Teradata Database

SQL Functions, Operators, Expressions, and Predicates

Release 13.0
B035-1145-098A
November 2009

Preface

Purpose

SQL Functions, Operators, Expressions, and Predicates describes the functions, operators, expressions, and predicates of Teradata SQL.

Use this book with the other books in the SQL book set.

Audience

Application programmers and end users are the principal audience for this manual. System administrators, database administrators, security administrators, Teradata field engineers, and other technical personnel responsible for designing, maintaining, and using Teradata Database might also find this manual to be useful.

Supported Software Release

This book supports Teradata® Database 13.0.

Prerequisites

You should be familiar with basic relational database management technology and SQL. This book is not an SQL primer.

If you are not familiar with Teradata Database, read *Introduction to Teradata* before reading this book.

For information about developing applications using embedded SQL, see *Teradata Preprocessor2 for Embedded SQL Programmer Guide*.

Changes to This Book

Release	Description
Teradata Database 13.0 November 2009	• Removed information not longer applicable to the documentation of the IN/NOT IN logical predicate. • Clarified RESET WHEN example 3. • Clarified the arguments to window aggregate functions. • Clarified conversion to FLOAT, REAL, and DOUBLE PRECISION data types and truncation and rounding during conversion.
Teradata Database 13.0 April 2009	Added the following: • Clarification that UDT expressions cannot be used as input arguments to UDFs written in Java, and they cannot be used as IN and INOUT parameters of external stored procedures written in Java. • Restriction that the HASH BY or LOCAL ORDER BY clauses cannot be used in derived tables with set operators. • Information about Period data types. • Information about the CURRENT_USER and CURRENT_ROLE built-in functions. • Information about the RESET WHEN clause. • Information about the NEW VARIANT_TYPE expression for constructing dynamic UDTs. • Additional information about implicit DateTime conversions. • Clarification for determining the server character set of the result of a CASE expression. • Information on calculating the interval difference between two DateTime values. • A new chapter about UDF expressions.

Teradata Database 12.0
 	
September 2007
	• Added new rules to RANGE_N and CASE_N functions for multilevel PPI
	• Modified TRANSLATE and TRANSLATE_CHK functions to support LOCALE as an option for the source or target repertoire name
	• Documented the DEGREES, RADIANS, and STRING_CS functions
	• Changed HASHBUCKET, HASHAMP, and HASHBAKAMP functions to reflect new value for the maximum number of hash buckets
	• Added new character sets to the list of character sets in OCTET_LENGTH
	• Added the following aggregate functions to the list of window aggregate functions in Chapter 8 that support the windowing specification:
	  • CORR
	  • COVAR_POP
	  • COVAR_SAMP
	  • REGR_AVGX
	  • REGR_AVGY
	  • REGR_COUNT
	  • REGR_INTERCEPT
	  • REGR_R2
	  • REGR_SLOPE
	  • REGR_SXX
	  • REGR_SXY
	  • REGR_SYY
	  • STDDEV_POP
	  • STDDEV_SAMP
	  • VAR_POP
	  • VAR_SAMP

Additional Information

URL	Description
http://www.info.teradata.com/	Use the Teradata Information Products Publishing Library site to: • View or download a manual: 1 Under **Online Publications**, select **General Search**. 2 Enter your search criteria and click **Search**. • Download a documentation CD-ROM: 1 Under **Online Publications**, select **General Search**. 2 In the **Title or Keyword** field, enter *CD-ROM*, and click **Search**. • Order printed manuals: Under **Print & CD Publications**, select **How to Order**.
http://www.teradata.com	The Teradata home page provides links to numerous sources of information about Teradata. Links include: • Executive reports, case studies of customer experiences with Teradata, and thought leadership • Technical information, solutions, and expert advice • Press releases, mentions and media resources
http://teradatauniversitynetwork.com	Teradata University Network fosters education on data warehousing, business intelligence (BI) and database administration (DBA).

To maintain the quality of our products and services, we would like your comments on the accuracy, clarity, organization, and value of this document. Please e-mail: teradata-books@lists.teradata.com

References to Microsoft Windows and Linux

This book refers to “Microsoft Windows” and “Linux.” For Teradata Database 13.0, these references mean:

- “Windows” is Microsoft Windows Server 2003 64-bit.
- “Linux” is SUSE Linux Enterprise Server 9 and SUSE Linux Enterprise Server 10.

Table of Contents

Preface 3

Purpose 3
Audience 3
Supported Software Release 3
Prerequisites 3
Changes to This Book 4
Additional Information 5
References to Microsoft Windows and Linux 5

Chapter 1: Introduction 17

SQL Functions 17
SQL Operators 18
SQL Expressions 19
SQL Predicates 20

Chapter 2: CASE Expressions 23

CASE 23
Valued CASE Expression 24
Searched CASE Expression 27
Error Conditions 31
Rules for the CASE Expression Result Type 32
Format for a CASE Expression 37
CASE and Nulls 38
COALESCE Expression 40
NULLIF Expression 42

Chapter 3: Arithmetic Operators and Functions / Trigonometric and Hyperbolic Functions 45

Arithmetic Operators 46
Binary Arithmetic Result Data Types 47
Structure of Arithmetic Expressions 51
Arithmetic Functions 53
ABS 54
CASE_N 56
EXP 62
LN 64
LOG 66
NULLIFZERO 68
RANDOM 71
RANGE_N 75
SQRT 82
WIDTH_BUCKET 84
ZEROIFNULL 88
Trigonometric Functions
(COS, SIN, TAN, ACOS, ASIN, ATAN, ATAN2) 91
DEGREES
RADIANS 94
Hyperbolic Functions
(COSH, SINH, TANH, ACOSH, ASINH, ATANH) 97

Chapter 4: Comparison Operators 101

Comparison Operators 101
Comparison Operators in Logical Expressions 103
Comparisons That Produce TRUE Results 105
Data Type Evaluation 106
Implicit Type Conversion of Comparison Operands 108
Comparison of ANSI DateTime and Interval in USING Clause 110
Proper Forms of DATE Types in Comparisons 111
Character String Comparisons 112
Comparison of KANJI1 Characters 115
Comparison Operators and the DEFAULT Function in Predicates 117
Comparison of Period Types 118

Chapter 5: Set Operators 121

Overview of Set Operators 121
Rules for Set Operators 123
Precedence of Set Operators 124
Retaining Duplicate Rows Using the ALL Option 125
Attributes of a Set Result 125
Set Operators With Derived Tables 127
Set Operators in Subqueries 128
Set Operators in INSERT … SELECT Statements 130
Set Operators in View Definitions 131
Queries Connected by Set Operators 133
INTERSECT Operator 137
MINUS/EXCEPT Operator 140
UNION Operator 142

Chapter 6: DateTime and Interval Functions and Expressions 151

Overview 151
ANSI DateTime and Interval Data Type Assignment Rules 152
Scalar Operations on ANSI SQL:2008 DateTime and Interval Values 154
ANSI DateTime Expressions 155
ANSI Interval Expressions 160
Arithmetic Operators 168
Aggregate Functions and ANSI DateTime and Interval Data Types 170
Scalar Operations and DateTime Functions 171
Teradata Date and Time Expressions 171
Scalar Operations on Teradata DATE Values 173
ADD_MONTHS 174
EXTRACT 180

Chapter 7: Period Functions and Operators 185

BEGIN 186
END 188

LAST 190
INTERVAL 192
PRIOR 195
NEXT 197
P_INTERSECT 199
LDIFF 201
RDIFF 204
P_NORMALIZE 207
Period Value Constructor 210
Arithmetic Operators 212

Chapter 8: Aggregate Functions 215

Aggregate Functions 215
AVG 220
CORR 223
COUNT 226
COVAR_POP 231
COVAR_SAMP 234
GROUPING 237
KURTOSIS 240
MAX 242
MIN 245
REGR_AVGX 248
REGR_AVGY 251
REGR_COUNT 254
REGR_INTERCEPT 258
REGR_R2 262
REGR_SLOPE 266
REGR_SXX 270
REGR_SXY 273
REGR_SYY 276
SKEW 279
STDDEV_POP 282
STDDEV_SAMP 285
SUM 288
VAR_POP 291

VAR_SAMP 294

Chapter 9: Ordered Analytical Functions 297

Ordered Analytical Functions 298
Ordered Analytical Functions Benefits 298
Syntax Alternatives for Ordered Analytical Functions 299
Window Feature 300
Applying Windows to Aggregate Functions 307
Characteristics of Ordered Analytical Functions 308
Nesting Aggregates in Ordered Analytical Functions 311
GROUP BY Clause 314
Using Ordered Analytical Functions Examples 316
Window Aggregate Functions
(AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP) 320
CSUM 338
MAVG 341
MDIFF 344
MLINREG 347
MSUM 350
PERCENT_RANK 352
QUANTILE 356
RANK 359
RANK 362
ROW_NUMBER 365

Chapter 10: String Operator and Functions 369

Concatenation Operator 373
CHAR2HEXINT 379
INDEX 382
LOWER 388
POSITION 391
SOUNDEX 394

STRING_CS. 398
SUBSTRING/SUBSTR 401
TRANSLATE 407
TRANSLATE_CHK 416
TRIM 420
UPPER 424
VARGRAPHIC 427
VARGRAPHIC Function Conversion Tables. 430

Chapter 11: Logical Predicates. 441

Logical Predicates 441
ANY/ALL/SOME Quantifiers 445
BETWEEN/NOT BETWEEN 450
CONTAINS 451
EXISTS/NOT EXISTS. 453
IN/NOT IN 459
IS NULL/IS NOT NULL. 466
IS UNTIL_CHANGED/IS NOT UNTIL_CHANGED 468
LIKE 470
MEETS 480
OVERLAPS 482
PRECEDES 486
SUCCEEDS 488
Logical Operators and Search Conditions 490

