D
29	)	425D	425D
2A	*	425C	425C
2B	+	424E	424E
2C	,	426B	426B
2D	-	4260	4260
2E	.	424B	424B
2F	/	4261	4261
30	0	42F0	42F0
31	1	42F1	42F1
32	2	42F2	42F2
33	3	42F3	43F3

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
34	4	42F4	42F4
35	5	42F5	42F5
36	6	42F6	42F6
37	7	42F7	42F7
38	8	42F8	42F8
39	9	42F9	42F9
3A	:	427A	427A
3B	;	425E	425E
3C	<	424C	424C
3D	=	427E	427E
3E	>	426E	426E
3F	?	426F	426F
40	@	427C	427C
41	A	42C1	42C1
42	B	42C2	42C2
43	C	42C3	42C3
44	D	42C4	42C4
45	E	42C5	42C5
46	F	42C6	42C6
47	G	42C7	42C7
48	H	42C8	42C8
49	I	42C9	42C9
4A	J	42D1	42D1
4B	K	42D2	42D2
4C	L	42D3	42D3
4D	M	42D4	42D4
4E	N	42D5	42D5

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
4F	O	42D6	42D6
50	P	42D7	42D7
51	Q	42D8	42D8
52	R	42D9	42D9
53	S	42E2	42E2
54	T	42E3	42E3
55	U	42E4	42E4
56	V	42E5	42E5
57	W	42E6	42E6
58	X	42E7	42E7
59	Y	42E8	42E8
5A	Z	42E9	42E9
5B	[	4444	FEFE
5C	\	425B	425B
5D	]	4445	FEFE
5E	^	4470	425F
5F	_	426D	426D
60	`	4279	FEFE
61	a	4281	FEFE
62	b	4282	FEFE
63	c	4283	FEFE
64	d	4284	FEFE
65	e	4285	FEFE
66	f	4286	FEFE
67	g	4287	FEFE
68	h	4288	FEFE
69	i	4289	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
6A	j	4291	FEFE
6B	k	4292	FEFE
6C	l	4293	FEFE
6D	m	4294	FEFE
6E	n	4295	FEFE
6F	o	4296	FEFE
70	p	4297	FEFE
71	q	4298	FEFE
72	r	4299	FEFE
73	s	42A2	FEFE
74	t	42A3	FEFE
75	u	42A4	FEFE
76	v	42A5	FEFE
77	w	42A6	FEFE
78	x	42A7	FEFE
79	y	42A8	FEFE
7A	z	42A9	FEFE
7B	{	42C0	FEFE
7C	|	424F	424F
7D	}	42D0	FEFE
7E	- (Overline)	42A1	42A1
7F		FEFE	FEFE
80		FEFE	FEFE
81		FEFE	FEFE
82		FEFE	FEFE
83		FEFE	FEFE
84		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
85		FEFE	FEFE
86		FEFE	FEFE
87		FEFE	FEFE
88		FEFE	FEFE
89		FEFE	FEFE
8A		FEFE	FEFE
8B		FEFE	FEFE
8C		FEFE	FEFE
8D		FEFE	FEFE
8E		FEFE	FEFE
8F		FEFE	FEFE
90		FEFE	FEFE
91		FEFE	FEFE
92		FEFE	FEFE
93		FEFE	FEFE
94		FEFE	FEFE
95		FEFE	FEFE
96		FEFE	FEFE
97		FEFE	FEFE
98		FEFE	FEFE
99		FEFE	FEFE
9A		FEFE	FEFE
9B		FEFE	FEFE
9C		FEFE	FEFE
9D		FEFE	FEFE
9E		FEFE	FEFE
9F		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
A0		FEFE	FEFE
A1	Ideographic period	4341	4341
A2	Left corner bracket	4342	4342
A3	Right corner bracket	4343	4343
A4	Ideographic comma	4344	4344
A5	Katakana middle dot	4345	4345
A6	Katakana letter WO	4346	4346
A7	Katakana letter A	4347	4347
A8	Katakana letter small I	4348	4348
A9	Katakana letter small U	4349	4349
AA	Katakana letter small E	4351	4351
AB	Katakana letter small O	4352	4352
AC	Katakana letter small YA	4353	4353
AD	Katakana letter small YU	5454	4354
AE	Katakana letter small YO	4355	4355
AF	Katakana letter small WO	4356	4356
B0	Katakana-Hiragana prolonged sound mark	4358	4358
B1	A	4381	4381
B2	I	4382	4382
B3	U	4383	4383
B4	E	4384	4384
B5	O	4385	4385
B6	KA	4386	4386
B7	KI	4387	4387
B8	KU	4388	4388
B9	KE	4389	4389
BA	KO	438A	438A