Chapter 12: Attribute Functions 495

BYTES 496
CHARACTER_LENGTH 498
CHARACTERS 501
DEFAULT 503
FORMAT 507
OCTET_LENGTH 508
TITLE 511
TYPE 512

Chapter 13: Hash-Related Functions 515

Features 515
HASHAMP 516
HASHBAKAMP 519
HASHBUCKET 522
HASHROW 525

Chapter 14: Built-In Functions 527

ACCOUNT 528
CURRENT_DATE 529
CURRENT_ROLE 531
CURRENT_TIME 533
CURRENT_TIMESTAMP 535
CURRENT_USER 537
DATABASE 538
DATE 539
PROFILE 541
ROLE 542
SESSION 545
TIME 546
USER 548

Chapter 15: UDF Expressions 551

Scalar UDF Expression 552
Aggregate UDF Expression 555
Table UDF Expression 558

Chapter 16: UDT Expressions and Methods 561

UDT Expression 562
NEW 566
NEW VARIANT_TYPE 569

Method Invocation 572

Chapter 17: Data Type Conversions 577

Forms of Data Type Conversions 577
Implicit Type Conversions 577
CAST in Explicit Data Type Conversions 582
Teradata Conversion Syntax in Explicit Data Type Conversions 585
Data Conversions in Field Mode 587
Byte Conversion 588
Character-to-Character Conversion 592
Implicit Character-to-Character Translation 595
Character-to-DATE Conversion 597
Character-to-INTERVAL Conversion 603
Character-to-Period Conversion 605
Character-to-Numeric Conversion 608
Character-to-TIME Conversion 614
Character-to-TIMESTAMP Conversion 620
Character-to-UDT Conversion 625
Character Data Type Assignment Rules 627
DATE-to-Character Conversion 628
DATE-to-DATE Conversion 632
DATE-to-Numeric Conversion 634
DATE-to-Period Conversion 637
DATE-to-TIMESTAMP Conversion 639
DATE-to-UDT Conversion 642
INTERVAL-to-Character Conversion 644
INTERVAL-to-INTERVAL Conversion 646
INTERVAL-to-Numeric Conversion 650
INTERVAL-to-UDT Conversion 652
Numeric-to-Character Conversion 654
Numeric-to-DATE Conversion 659
Numeric-to-INTERVAL Conversion 662
Numeric-to-Numeric Conversion 664
Numeric-to-UDT Conversion 668
Period-to-Character Conversion 670
Period-to-DATE Conversion 673

Period-to-Period Conversion 675
Period-to-TIME Conversion 680
Period-to-TIMESTAMP Conversion 682
Signed Zone DECIMAL Conversion. 684
TIME-to-Character Conversion 688
TIME-to-Period Conversion 691
TIME-to-TIME Conversion. 693
TIME-to-TIMESTAMP Conversion 696
TIME-to-UDT Conversion 699
TIMESTAMP-to-Character Conversion 701
TIMESTAMP-to-DATE Conversion 705
TIMESTAMP-to-Period Conversion 708
TIMESTAMP-to-TIME Conversion 710
TIMESTAMP-to-TIMESTAMP Conversion 713
TIMESTAMP-to-UDT Conversion 716
UDT-to-Byte Conversion. 718
UDT-to-Character Conversion 721
UDT-to-DATE Conversion 725
UDT-to-INTERVAL Conversion 728
UDT-to-Numeric Conversion. 731
UDT-to-TIME Conversion 734
UDT-to-TIMESTAMP Conversion 737
UDT-to-UDT Conversion 740

Appendix A: Notation Conventions. 743

Syntax Diagram Conventions 743
Character Shorthand Notation Used In This Book 748
Predicate Calculus Notation Used In This Book 750

Glossary 751

Index. 753

CHAPTER 1 Introduction

This chapter provides a brief introduction and description of the SQL functions, operators, expressions, and predicates described in this book.

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 do return values. An example of a niladic SQL function is CURRENT_DATE.

Types of SQL Functions

There are four types of SQL functions:

- Scalar
- Aggregate
- Table
- Ordered Analytical Function

The following table defines these types.

Function Type	Definition
Scalar	The arguments are individual scalar values of either same or mixed type that can have different meanings. The result is a single value or null. 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 used in the expression list of a SELECT statement and in the summary list of a WITH clause.

Function Type	Definition
Table	The arguments are individual scalar values of either same or mixed type that can have different meanings. The result is a table. 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 *SQL External Routine Programming*.
Ordered Analytical Function	The arguments are any normal SQL expression. The result is handled the same as any other SQL expression. It can be a result column or part of a more complex arithmetic expression. Used in operations that require an ordered set of results rows or depend on values in a previous row. See “Ordered Analytical Functions” on page 298.

Examples of Functions

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

For examples of table functions, see *SQL External Routine Programming*.

SQL Operators

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

Types of Operators

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. (Comparison operators are a type of logical predicate. See also “Types of Logical Predicates” on page 20.)
- 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.

They allow you to perform arithmetic and logical operations, and to generate new values or Boolean results from constants and stored values.

An expression can consist of any of the following things:

- Column name
- Constant (also referred to as literal)
- Function
- USING variable
- parameter
- parameter marker (question mark (?) placeholder)
- Combination of column names, constants, 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, constant, function, or combination of column names, constants, and functions connected by arithmetic operators where the result is a numeric type.
String expression	
DateTime expression	
Interval expression	
Period expression	
Conditional expression	An expression that results in a value of TRUE, FALSE, or unknown (NULL). Conditional expressions are also referred to as logical predicates. See “SQL Predicates” on page 20.
CASE expressions	CASE expressions consist 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

The following are examples of expressions.

Expression	Description
`'Test Tech'`	Character string constant
`1024`	Numeric constant
`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
`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 conditional expressions, 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.

CHAPTER 2 CASE Expressions

This chapter describes SQL CASE expressions.

CASE

Purpose

Specifies alternate values for a conditional expression or expressions based on equality comparisons and conditions that evaluate to TRUE.

ANSI Compliance

CASE is ANSI SQL:2008 compliant.

Overview

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 Teradata Database.

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.

Two Forms of CASE Expressions

CASE expressions are specified in two different forms: Valued and Searched.

- Valued CASE is described under "Valued CASE Expression" on page 24.
- Searched CASE is described under "Searched CASE Expression" on page 27.

CASE Shorthands for Handling Nulls

Two shorthand forms of CASE are provided to handle nulls:

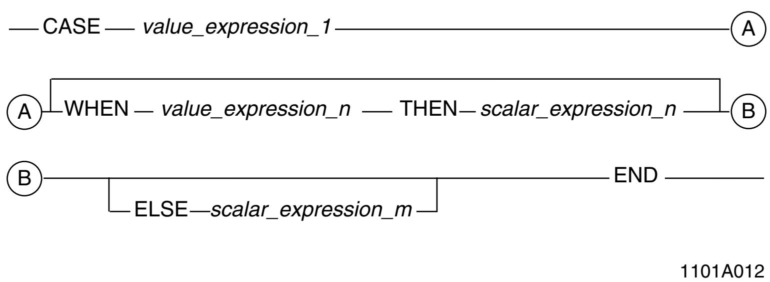
- COALESCE is described under "COALESCE Expression" on page 40.
- NULLIF is described under "NULLIF Expression" on page 42.

Valued CASE Expression

Purpose

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.

Syntax



where:

Syntax element ...	Specifies ...
value_expression_1	an expression whose value is tested for equality with *value_expression_n*.
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*.
scalar_expression_m	an expression whose value is returned if evaluation falls through to the ELSE clause.

ANSI Compliance

Valued CASE is ANSI SQL:2008 compliant.

Teradata Database does not enforce the ANSI restriction that *value_expression_1* must be a deterministic function. In particular, Teradata Database 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" on page 32.

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.

Default Title

The default title for a CASE expression appears as:

```
<CASE expression>
```

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 only when they are identical types because Teradata Database 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 *SQL Data Definition Language*.

Related Topics

For additional notes on ...	See ...
error conditions	"Error Conditions" on page 31.
the result data type of a CASE expression	"Rules for the CASE Expression Result Type" on page 32.
format of the result of a CASE expression	"Format for a CASE Expression" on page 37.
nulls and CASE expressions	"CASE and Nulls" on page 38.

Example 1

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 2

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 3

The following example shows how to combine a CASE expression with a concatenation operator:

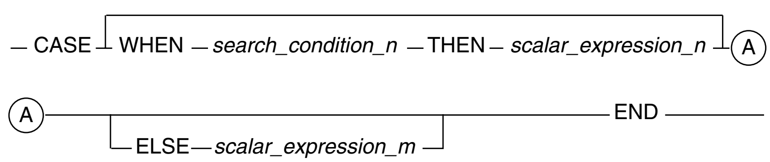
```
SELECT prodID, CASE prodSTATUS
                 WHEN 1
                 THEN 'SENT'
                 ELSE 'BACK ORDER'
                 END || ' STATUS'
FROM t1;
```

Searched CASE Expression

Purpose

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.

Syntax



FF07D224

where:

Syntax element ...	Specifies ...
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.
scalar_expression_m	a scalar expression whose value is returned when no *search_condition_n* evaluates to TRUE.

ANSI Compliance

Searched CASE is ANSI SQL:2008 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.

Default Title

The default title for a CASE expression appears as:

```
<CASE expression>
```

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 *search_condition_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in *search_condition_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 only when they are identical types because Teradata Database 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 *SQL Data Definition Language*.

Related Topics

For additional notes on …	See …
error conditions	“Error Conditions” on page 31.
the result data type of a CASE expression	“Rules for the CASE Expression Result Type” on page 32.

For additional notes on ...	See ...
format of the result of a CASE expression	"Format for a CASE Expression" on page 37.
nulls and CASE expressions	"CASE and Nulls" on page 38.

Example 1

The following statement is equivalent to the first example of the valued form of CASE on "Example 1" on page 26:

```
SELECT SUM(CASE
             WHEN part='1'
             THEN cost
             ELSE 0
            END
           ) / SUM(cost)
FROM t;
```

Example 2

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 3

The following example uses an ELSE clause.

```
SELECT *
FROM t
WHERE x = CASE
           WHEN y=2
           THEN 1
           ELSE 2
          END;
```

Example 4

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 5

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

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;`
An invalid WHEN expression is supplied in a valued CASE expression.	`SELECT CASE a WHEN a=1 THEN 1 ELSE 0 END FROM t;`

Condition or Expression	Example

An invalid 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" on page 512.

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" on page 47.

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 will be CHARACTER. If one or more expressions are VARCHAR (or LONG VARCHAR), the result data type will be VARCHAR.
- The server character set of the result is determined by scanning all the server character sets of the THEN/ELSE character expressions.

  If any THEN/ELSE character expression is a KANJI1 constant (for example, _Kanji1'<hex value>'XC), then all other THEN/ELSE character expressions must be of KANJI1 server character set. Otherwise, an error is returned.

  In all other cases, the server character set of the result is set to the server character set of the first THEN/ELSE character expression that is not a constant. The remaining THEN/ELSE character expressions must be translatable to this server character set.

  If all THEN/ELSE character expressions are constants, the server character set of the result is Unicode.

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_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
);
```

Example 1

The server character set of the result of the following query is UNICODE, because the server character set of the first THEN expression is UNICODE:

```
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;
```

Example 2

The result of the following query is a failure because one THEN/ELSE expression is a KANJI1 constant, 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;
```

Example 3

One THEN/ELSE expression in the following query has a KANJI1 constant. The query is successful and the result data type is KANJI1 because the server character set of all the other THEN/ELSE expressions are KANJI1.

```
SELECT i, CASE
                WHEN i=1 THEN column_k
                WHEN i=2 THEN ‘abc’
                WHEN i=3 THEN 8
                WHEN i=4 THEN _Kanji1’4142’XC
                ELSE 10
              END
FROM table_1
ORDER BY 1;
```

THEN/ELSE Expressions Having Mixed Data Types

The rules for mixed data appear in the following table:

<table>
<tr><th>IF the THEN/ELSE clause expressions …</th><th>THEN …</th></tr>
<tr><td>consist of BYTE and/or VARBYTE data types</td><td>if the data types of all of the THEN/ELSE expressions are BYTE, the result data type will be BYTE. If one or more expressions are VARBYTE, the result data type will be VARBYTE.</td></tr>
<tr><td>contain a DateTime or Interval data type</td><td>all of the THEN/ELSE clause expressions must have the same data type.</td></tr>
<tr><td>contain a FLOAT (approximate numeric) and no character strings</td><td>the CASE expression returns a FLOAT result.

Note: Some inaccuracy is inherent and unavoidable when FLOAT data types are involved.</td></tr>
<tr><td>are composed only of DECIMAL data</td><td rowspan="2">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” on page 47.</td></tr>
<tr><td>are composed only of mixed DECIMAL, BYTEINT, SMALLINT, INTEGER, and BIGINT data</td></tr>
</table>

IF the THEN/ELSE clause expressions …	THEN …
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. **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 1

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 * 6.1122334455667788800000
             ELSE column_3
            END )
FROM dec22;
```

Example 2

The following query corrects the problem in Example 1 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 3

In the following query, the CASE expression returns a DECIMAL result because its THEN and ELSE clauses contain both INTEGER and 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.

Example 1

In the following query, the CASE expression returns a VARCHAR result because its THEN and ELSE clause contains both 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.

```
SELECT a, CASE
             WHEN a=1
             THEN TIME
             ELSE USER
            END
FROM table_1
ORDER BY 1;
```

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;
```

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 *SQL Data Types and Literals*.

CASE and Nulls

The ANSI SQL:2008 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, Teradata Database 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, Teradata Database 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 constant.

CASE Shorthands

ANSI also defines two shorthand special cases of CASE specifically for handling nulls.

- COALESCE expression (see "COALESCE Expression" on page 40)
- NULLIF expression (see "NULLIF Expression" on page 42)

COALESCE Expression

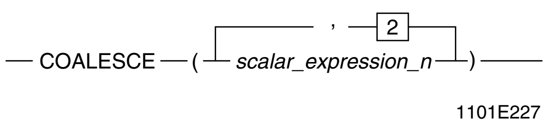
Purpose

COALESCE 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
```

Syntax



where:

Syntax element ...	Specifies ...
scalar_expression_n	an argument list. Each COALESCE function must have at least two operands.

ANSI Compliance

COALESCE is ANSI SQL:2008 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.

For additional information, such as the rules for evaluation and result data type, see "CASE" on page 23.

Default Title

The default title for a COALESCE expression appears as:

```
<CASE expression>
```

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 Teradata Database does not perform implicit type conversion on UDTs in a COALESCE expression.

Example 1

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 2

The following example uses COALESCE with an arithmetic operator.

```
SELECT COALESCE(Boxes,0) * 100
FROM Shipments;
```

Example 3

The following example uses COALESCE with a comparison operator.

```
SELECT Name
FROM Directory
WHERE Organization <> COALESCE (Level1, Level2, Level3);
```

NULLIF Expression

Purpose

NULLIF 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
```

Syntax

```
— NULLIF — ( — scalar_expression1, scalar_expression2 — ) —
```

HH01B094

where:

Syntax element ...	Specifies ...
scalar_expression_1	the scalar expression to the left of the = in the expanded CASE expression, as shown previously in "Purpose."
scalar_expression_2	the scalar expression to the right of the = in the expanded CASE expression, as shown previously in "Purpose."

ANSI Compliance

NULLIF is ANSI SQL:2008 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.

For additional information, such as the rules for evaluation and result data type, see "CASE" on page 23.

Default Title

The default title for a NULLIF expression appears as:

```
<CASE expression>
```

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 1

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 2

In the following query, blanks indicate no value.

```
SELECT FullName, NULLIF (Code, '    ') FROM Membership;
```

Example 3

The following example uses NULLIF in an expression with an arithmetic operator.

```
SELECT NULLIF(Age,0) * 100;
```

CHAPTER 3 Arithmetic Operators and Functions / Trigonometric and Hyperbolic Functions

This chapter describes the SQL arithmetic operators and functions/trigonometric and hyperbolic functions.

Arithmetic Operators

Teradata SQL supports the following arithmetic operators.

Operator	Function
**	Exponentiate This is a Teradata extension to the ANSI SQL:2008 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:2008 standard.
+	Add
-	Subtract
+	Unary plus (positive value)
-	Unary minus (negative value)

ANSI Compliance

Except for MOD and **, the arithmetic operators are ANSI SQL:2008 compliant.

Arithmetic Operators and LOBs

Arithmetic operators do not support BLOB or CLOB types.

Arithmetic Operators and DateTime and Interval Data Types

For details on the arithmetic operators permitted for DateTime and Interval data types, see "Arithmetic Operators" on page 168.

Arithmetic Operators and Period Data Types

For details on the arithmetic operators permitted for Period data types, see Chapter 7: "Period Functions and Operators."

Arithmetic Operators and UDTs

By default, Teradata Database 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. For more information on CREATE CAST, see *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see "Implicit Type Conversions" on page 577.

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 Operators and Result Types" on page 168.

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 type	any type	**	FLOAT
DATE	BYTEINT SMALLINT INTEGER BIGINT	+ -	DATE[1]
	BYTEINT SMALLINT INTEGER	* / MOD	INTEGER[4]

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 (*continued*)	BIGINT	* / MOD	BIGINT[4]
	DECIMAL(k,j)	+ -	DATE[2,4]
		* / MOD	DECIMAL(p,j)[4,6]
	FLOAT	* / + - MOD	FLOAT
	DATE	-	INTEGER[5]
		+ * / MOD	INTEGER[4]
	CHAR(n) VARCHAR(n)	, / + - MOD	FLOAT[3,4]
BYTEINT SMALLINT INTEGER	BYTEINT SMALLINT INTEGER	* / + - MOD	INTEGER
	BIGINT	* / + - MOD	BIGINT
	DECIMAL(k,j)	* / + - MOD	DECIMAL(p,j)[6]
	FLOAT	* / + - MOD	FLOAT
	CHAR(n) VARCHAR(n)	* / + - MOD	FLOAT[3]
	DATE	+	DATE[1]
		-	error
		* / MOD	INTEGER[4]
BIGINT	BYTEINT SMALLINT INTEGER BIGINT	* / + - MOD	BIGINT
	DECIMAL(k,j)	* / + - MOD	DECIMAL(p,j)[6]
	FLOAT	* / + - MOD	FLOAT
	CHAR(n) VARCHAR(n)	* / + - MOD	FLOAT[3]
	DATE	+	DATE[1]
		-	error
		* / MOD	BIGINT[4]

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)	BYTEINT SMALLINT INTEGER BIGINT	+ - *	DECIMAL(p,n)[6]
		/ MOD	DECIMAL(m,n)
	DECIMAL(k,j)	+ -	DECIMAL (min(p,(1+max(n,j)+max(m-n,k-j))), max(n,j))[7]
		*	DECIMAL(min(p,m+k),(n+j))[7]
		/ MOD	DECIMAL(p,max(n,j))[7]
	FLOAT	* / + - MOD	FLOAT
	CHAR(n) VARCHAR(n)	* / + - MOD	FLOAT[3]
	DATE	+	DATE[2]
		-	error
		*	DECIMAL(p,n)[4,6]
		/ MOD	DECIMAL(m,n)[4]
FLOAT	BYTEINT SMALLINT INTEGER BIGINT DECIMAL(k,j) FLOAT	* / + - MOD	FLOAT
	DATE	* / + - MOD	FLOAT[4]
	CHAR(n) VARCHAR(n)	* / + - MOD	FLOAT[3]
CHAR(n) VARCHAR(n)	BYTEINT SMALLINT INTEGER BIGINT DECIMAL(k,j) FLOAT CHAR(n) VARCHAR(n)	* / + - MOD	FLOAT[3]
	DATE	* / + - MOD	FLOAT[3,4]

1 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.