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
BB	SA	438C	438C
BC	SHI	438D	438D
BD	SU	438E	438E
BE	SEE	438F	438F
BF	SO	4390	4390
C0	TAI	4391	4391
C1	CHI	4392	4392
C2	TSU	4393	4393
C3	TE	4394	4394
C4	TO	4395	4395
C5	NA	4396	4396
C6	NI	4397	4397
C7	NU	4398	4398
C8	NE	4399	4399
C9	NO	439A	439A
CA	HA	439D	439D
CB	HI	439E	439E
CC	FU	439F	439F
CD	HE	43A2	43A2
CE	HO	43A3	43A3
CF	MA	43A4	43A4
D0	MI	43A5	43A5
D1	MU	43A6	43A6
D2	ME	43A7	43A7
D3	MO	43A8	43A8
D4	YA	43A9	43A9
D5	YU	43AA	43AA

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
D6	YO	43AC	43AC
D7	RA	43AD	43AD
D8	RI	43AE	43AE
D9	RU	43AF	43AF
DA	RE	43BA	43BA
DB	RO	43BB	43BB
DC	WA	43BC	43BC
DD	N	43BD	43BD
DE	Katakana-Hiragana voiced sound mark	43BE	43BE
DF	Katakana-Hiragana semi-voice sound mark	43BF	43BF
E0		FEFE	FEFE
E1		FEFE	FEFE
E2		FEFE	FEFE
E3		FEFE	FEFE
E4		FEFE	FEFE
E5		FEFE	FEFE
E6		FEFE	FEFE
E7		FEFE	FEFE
E8		FEFE	FEFE
E9		FEFE	FEFE
EA		FEFE	FEFE
EB		FEFE	FEFE
EC		FEFE	FEFE
ED		FEFE	FEFE
EE		FEFE	FEFE
EF		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	**JIS X 0201 Printable Character**	**KanjiEBCDIC 5026/5035**	**Katakana EBCDIC**
F0		FEFE	FEFE
F1		FEFE	FEFE
F2		FEFE	FEFE
F3		FEFE	FEFE
F4		FEFE	FEFE
F5		FEFE	FEFE
F6		FEFE	FEFE
F7		FEFE	FEFE
F8		FEFE	FEFE
F9		FEFE	FEFE
FA		FEFE	FEFE
FB		FEFE	FEFE
BC		FEFE	FEFE
FD		FEFE	FEFE
FE		FEFE	FEFE
FF		FEFE	FEFE

Examples

Example

The following table shows examples of converting strings that use the UNICODE and LATIN server character sets to GRAPHIC data.

Function	Result
VARGRAPHIC('92年 abcΔ ')	'92 年 abc Δ '
VARGRAPHIC('abc')	'abc '

Consider the following table definition with two character columns that use the KANJI1 server character set:

```
   CREATE TABLE t1
     (c1 VARCHAR(12) CHARACTER SET KANJI1
     ,c2 VARCHAR(12) CHARACTER SET KANJI1);
```

Use the KanjiEBCDIC client character set and insert the following strings:

```
   INSERT t1 ('de
f', 'gH<ABC
>X
');
```

Convert the strings to GRAPHIC data:

Function	Result
SELECT VARGRAPHIC (c1) FROM t1;	'def '
SELECT VARGRAPHIC (c2) FROM t1;	'gH ABCX ' (The single byte Hankaku Katakana X is converted to double byte X .)