2 Fractions of decimal values are truncated when added to or subtracted from date values.

Note 1 also applies.

3 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.

4 These operations on DATE do not report an error, but results are generally not meaningful.

5 The difference between two dates is the number of days between those dates.

Note that this is *not* the numeric difference between the values.

6 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 the *Utilities* book.

- 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

7 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

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.

Decimal Results and Rounding

When computing an expression, decimal results that are not exact are rounded, not truncated.

For more information on rounding rules and how the RoundHalfwayMagUp field in DBSControl affects rounding, see “Decimal/Numeric Data Types” in *SQL Data Types and Literals* and “DBS Control utility” in *Utilities*.

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

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
- EXP
- LN
- LOG
- NULLIFZERO
- RANDOM
- RANGE_N
- SQRT
- ZEROIFNULL

ABS

Purpose

Computes the absolute value of an argument.

Syntax

```
—— ABS —( arg )——
```

1101A480

where:

Syntax element …	Specifies …
arg	a numeric argument.

ANSI Compliance

ABS is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

The following table lists the default attributes for the result of ABS(*arg*).

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>Same data type as arg[a]</td><td><table>
<tr><th>IF the operand is …</th><th>THEN the format is the default format for …</th></tr>
<tr><td>numeric</td><td>the resulting data type.</td></tr>
<tr><td>character</td><td>FLOAT.</td></tr>
<tr><td>a UDT</td><td>the predefined type to which the UDT is implicitly cast.</td></tr>
</table></td><td>ABS(arg)</td></tr>
</table>

a. Note that the NULL keyword has a data type of INTEGER.

For information on data type formats, see *SQL Data Types and Literals*.

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 “Implicit Type Conversions” on page 577.

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 *SQL Data Definition Language*.
- 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 *Utilities*.

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

Examples

Representative ABS arithmetic function expressions and the results are as follows.

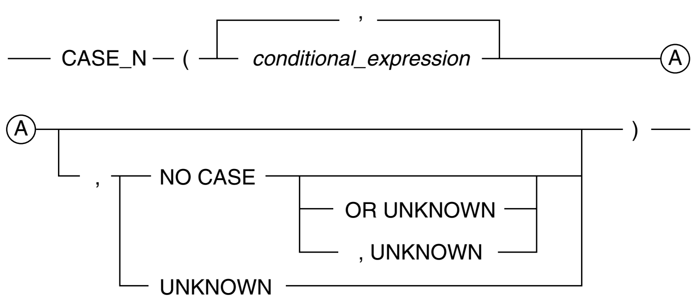
Expression	Result
ABS(-12)	12
ABS('23')	2.30000000000000E+001

CASE_N

Purpose

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.

Syntax



1101A069

where:

Syntax element ...	Specifies ...
conditional_expression	a conditional expression or comma-separated list of condition expressions to evaluate. A conditional expression must evaluate to TRUE, FALSE, or UNKNOWN.
NO CASE	an optional condition that evaluates to TRUE if every *conditional_expression* in the list evaluates to FALSE.
OR UNKNOWN	an optional condition to use with NO CASE. 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	an optional condition that evaluates to TRUE if a *conditional_expression* evaluates to UNKNOWN and all prior conditions in the list evaluate to FALSE.

ANSI Compliance

CASE_N is a Teradata extension to the ANSI SQL:2008 standard.

Evaluation

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.

<table>
<tr><th>IF …</th><th>THEN …</th></tr>
<tr><td>a conditional_expression evaluates to TRUE, and all prior conditions evaluate to FALSE</td><td>CASE_N returns the position of the conditional_expression.</td></tr>
<tr><td>a conditional_expression evaluates to UNKNOWN, and all prior conditions evaluate to FALSE</td><td>
<table>
<tr><th>IF …</th><th>THEN CASE_N returns …</th></tr>
<tr><td>NO CASE OR UNKNOWN is specified</td><td>n + 1.</td></tr>
<tr><td>UNKOWN is specified and NO CASE is not specified</td><td>n + 1.</td></tr>
<tr><td>NO CASE and UNKNOWN are specified</td><td>n + 2.</td></tr>
<tr><td>neither UNKNOWN nor NO CASE OR UNKNOWN is specified</td><td>NULL.</td></tr>
</table>
</td></tr>
<tr><td>every conditional_expression evaluates to FALSE</td><td>
<table>
<tr><th>IF …</th><th>THEN CASE_N returns …</th></tr>
<tr><td>NO CASE or NO CASE OR UNKNOWN is specified</td><td>n + 1.</td></tr>
<tr><td>neither NO CASE nor NO CASE OR UNKNOWN is specified</td><td>NULL.</td></tr>
</table>
</td></tr>
</table>

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 *SQL Data Types and Literals*.

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" on page 75.

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.

Restrictions

If CASE_N is used in a PARTITION BY phrase, it:

- Must not involve character or graphic comparisons
- 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

If CASE_N is used in a partitioning expression for a multilevel PPI, it must define at least two partitions.

Example 1

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 2

Here is an example that modifies "Example 1" 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'.

Level 1 Partition Number	Level 2 Partition Number	Condition
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 3

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
```

Related Topics

For information on ...	See ...
PPI properties and performance considerations	*Database Design*.
PPI considerations and capacity planning	
the specification of a PPI for a table	CREATE TABLE in *SQL Data Definition Language*.
the specification of a PPI for a join index	CREATE JOIN INDEX in *SQL Data Definition Language*.
the modification of the partitioning of the primary index for a table	ALTER TABLE in *SQL Data Definition Language*.

EXP

Purpose

Raises *e* (the base of natural logarithms) to the power of the argument, where $e = 2.71828182845905$.

Syntax

```
—— EXP —( arg )——
```

1101A484

where:

Syntax element …	Specifies …
arg	a numeric argument.

ANSI Compliance

EXP is a Teradata extension to the ANSI SQL:2008 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*)

For information on default data type formats, see *SQL Data Types and Literals*.

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 "Implicit Type Conversions" on page 577.

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 on CREATE CAST, see *SQL Data Definition Language*.

- 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. For details, see *Utilities*.

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

Usage Notes

Executing EXP may sometimes result in a numeric overflow error.

Examples

Representative EXP arithmetic function expressions and the results are as follows.

Expression	Result
EXP(1)	2.71828182845905E+000
EXP(0)	1.00000000000000E+000

LN

Purpose

Computes the natural logarithm of the argument.

Syntax

```
—— LN —( arg )——
```

1101A485

where:

Syntax element ...	Specifies ...
arg	a nonzero, positive numeric argument.

ANSI Compliance

LN is a Teradata extension to the ANSI SQL:2008 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)

For information on default data type formats, see *SQL Data Types and Literals*.

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 "Implicit Type Conversions" on page 577.

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 on CREATE CAST, see *SQL Data Definition Language*.

- 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. For details, see *Utilities*.

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

Examples

Representative LN arithmetic function expressions and the results are as follows.

Expression	Result
LN(2.71828182845905)	1.00000000000000E+000
LN(0)	Error

LOG

Purpose

Computes the base 10 logarithm of an argument.

Syntax

```
—— LOG —( arg )——
```

1101A486

where:

Syntax element ...	Specifies ...
arg	a nonzero, positive numeric argument.

ANSI Compliance

LOG is a Teradata extension to the ANSI SQL:2008 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 *SQL Data Types and Literals*.

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 "Implicit Type Conversions" on page 577.

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 on CREATE CAST, see *SQL Data Definition Language*.

- 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. For details, see *Utilities*.

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

Examples

Representative LOG arithmetic function expressions and the results are as follows.

Expression	Result
LOG(50)	1.69897000433602E+000
LOG(100)	2.00000000000000E+000

NULLIFZERO

Purpose

Converts data from zero to null to avoid problems with division by zero.

Syntax

```
— NULLIFZERO — ( arg ) —
```

1101F225

where:

Syntax element ...	Specifies ...
arg	a numeric argument, or an argument that can be converted to a numeric argument based on implicit type conversion rules.

ANSI Compliance

NULLIFZERO is a Teradata extension to the ANSI SQL:2008 standard.

The ANSI form of this function is the CASE shorthand expression NULLIF. For more information, see "NULLIF Expression" on page 42.

Result Type and Attributes

Here are the default attributes for the result of NULLIFZERO(*arg*).

<table>
<tr><th>Data Type and Format</th><th>Title</th></tr>
<tr><td>
<table>
<tr><th>IF arg is ...</th><th>THEN the data type is ...</th><th>AND the format is the ...</th></tr>
<tr><td>numeric</td><td>the same type as arg[a]</td><td>same format as arg.</td></tr>
<tr><td>character</td><td>FLOAT</td><td>default format for FLOAT.</td></tr>
<tr><td>a UDT</td><td>the type to which the UDT is implicitly cast</td><td>the format of the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>NULLIFZERO(arg)</td></tr>
</table>

a. Note that the NULL keyword has a data type of INTEGER.

For information on data type formats, see *SQL Data Types and Literals*.

Result Value

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, 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 "Implicit Type Conversions" on page 577.

If *arg* is a character string, it 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 on CREATE CAST, see *SQL Data Definition Language*.
- 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. For details, see *Utilities*.

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

Example 1

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 2

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 Topics

For additional expressions involving checks for nulls, see:

- "COALESCE Expression" on page 40
- "NULLIF Expression" on page 42
- "ZEROIFNULL" on page 88

RANDOM

Purpose

Returns a random integer number for each row of the results table.

Syntax

```
── RANDOM ─ ( lower_bound, upper_bound ) ─
```

1101C025

where:

Syntax element ...	Specifies ...
lower_bound	an integer constant 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, inclusive. *lower_bound* must be less than or equal to *upper_bound*.
upper_bound	an integer constant 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, inclusive. *upper_bound* must be greater than or equal to *lower_bound*.

ANSI Compliance

RANDOM is a Teradata extension to the ANSI SQL:2008 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 *SQL Data Types and Literals*.

Computation

RANDOM uses the linear congruential algorithm and 48-bit integer arithmetic.

The algorithm works by generating a sequence of 48-bit integer values, X_i, using the following equation:

$$X_{n+1} = (aX_n + c) \% m$$

where:

This variable ...	Represents ...
X	a random number over a defined closed interval
n	an integer >= 0
a	0x5DEECE66D
c	0xB
%	the modulo operator
m	2^{48}

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.

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.

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

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

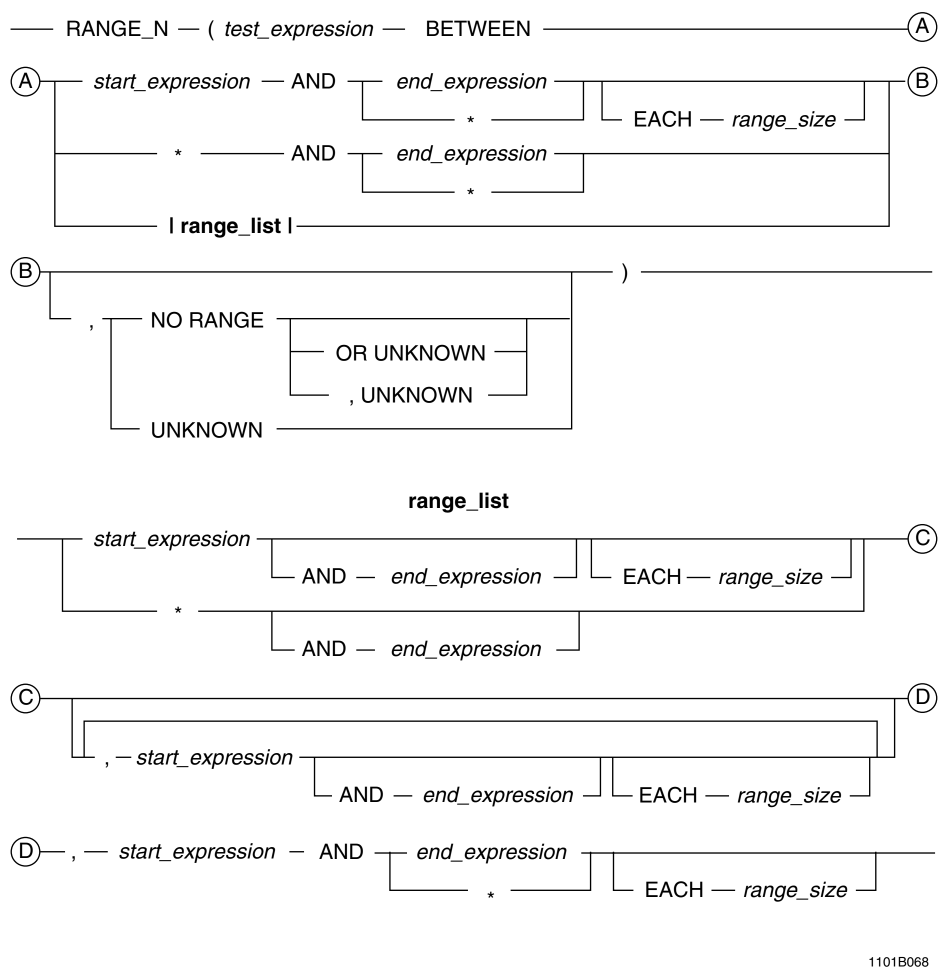
Store_ID	Product_ID	Sales	RANDOM(1,3)
1002	D	50000	3
1003	D	50000	2
1001	D	35000	3
1001	A	100000	2
1002	A	40000	1
1001	E	30000	2

RANGE_N

Purpose

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.

Syntax



where:

Syntax element ...	Specifies ...
test_expression	an expression that results in a BYTEINT, SMALLINT, INTEGER, or DATE data type.

Syntax element ...	Specifies ...
start_expression *	a constant or constant 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 such that it can 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, inclusively, 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 constant or constant 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 such that it can 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, inclusively, 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).
EACH *range_size*	a constant or constant 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. For more information on ADD_MONTHS, see "ADD_MONTHS" on page 174. The data type of *range_size* must be compatible for adding to *test_expression*.
NO RANGE	an optional range to handle a *test_expression* that does not map into any of the specified ranges.
OR UNKNOWN	an option to use with NO RANGE. The NO RANGE OR UNKNOWN option 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	an option to handle a *test_expression* that evaluates to NULL when RANGE_N does not specify the range BETWEEN * AND *.

ANSI Compliance

RANGE_N is a Teradata extension to the ANSI SQL:2008 standard.

Range Definition

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.

Evaluation

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.

<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>the result of test_expression is within a range</td><td>RANGE_N returns the position of the range.</td></tr>
<tr><td>the result of test_expression is NULL</td><td>
<table>
<tr><th>IF RANGE_N ...</th><th>THEN ...</th></tr>
<tr><td>does not specify one of the following:
• BETWEEN * AND *
• UNKNOWN
• NO RANGE OR UNKNOWN</td><td>RANGE_N returns NULL.</td></tr>
<tr><td>specifies the range BETWEEN * AND *</td><td>RANGE_N returns 1, regardless of whether NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN is specified.</td></tr>
<tr><td>does not specify the range BETWEEN * AND *</td><td>
<table>
<tr><th>IF ...</th><th>THEN RANGE_N returns ...</th></tr>
<tr><td>NO RANGE OR UNKNOWN is specified</td><td>n + 1.</td></tr>
<tr><td>UNKNOWN is specified and NO RANGE is not specified</td><td>n + 1.</td></tr>
<tr><td>NO RANGE and UNKNOWN are specified</td><td>n + 2.</td></tr>
</table>
</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>test_expression is outside all the ranges in the list</td><td>
<table>
<tr><th>IF ...</th><th>THEN RANGE_N returns ...</th></tr>
<tr><td>NO RANGE or NO RANGE OR UNKNOWN is specified</td><td>n + 1.</td></tr>
<tr><td>neither NO RANGE nor NO RANGE OR UNKNOWN is specified</td><td>NULL.</td></tr>
</table>
</td></tr>
</table>

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 *SQL Data Types and Literals*.

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" on page 56.

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.

Restrictions

If RANGE_N appears in a PARTITION BY phrase, it:

- Must not use character or graphic comparisons.
- Can specify a maximum of 65533 ranges (unless it is part of a larger partitioning expression)
- Must not contain the system-derived columns PARTITION or PARTITION#L1 through PARTITION#L15

If RANGE_N is used in a partitioning expression for a multilevel PPI, it must define at least two partitions.

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, or DATE instead of depending upon any implicit conversions.

Example 1

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

Here is an example that modifies "Example 1" 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'.
	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

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

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
```

Related Topics

For information on ...	See ...
PPI properties and performance considerations	*Database Design*.
PPI considerations and capacity planning	
specifying a PPI for a table	CREATE TABLE in *SQL Data Definition Language*.
specifying a PPI for a join index	CREATE JOIN INDEX in *SQL Data Definition Language*.
modifying the partitioning of the primary index for a table	ALTER TABLE in *SQL Data Definition Language*.

SQRT

Purpose

Computes the square root of an argument.

Syntax

```
—— SQRT — ( arg ) ——
```

1101A487

where:

Syntax element ...	Specifies ...
arg	a positive, numeric argument.

ANSI Compliance

SQRT is a Teradata extension to the ANSI SQL:2008 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*)

For information on default data type formats, see *SQL Data Types and Literals*.

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 "Implicit Type Conversions" on page 577.

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 on CREATE CAST, see *SQL Data Definition Language*.

- 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. For details, see *Utilities*.

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

Examples

Representative SQRT arithmetic function expressions and the results are as follows.

Expression	Result
SQRT(2)	1.41421356237309E+000
SQRT(-2)	Error

WIDTH_BUCKET

Purpose

Returns the number of the partition to which *value_expression* is assigned.

Syntax

```
— WIDTH BUCKET — ( value_expression, lower_bound, upper_bound, partition_count ) —
```

1101A492

where:

Syntax element ...	Specifies the ...
value_expression	value for which a partition number is to be returned.
lower_bound	lower boundary for the range of values to be partitioned equally.
upper_bound	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.

ANSI Compliance

WIDTH_BUCKET is ANSI SQL:2008 compliant.

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)

For information on default data type formats, see *SQL Data Types and Literals*.

Argument Types and Rules

Use the following table for rules concerning WIDTH_BUCKET arguments.

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 them 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 following rules apply to UDT arguments:
	• 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 on CREATE CAST, see *SQL Data Definition Language*.
	• 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 *Utilities*.

If an argument cannot be implicitly converted to an acceptable type, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 577.