Example

Consider the following table definition with two character columns that use the KANJI1 server character set:

```
   CREATE TABLE t1
     (c1 VARCHAR(12) CHARACTER SET KANJI1
     ,c2 VARCHAR(12) CHARACTER SET KANJI1);
```

Use the KanjiEBCDIC client character set and insert the following strings:

```
   INSERT t1 ('de
f', 'gH<ABC
>X
');
```

Convert the strings to GRAPHIC data:

Function	Result
SELECT VARGRAPHIC (c1) FROM t1;	'def '
SELECT VARGRAPHIC (c2) FROM t1;	'gH ABCX '

Function	Result
	(The single byte Hankaku Katakana X is converted to double byte X .)

Related Information

- For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

16

User-Defined Functions

SQL UDF

A user-defined function is written using regular SQL expressions, and is used like a standard SQL function.

Self-referencing, forward-referencing, and circular-referencing SQL UDFs are not allowed.

SQL UDF Function Syntax

```
udf_name ( [ argument [,...] ] )
```

Syntax Elements

udf_name

The name of the SQL UDF.

argument

A valid SQL expression.

Required Privileges

You must have EXECUTE FUNCTION privilege on the function or on the database containing the function.

You can specify an SQL SECURITY clause with the DEFINER option in the CREATE/REPLACE FUNCTION statement. This option is the default for an SQL UDF. SQL SECURITY DEFINER means that when an SQL UDF is invoked, Vantage verifies that the immediate owner and the creator of the UDF have the appropriate privileges on the underlying database objects referenced in the UDF. If the creator does not exist when the privileges are checked, an error is returned.

To invoke a UDF that takes a UDT argument or returns a UDT, you must have the UDTUSAGE privilege on the SYSUDTLIB database or on the specified UDT.

ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Usage Notes

An SQL UDF is a function that is defined by a user and is written using SQL expressions. When Vantage evaluates an SQL UDF expression, it invokes the function with the arguments passed to it. The following rules apply to the arguments in the function call:

- The arguments must be comma-separated expressions in the same order as the parameters declared in the function.
- The number of arguments passed to the SQL UDF must be the same as the number of parameters declared in the function.
- The data types of the arguments must be compatible with the corresponding parameter declarations in the function and follow the precedence rules that apply to compatible types.

  To pass an argument that is not compatible with the corresponding parameter type, use CAST to explicitly convert the argument to the proper type. For more information, see "CAST in Explicit Data Type Conversions" in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- A NULL argument is compatible with a parameter of any data type. You can pass a NULL argument explicitly or by omitting the argument.
- Any form of SQL expression can be used as an argument with three important rules:
  - The SQL expression must not be a Boolean value expression (that is, a conditional expression).
  - If the expression is a nondeterministic SQL expression (expressions involving random functions and/or nondeterministic UDFs), it must not correspond to a parameter that is used more than once in the RETURN statement.
  - The SQL expression must not be a scalar subquery.

When an SQL UDF is invoked, Vantage searches for the UDF in the following locations:

- In the database specified if the function call is qualified by a database name.
- In the current database.
- In the SYSLIB database.

The result type of an SQL UDF is based on the return type specified in the RETURNS clause of the CREATE FUNCTION statement.

The default title of an SQL UDF appears as:

```
    UDF_name
(argument_list
)
```

Examples

Example: Defining the Function and Query

Consider the following function definition and query:

```
CREATE FUNCTION Test.MyUDF (a INT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a + b - c;
SELECT Test.MyUDF(t1.a1, t2.a2, t3.a3) FROM t1, t2, t3;
```

The user executing the SELECT statement must have the following privileges:

- SELECT privilege on tables t1, t2, and t3, their containing databases, or on the columns t1.a1, t2.a2, and t3.a3.
- EXECUTE FUNCTION privilege on MyUDF or on the database named Test.