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 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

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 1
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
```

ZEROIFNULL

Purpose

Converts data from null to 0 to avoid cases where a null result creates an error.

Syntax

```
—ZEROIFNULL —( arg )—
                     1101F226
```

where:

Syntax element ...	Specifies ...
arg	a numeric argument.

ANSI Compliance

ZEROIFNULL is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

Here are the default attributes for the result of ZEROIFNULL(*arg*).

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>Same data type as arg[a]</td><td><table>
<tr><th>IF the operand is ...</th><th>THEN the format is the ...</th></tr>
<tr><td>numeric</td><td>same format as arg.</td></tr>
<tr><td>character</td><td>default format for FLOAT.</td></tr>
<tr><td>UDT</td><td>format of the predefined type to which the UDT is implicitly cast.</td></tr>
</table></td><td>ZEROIFNULL(arg)</td></tr>
</table>

a. Note that the NULL keyword has a data type of INTEGER.

For information on data type formats, see *SQL Data Types and Literals*.

Argument Types and Rules

IF the value of arg is ...	THEN ZEROIFNULL returns ...
not null	the value of the numeric argument.
null or zero[a]	zero.

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

If the argument is not numeric, it 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 on implicit type conversion, see "Implicit Type Conversions" on page 577.

If *arg* is a character string, it 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 on CREATE CAST, see *SQL Data Definition Language*.
- 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 *Utilities*.

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

Example

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 Topics

For additional expressions involving checks for nulls, see:

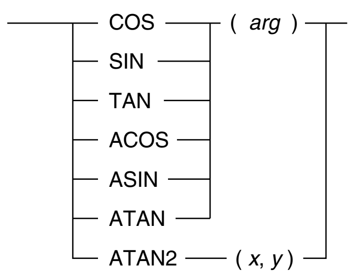
- "COALESCE Expression" on page 40
- "NULLIF Expression" on page 42
- "NULLIFZERO" on page 68

Trigonometric Functions (COS, SIN, TAN, ACOS, ASIN, ATAN, ATAN2)

Purpose

Performs the trigonometric or inverse trigonometric function of an argument.

Syntax



1101A482

where:

Syntax element ...	Specifies ...
arg	any valid numeric expression that expresses an angle in radians.
x	the x-coordinate of a point to use in the arctangent calculation.
y	the y-coordinate of a point to use in the arctangent calculation.

ANSI Compliance

Trigonometric and inverse trigonometric functions are Teradata extensions to the ANSI SQL:2008 standard.

Definitions

Function	Definition
Arccosine	The arccosine is the angle whose cosine is the argument.
Arcsine	The arcsine is the angle whose sine is the argument.
Arctangent	The arctangent is the angle whose tangent is the argument.
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.

Function	Definition
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.
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.

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 *SQL Data Types and Literals*.

Result Value

Function	Result Value
COS(*arg*)	The cosine of *arg* in radians in the range -1 to 1, inclusive.
SIN(*arg*)	The sine of *arg* in radians in the range -1 to 1, inclusive.
TAN(*arg*)	The tangent of *arg* in radians.
ACOS(*arg*)	An angle in the range 0 to π radians, inclusive.
ASIN(*arg*)	An angle in the range -π/2 to π/2 radians, inclusive.
ATAN(*arg*)	An angle in the range -π/2 to π/2 radians, inclusive.
ATAN2(*x*,*y*)	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.

Argument Types and Rules

Arguments that are not FLOAT are converted to FLOAT based on implicit type conversion rules. If an argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 577.

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. For more information on CREATE CAST, see *SQL Data Definition Language*.
- 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 *Utilities*.

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

Examples

The following are representative function expressions and results.

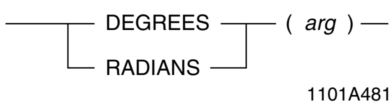
Expression	Result
COS(5-4)	5.40302305868140E -001
SIN(LOG(0.5))	-2.96504042171437E -001
SIN(RADIANS(180.0))	1.22464679914735E-016
TAN(ABS(-3))	-1.42546543074278E -001
ACOS(-0.5)	2.09439510239320E 000
ASIN(1)	1.57079632679490E 000
ATAN(1+2)	1.24904577239825E 000
ATAN2(1,1)	7.85398163397448E -001

DEGREES RADIANS

Purpose

DEGREES takes a value specified in radians and converts it to degrees. RADIANS takes a value specified in degrees and converts it to radians.

Syntax



where:

<table>
<tr><th>Syntax element ...</th><th>Specifies ...</th></tr>
<tr><td>arg</td><td>a numeric expression.
<table>
<tr><th>IF the function is ...</th><th>THEN arg is interpreted as an angle in ...</th></tr>
<tr><td>DEGREES</td><td>radians.</td></tr>
<tr><td>RADIANS</td><td>degrees.</td></tr>
</table>
</td></tr>
</table>

ANSI Compliance

DEGREES and RADIANS are Teradata extensions to the ANSI SQL:2008 standard.

Result Title

The following table lists the default titles for DEGREES(*arg*) and RADIANS(*arg*).

Function	Title
DEGREES(*arg*)	(5.72957795130823E001**arg*)
RADIANS(*arg*)	(1.74532925199433E-002**arg*)

Result Type and Format

The following table lists the result type and format of DEGREES(*arg*) and RADIANS(*arg*).

<table>
<tr><th>Data Type</th><th>Format</th></tr>
<tr><td>Same data type as arg[a]</td><td>
<table>
<tr><th>IF the operand is …</th><th>THEN the format is the default format for …</th></tr>
<tr><td>numeric</td><td>the resulting data type.</td></tr>
<tr><td>character</td><td>FLOAT.</td></tr>
<tr><td>a UDT</td><td>the predefined type to which the UDT is implicitly cast.</td></tr>
</table>
</td></tr>
</table>

a. Note that the NULL keyword has a data type of INTEGER.

For information on data type formats, see *SQL Data Types and Literals*.

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 "Implicit Type Conversions" on page 577.

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 *SQL Data Definition Language*.
- 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 *Utilities*.

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

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 and RADIANS function expressions and the results are as follows.

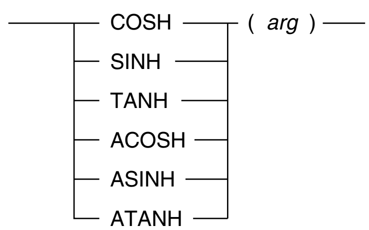
Expression	Result
SIN(RADIANS(60.0))	8.66025403784439E-001
DEGREES(1.0)	5.72957795130823E 001

Hyperbolic Functions (COSH, SINH, TANH, ACOSH, ASINH, ATANH)

Purpose

Performs the hyperbolic or inverse hyperbolic function of an argument.

Syntax



1101A483

where:

Syntax element …	Specifies …
arg	any real number.

ANSI Compliance

Hyperbolic and inverse hyperbolic functions are Teradata extensions to the ANSI SQL:2008 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 *SQL Data Types and Literals*.

Result Value

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 on implicit type conversion, see "Implicit Type Conversions" on page 577.

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 on CREATE CAST, see *SQL Data Definition Language*.
- 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 *Utilities*.

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

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

CHAPTER 4 Comparison Operators

This chapter describes SQL comparison operators.

Comparison Operators

Purpose

Comparison operators test the truth of relations between expressions.

Comparison operators are a type of logical predicate and can appear in conditional expressions in:

- 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

Syntax

```
—— scalar_expression —— comparison_operator —— scalar_expression ——
```

FF07D160

where:

Syntax element ...	Specifies ...
scalar_expression	an expression to be evaluated in comparison with a second *scalar_expression*. 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.
comparison_operator	the type of comparison to be evaluated for truth. For a list of the supported comparison operators, see "Supported Comparison Operators" on page 102.

ANSI Compliance

The following comparison operators are ANSI SQL:2008 compliant.

- =
- >
- <
- <>
- <=
- >=

The following comparison operators are Teradata extensions to the ANSI SQL:2008 standard. Their use is deprecated.

- EQ
- ^=
- NE
- NOT=
- LT
- LE
- GT
- GE

Supported Comparison Operators

Teradata Database 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.

Further Information on Predicates

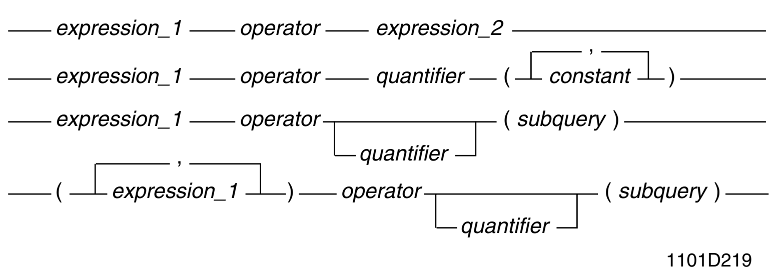
FOR more information on …	SEE …
using predicates in conditional expressions in searched CASE expressions	Chapter 2: "CASE Expressions."
using predicates in conditional expressions in WHERE, ON, or HAVING clauses in SELECT statements	"The SELECT Statement" in *SQL Data Manipulation Language*.
using predicates in conditional expressions in IF, WHILE, or REPEAT statements in stored procedures	*SQL Stored Procedures and Embedded SQL*.

FOR more information on …	SEE …
other logical predicates, including: • [NOT] EXISTS • [NOT] IN • LIKE • IS [NOT] NULL • OVERLAPS • [NOT] BETWEEN … AND …	Chapter 11: "Logical Predicates."
predicate quantifiers: • ALL • ANY • SOME	

Comparison Operators in Logical Expressions

Syntax

A logical expression using comparison operators has the following valid forms.



where:

Syntax Element …	Specifies …
operator	one of the comparison operators.
expression_1 *expression_2*	an SQL scalar expression.
quantifier	one of the following quantifier keywords: • ANY • SOME • ALL For information, see "ANY/ALL/SOME Quantifiers" on page 445.

Syntax Element ...	Specifies ...
constant	one or more constant values. A constant may be 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 constants. The data types of *expression* and *constant* must be compatible. If the data types of the operands differ, Teradata Database performs an implicit conversion from one type to another in some cases. For details, see "Implicit Type Conversion of Comparison Operands" on page 108.
subquery	an SQL SELECT statement. Using a subquery in a condition is restricted in certain cases.

Results

A logical expression that uses a comparison operator evaluates to TRUE, FALSE, or UNKNOWN.

Using Subqueries in Comparison Operations

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.

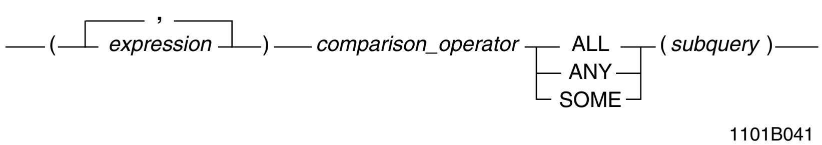
The subquery must refer to at least one table. A table that is in the WHERE clause, but that is not referred to in any other parts of the subquery, is not applicable.

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

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) ;
```

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 δ^{th} item in the range is notated as row δ such that the δ^{th} component of *expression_1* is notated as $expression\_1_\delta$ and the δ^{th} component of *expression_2* is notated as $expression\_2_\delta$.

The data types of *expression_1* and *expression_2* must be compatible. If the data types of the expressions differ, Teradata Database performs an implicit conversion from one type to another in some cases. For details, see "Implicit Type Conversion of Comparison Operands" on page 108.

For an explanation of the symbols used in this table, see "Predicate Calculus Notation Used In This Book" on page 750.

This comparison ...	Is TRUE iff ...
expression_1 = *expression_2*	$\forall r$, $expression\_1_r = expression\_2_r$ is TRUE.
expression_1 <> *expression_2*	$\exists\ \delta$ such that $expression\_1\delta$ <> $expression\_2\delta$ is TRUE.
expression_1 < *expression_2*	$\exists\ \delta$ such that $expression\_1\delta < expression\_2\delta$ is TRUE and for all $r < \delta$, $expression\_1_r = expression\_2_r$ is TRUE.
expression_1 > *expression_2*	$\exists\ \delta$ such that $expression\_1\delta > expression\_2\delta$ is TRUE and for all $r > \delta$, $expression\_1_r = expression\_2_r$ is TRUE.

This comparison ...	Is TRUE iff ...
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;
```

For more information on potential problems associated with floating point values in comparison operations, see *SQL Data Types and Literals*.

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.

<table>
<tr><th>This data type ...</th><th>Is evaluated in this way ...</th></tr>
<tr><td>Byte</td><td>Bit-by-bit from left to right. A 0 bit is less than a 1 bit.
<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>every pairwise comparison is equal</td><td>the two byte strings are equal.</td></tr>
<tr><td>any pairwise comparison is not equal</td><td>that comparison determines the result.</td></tr>
<tr><td>two byte strings of different lengths are compared</td><td>the shorter string is padded to the right with binary zeros to make the lengths equal prior to making the comparison.</td></tr>
</table></td></tr>
<tr><td>Character</td><td>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
<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>every pairwise comparison is equal</td><td>the two character strings are equal.</td></tr>
<tr><td>any pairwise comparison is not equal</td><td>that comparison determines the result.</td></tr>
</table>
For more information on character comparison, see “Character String Comparisons” on page 112.</td></tr>
<tr><td>DateTime</td><td>Chronologically.
For information on how Time Zone affects Time comparison, see “Time Zone Sort Order” on page 159.</td></tr>
<tr><td>Interval</td><td>According to sign and magnitude.</td></tr>
<tr><td>Period</td><td>Assuming p1 and p2 are Period value expressions, the evaluation of a Period comparison predicate uses the following logic:
<code>IF BEGIN(p1) = BEGIN(p2) is TRUE, return END(p1) operator END(p2)</code>
<code>ELSE return (BEGIN(p1) operator BEGIN(p2))</code>
For details on BEGIN and END, see Chapter 7: “Period Functions and Operators.”</td></tr>
<tr><td>UDT</td><td>According to the ordering definition of the UDT.
Teradata Database 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 SQL Data Definition Language.</td></tr>
</table>

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, then Teradata Database performs an implicit conversion according to the following table. Implicit conversions are Teradata extensions to the ANSI SQL:2008 standard.

<table>
<tr><th>IF one expression operand is ...</th><th>AND the other expression operand is ...</th><th>THEN Teradata Database compares the data as ...</th></tr>
<tr><td>Character</td><td>Character</td><td>Character.
For more details, see “Character String Comparisons” on page 112.</td></tr>
<tr><td rowspan="3">Character</td><td>Date</td><td>Date[a].</td></tr>
<tr><td>BYTEINT
SMALLINT
INTEGER
FLOAT</td><td>FLOAT[a,b].</td></tr>
<tr><td>Period</td><td>Period. For more details, see “Comparison of Period Types” on page 118.</td></tr>
<tr><td rowspan="2">CHAR(k)
VARCHAR(k)
where k <= 16</td><td>BIGINT</td><td rowspan="3">FLOAT[a,b].
Note: Teradata Database returns an error if a comparison involves either of the following combination of operand types:
• BIGINT and CHAR(k) or VARCHAR(k) where k > 16.
• DECIMAL(m,n) where m > 16 and CHAR(k) or VARCHAR(k) where k > 16.</td></tr>
<tr><td>DECIMAL(m,n)</td></tr>
<tr><td>CHAR(k)
VARCHAR(k)
where k > 16</td><td>DECIMAL(m,n)
where m <= 16</td></tr>
<tr><td>BYTEINT</td><td>SMALLINT</td><td>SMALLINT.</td></tr>
<tr><td>BYTEINT
SMALLINT</td><td>INTEGER</td><td>INTEGER.</td></tr>
<tr><td>BYTEINT
SMALLINT
INTEGER
BIGINT</td><td>BIGINT</td><td>BIGINT.</td></tr>
</table>

IF one expression operand is …	AND the other expression operand is …	THEN Teradata Database compares the data as …
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 DATE	DECIMAL(*m*,*n*) where *m* <= 18 and *m*-*n* >= 10	
BYTEINT	DECIMAL(*m*,*n*) where *m* > 18 or *m*-*n* < 3	DECIMAL(38,*n*).
SMALLINT	DECIMAL(*m*,*n*) where *m* > 18 or *m*-*n* < 5	
INTEGER DATE	DECIMAL(*m*,*n*) where *m* > 18 or *m*-*n* < 10	
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*)).
DATE	BYTEINT SMALLINT INTEGER	INTEGER.
	BIGINT	BIGINT.
	FLOAT	FLOAT.
FLOAT	BYTEINT SMALLINT INTEGER BIGINT DECIMAL(*m*,*n*)	FLOAT.
Period	Character	Period. For more details, see "Comparison of Period Types" on page 118.

a. Returns an error for character data with GRAPHIC server character set.

b. Comparisons between character and numeric data types require that the character field be convertible to a numeric value.

Implicit Conversion of DateTime Types

In comparisons involving DateTime operands that differ, Teradata Database performs an implicit conversion according to the following table.

IF one expression operand is ...	AND the other expression operand is ...	THEN Teradata Database compares the data as ...
TIMESTAMP	DATE[b]	DATE. See "Implicit TIMESTAMP-to-DATE Conversion" on page 706.
TIMESTAMP WITH TIME ZONE		
Interval[a]	Exact Numeric	Numeric. See "Implicit INTERVAL-to-Numeric Conversion" on page 651.

a. The INTERVAL type must have only one field, e.g. INTERVAL YEAR.

b. ANSIDate dateform mode or IntegerDate dateform mode

Data Types on Which Implicit Conversion is Not Performed

The following table identifies data types on which Teradata Database 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.
TIME	Teradata Database does not perform implicit type conversion from TIME to TIMESTAMP and from TIMESTAMP to TIME in comparison operations.
TIMESTAMP	
UDT	Teradata Database 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 more information, see Chapter 17: "Data Type Conversions."

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" on page 171.

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 1

In the first example, the operand is entered as an integer.