The privileges of the creator or owner are not checked since the UDF does not reference any database objects in its definition.

Example: Referencing an External UDF

In this example, the SQL UDF named MySQLUDF references an external UDF named MyExtUDF in the RETURN statement.

Consider the following function definition and query:

```
CREATE FUNCTION Test.MySQLUDF (a INT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a + b * MyExtUDF(a, b) - c;
SELECT Test.MySQLUDF(t1.a1, t2.a2, t3.a3) FROM t1, t2, t3;
```

The user executing the SELECT statement must have the following privileges:

- SELECT privilege on tables t1, t2, and t3, their containing databases, or on the columns t1.a1, t2.a2, and t3.a3.
- EXECUTE FUNCTION privilege on MySQLUDF or on the database named Test.

Because the SQL UDF references MyExtUDF, the following privileges are also checked:

- The creator of MySQLUDF must exist and have the EXECUTE FUNCTION privilege on MyExtUDF or its containing database.
- The database named Test (the immediate owner of MySQLUDF) must have the EXECUTE FUNCTION privilege on MyExtUDF or its containing database.

Example: Invoking the SQL UDF

In this example, invocations of the SQL UDF named MyUDF2 are passed as arguments to the SQL UDF named MyUDF1.

```
CREATE FUNCTION test.MyUDF1 (a INT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
COLLATION INVOKER
INLINE TYPE 1
RETURN a * b * c;
CREATE FUNCTION test.MyUDF2 (d INT, e INT, f INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
COLLATION INVOKER
INLINE TYPE 1
RETURN d + e + f;
SELECT test.MyUDF1(test.MyUDF2(t1.a1, 1, 2),
       test.MyUDF2(t1.b1, 2, 3), 5) FROM t1;
```

Example: Invoking Compatible Argument Data Types

In this example, the UDF invocation argument data type (BYTEINT) is not the same as that of the corresponding UDF parameter data type (INTEGER) since the size of the argument data type is less than the UDF parameter data type. However, because the two data types are compatible and a BYTEINT argument can fit inside an INTEGER parameter, this is allowed.

```
CREATE FUNCTION test.MyUDF (a INT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
```

```
COLLATION INVOKER
INLINE TYPE 1
RETURN a * b * c;
CREATE TABLE t1 (a1 BYTEINT, b1 INT);
SELECT test.MyUDF(t1.a1, t1.b1, 2) FROM t1;
```

Example: Invoking Argument Data Types of Different Sizes

In this example, the UDF invocation argument data type (INTEGER) is not the same as that of the corresponding UDF parameter data type (BYTEINT) since the size of the argument data type is greater than the UDF parameter data type. Although the two data types are compatible, an INTEGER argument cannot fit inside a BYTEINT parameter, so an error is returned.

```
CREATE FUNCTION test.MyUDF (a BYTEINT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
COLLATION INVOKER
INLINE TYPE 1
RETURN a * b * c;
CREATE TABLE t1 (a1 INT, b1 INT);
SELECT test.MyUDF(t1.a1, t1.b1, 2) FROM t1;
```

The following error is returned:

```
Failure 5589: Function "test.MyUDF" does not exist.
```

To avoid the error, the caller must explicitly cast the value of t1.a1 to BYTEINT as follows:

```
SELECT test.MyUDF(CAST(t1.a1 AS BYTEINT), t1.b1, 2) FROM t1;
```

Related Information

- For more information about CREATE FUNCTION or REPLACE FUNCTION, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 or *Teradata Vantage™ - Database Administration*, B035-1093.
- For more information about EXECUTE FUNCTION and UDTUSAGE privileges, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.
- For details about data type compatibility, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

- For details regarding UDF search resolution, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

17

Elastic TCore Functions

GetSystemTCore

GetSystemTCore returns the current TCore setting of the system.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

The EXECUTE privilege on GetSystemTCore is required in order to run it.

GetSystemTCore Syntax

```
[SYSLIB.] GetSystemTCore ()
```

Syntax Elements

SYSLIB

Name of the database where the function is located.

GetSystemTCore Example

Example

```
SELECT * FROM TABLE(SYSLIB.getSystemTcore()) AS t;

*** Query completed. One row found. 7 columns returned.

*** Total elasped time was 1 second.

StatusFlag 1
ErrorMsg Success for get systemtcore
BaselineTcore 75
CurrentTcore 100
MinTcore 0
MaxTCore 102
```

SetSystemTCore

SetSystemTCore sets the TCore value of the system.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

When using SetSystemTCore, keep the following in mind:

- Teradata monitors use of Elastic TCore. Customers that use more TCore than they have purchased to date may incur additional charges as a result.
- The EXECUTE privilege on SetSystemTCore is required in order to run it.
- It is supported only on Intelliflex 2.1 and above systems.
- It is used only in the FROM clause of a SELECT request.
- It can be invoked only with constant expression input arguments.

SetSystemTCore Example

```
SELECT * FROM TABLE(SYSLIB.SetSystemTcore(75)) AS t;
```

SetSystemTCore Syntax

```
[SYSLIB.] SetSystemTCore ( SystemTCoreToSet )
```

Syntax Elements

SYSLIB

Name of the database where the function is located.

SystemTCoreToSet

An integer representing the amount of TCore to give the system. To see the range of valid values, run the GetSystemTCore function.

A

Notation Conventions

How to Read Syntax

This document uses the following syntax conventions.

Syntax Convention	Meaning
`KEYWORD`	Keyword. Spell exactly as shown. Many environments are case-insensitive. Syntax shows keywords in uppercase unless operating system restrictions require them to be lowercase or mixed-case.
variable	Variable. Replace with actual value.
number	String of one or more digits. Do not use commas in numbers with more than three digits. Example: 10045
`[ x ]`	`x` is optional.
`[ x | y ]`	You can specify `x`, `y`, or nothing.
`{ x | y }`	You must specify either `x` or `y`.
`x [...]`	You can repeat `x`, separating occurrences with spaces. Example: `x x x` See note after table.
`x [,...]`	You can repeat `x`, separating occurrences with commas. Example: `x, x, x` See note after table.
`x [`*delimiter*`...]`	You can repeat `x`, separating occurrences with specified delimiter. Examples: • If *delimiter* is semicolon: `x; x; x` • If *delimiter* is `{,|OR}`, you can do either of the following: ◦ `x, x, x` ◦ `x OR x OR x` See note after table.

Note:

You can repeat only the immediately preceding item. For example, if the syntax is:

```
KEYWORD x [...]
```

You can repeat `x`. Do not repeat `KEYWORD`.

If there is no white space between x and the delimiter, the repeatable item is x and the delimiter. For example, if the syntax is:

```
[ x, [...] ] y
```

- You can omit x: `y`
- You can specify x once: `x, y`
- You can repeat x and the delimiter: `x, x, x, y`

Character Shorthand Notation Used in This Document

This document uses the Unicode naming convention for characters. For example, the lowercase character 'a' is more formally specified as either LATIN CAPITAL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the document, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings:

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a-z A-Z 0-9	Any	Any single byte Latin letter or digit.
a-z A-Z 0-9	Any	Any fullwidth Latin letter or digit.

Symbol	Encoding	Meaning
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by $_{ss3}$.
$\underline{\mathbf{I}}$	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by $_{ss2}$, forming an individual multibyte character.
$\underline{\Delta}$	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss $_2$	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss $_3$	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set:

LMN<**TEST**>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

Pad Characters

The following table lists the pad characters for the various character data types.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

Additional Information

Teradata Links

Link	Description
https://docs.teradata.com/	Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Sign in to access Orange Books.
https://support.teradata.com	One-stop source for Teradata community support, software downloads, and product information. Log in for customer access to: • Community support • Software updates • Knowledge articles
https://www.teradata.com/University/Overview	Teradata education network
https://support.teradata.com/community	Link to Teradata community