```
SELECT *
FROM Employee
WHERE DOB = 420327 ;
```

Example 2

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 3

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 4

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 Translation" on page 595.

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.

For collation details, see:

- "SET SESSION COLLATION" in *SQL Data Definition Language*
- *International Character Set Support*
- "ORDER BY Clause" in *SQL Data Manipulation Language*

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:2008 compatible attribute—ANSI does *all* character comparisons as the equivalent of CASESPECIFIC.

The CASESPECIFIC attribute has higher precedence over the NOT CASESPECIFC 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.

For a description of the UPPER function, see "UPPER" on page 424.

Example

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.

<table>
<tr><th>IF column Firstname is ...</th><th>THEN ...</th></tr>
<tr><td>CASESPECIFIC</td><td>
<table>
<tr><th>IF the session mode is ...</th><th>THEN 'George' is ...</th><th>AND the match succeeds for rows with Firstname containing ...</th></tr>
<tr><td>ANSI</td><td>CASESPECIFIC</td><td rowspan="2">'George'
When either character string is CASESPECIFIC, the comparison is case specific.</td></tr>
<tr><td>Teradata</td><td>NOT CASESPECIFIC</td></tr>
</table>
</td></tr>
<tr><td>NOT CASESPECIFIC</td><td>
<table>
<tr><th>IF the session mode is ...</th><th>THEN 'George' is ...</th><th>AND the match succeeds for rows with Firstname containing ...</th></tr>
<tr><td>ANSI</td><td>CASESPECIFIC</td><td>'George'
When either character string is CASESPECIFIC, the comparison is case specific.</td></tr>
<tr><td>Teradata</td><td>NOT CASESPECIFIC</td><td>any combination of cases that spell the name George, such as:
• 'george'
• 'GEORGE'
• 'George'
When both character strings are NOT CASESPECIFIC, the comparison is case blind.</td></tr>
</table>
</td></tr>
</table>

Comparison of KANJI1 Characters

The following sections describe how Teradata Database compares KANJI1 characters.

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" on page 595.
- 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.

For further details on collation sequences, see *International Character Set Support.*

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, Teradata Database 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.

For more information on the DEFAULT function, see “DEFAULT” on page 503.

Comparison of Period Types

Two Period values are comparable if their element types are of same DateTime data type. The DateTime data types are DATE, TIME and TIMESTAMP. The PERIOD(DATE) date type is comparable with the PERIOD(DATE) data type, a PERIOD(TIME(n)[WITH TIME ZONE]) data type is comparable with a PERIOD(TIME(m)[WITH TIME ZONE]) data type, and a PERIOD(TIMESTAMP(n)[WITH TIME ZONE]) data type is comparable with a PERIOD(TIMESTAMP(m)[WITH TIME ZONE]) data type.

Teradata extends this to allow a CHARACTER and VARCHAR value to be implicitly cast as a Period data type for some operators and, therefore, have a Period data type. Since the Period data type is the data type of the other Period value expression, these Period value expressions will be comparable.

DateTime and Period data are saved internally with the maximum precision of 6 although the specified precision may be less than this and is padded with zeroes. Thus, the comparison operations with differing precisions work without any additional logic. Additionally, the internal value is saved in UTC for a Time or Timestamp value, or for a Period value with an element type of TIME or TIMESTAMP. All comparable Period value expressions can be compared directly due to this internal representation irrespective of whether they contain a time zone value, or whether they have the same precision.

Note: The time zone values are ignored when comparing values.

The following table describes the comparison operators.

Operator	Purpose
EQ or =	Assume p1 and p2 are Period value expressions and have comparable Period data types. If BEGIN(p1) = BEGIN(p2) AND END(p1) = END(p2), the result of the comparison is TRUE; otherwise, the result is FALSE. If either Period value expression is NULL, the result is UNKNOWN. If the Period value expressions have different element types, one of them must be explicitly CAST as the other. If one Period value expression has a Period data type and the other Period value expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR Period value expression is implicitly converted, before comparison, to the data type of the Period value expression based on the format of the Period value expression.

Operator	Purpose
LT or <	Assume p1 and p2 are Period value expressions and have comparable Period data types. If BEGIN(p1) < BEGIN(p2) OR (BEGIN(p1) = BEGIN(p2) AND END(p1) < END(p2)), the result of the comparison is TRUE; otherwise, the result is FALSE. If either Period value expression is NULL, the result is UNKNOWN. If the Period value expressions have different element types, one of them must be explicitly CAST as the other. If one Period value expression has a Period data type and the other Period value expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR operand is implicitly converted, before comparison, to the data type of the Period value expression based on the format of the Period value expression.
GT or >	Assume p1 and p2 are Period value expressions and have comparable Period data types. If BEGIN(p1) > BEGIN(p2) OR (BEGIN(p1) = BEGIN(p2) AND END(p1) > END(p2)), the result of the comparison is TRUE; otherwise, it is FALSE. If either Period expression is NULL, the result is UNKNOWN. If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR Period value expression is implicitly converted, before comparison, to the data type of the Period value expression based on the format of the Period value expression.
NE or <> or NOT= or ^= or LE or <= or GE or >=	These comparison operators are supported for comparable Period value expressions. Also, if one Period value expression has a Period data type and the other Period value expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR Period value expression is implicitly converted, before comparison, to the data type of the Period value expression based on the format of the Period value expression. Their behavior should be easily understandable from a reading of the previous operators. **Note:** NE, NOT=, ^=, GT, GE, LT, and LE are non-ANSI operators.

For details on BEGIN and END, see Chapter 7: "Period Functions and Operators."

CHAPTER 5 Set Operators

This chapter describes SQL set operators.

Overview of Set Operators

The SQL set operators manipulate the results sets of two or more queries by combining the results of each individual query into a single results set.

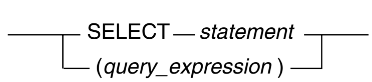
Teradata SQL Set Operators

Teradata SQL supports the following set operators:

Set Operator	Function
INTERSECT	Returns result rows that appear in all answer sets generated by the individual SELECT statements.
MINUS / EXCEPT	Result is those rows returned by the first SELECT except for those also selected by the second SELECT. MINUS is the same as EXCEPT.
UNION	Combines the results of two or more SELECT statements.

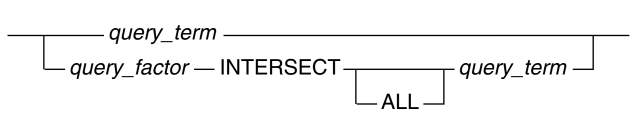
Set operators appear in query expressions. A query expression is a set of queries combined by the set operators INTERSECT, MINUS/EXCEPT, and UNION.

Syntax for *query_term*



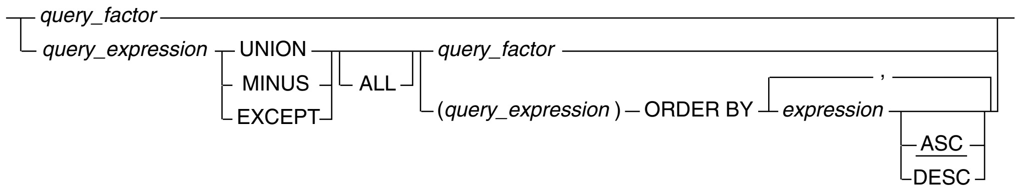
FF07D178

Syntax for *query_factor*



HH01A061

Syntax for *query_expression*



FF07D179

where:

Syntax Element ...	Specifies ...
query_term	
SELECT *statement*	a SELECT statement. For details, see *SQL Data Manipulation Language*.
query_expression	an optional expression that might or might not include set operators, other expressions, and an ORDER BY clause.
query_factor	
INTERSECT	a set operator returning the result rows appearing in all answer sets.
ALL	an optional keyword, allowing duplicate rows to be returned.
query_expression	
UNION MINUS/EXCEPT	optional set operators specifying how the two or more queries or subqueries are to combine and determine what result rows are required to be returned.
ALL	an optional keyword, allowing duplicate rows to be returned.
ORDER BY	the ORDER BY clause to order the result rows returned. For details, see *SQL Data Manipulation Language*.
expression	an expression used in the ORDER BY clause to determine the sort order of returned rows in the result.
ASC DESC	the sort order for the returned result rows. ASC is the default.

ANSI Compliance

INTERSECT, EXCEPT, and UNION are ANSI SQL:2008 compliant.

MINUS and the ALL option are Teradata extensions to the ANSI standard.

Rules for Set Operators

Duplicate Rows

By default, duplicate rows are not returned.

To permit duplicate rows to be returned, specify the ALL option. For an example, see "Retaining Duplicate Rows Using the ALL Option" on page 125.

Operations That Support Set Operators

You can use set operators within the following operations:

- Simple queries
- Derived tables

  Note: You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.
- Subqueries
- INSERT … SELECT clauses
- View definitions

SELECT statements connected by set operators can include all of the normal clause options for SELECT except the WITH clause.

SELECT AND CONSUME Statement

Set operations do not operate on SELECT AND CONSUME statements.

Support for ORDER BY Clause

A query expression can include only one ORDER BY specification, at the end.

Restrictions on the Data Types Involved in Set Operations

The following restrictions apply to CLOB, BLOB, and UDT types involved in set operations:

Data Type	Restrictions
BLOB	You cannot use set operators with CLOB or BLOB types.
CLOB	

Data Type	Restrictions
UDT	• Multiple UDTs involved in set operations must be identical types because Teradata Database does not perform implicit type conversion on UDTs involved in set operations. 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 set operation. • UDTs involved in set operations must have ordering definitions. Teradata Database 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 *SQL Data Definition Language*.

Precedence of Set Operators

The precedence for processing set operators is as follows:

1. INTERSECT
2. UNION and MINUS/EXCEPT

The set operators evaluate from left to right if no parentheses explicitly specify another order.

Example

For example, consider the following query.

```
SELECT statement_1
UNION
SELECT statement_2
EXCEPT
SELECT statement_3
INTERSECT
SELECT statement_4;
```

The operations are performed in the following order:

1. Intersect the results of statement_3 and statement_4.
2. Union the results of statement_1 and statement_2.
3. Subtract the intersected rows from the union.

Using Parentheses to Customize Precedence

To override precedence, use parentheses. Operations in parentheses are performed first.

For example, consider the following form:

```
( ( SELECT statement_1
    UNION
```

```
    SELECT statement_2 )
EXCEPT
  ( SELECT statement_3
    UNION
    SELECT statement_4 )
)
EXCEPT
SELECT statement_5
INTERSECT
SELECT statement_6;
```

The following list explains the precedence of operators for this example.

1 UNION SELECT `statement_1` and SELECT `statement_2`.
2 UNION SELECT `statement_3` and SELECT `statement_4`.
3 Subtract the result of the second UNION from the result of the first UNION.
4 INTERSECT SELECT `statement_5` and SELECT `statement_6`.
5 Subtract the INTERSECT result from the remainder of the UNION operations.

Retaining Duplicate Rows Using the ALL Option

Unless you specify the ALL option, duplicate rows are eliminated from the final result. The ALL option retains duplicate rows for the result set to which it is applied.

Example

The following query returns duplicate rows for each result set, including the final:

```
SELECT statement_1
UNION ALL
SELECT statement_2
MINUS ALL
SELECT statement_3
INTERSECT ALL
SELECT statement_4
```

Attributes of a Set Result

The data type, title, and format clauses contained in the first SELECT statement determine the data type, title, and format information that appear in the final result.

Attributes for all other SELECT statements in the query are ignored.

Example 1

```
SELECT level, param, 'GMKSA' (TITLE 'OWNER')
FROM gmksa
WHERE cycle = '03'
UNION
```

```
SELECT level, param, 'GMKSA CONTROL'
FROM gmksa_control
WHERE cycle = '03'
ORDER BY 1, 2;
```

The query returns the following results set:

```
***QUERY COMPLETED. 5 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL  PARAM  OWNER
-----  -----  -----
00     A      GMKSA
00     T      GMKSA
85     X      GMKSA
SF     A      GMKSA
SF     T      GMKSA
```

The first SELECT specifies GMKSA, which is CHAR(5)—that data type is then forced on the second SELECT. As a result, GMKSA_CONTROL entries are dropped because the first five characters are the same.

Because this query does not specify the ALL option, duplicate rows are dropped.

Example 2

In the next query, the SELECT order is reversed:

```
SELECT level, param 'GMKSA CONTROL' (TITLE 'OWNER')
FROM gmksa_control
WHERE cycle = '03'
UNION
SELECT level, param, 'GMKSA'
FROM gmksa
WHERE cycle = '03'
ORDER BY 1, 2;
```

This query returns the following answer set:

```
***QUERY COMPLETED.10 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL   PARAM     OWNER
-----   -----     -------------
00      A         GMKSA
00      A         GMKSA CONTROL
00      T         GMKSA
00      T         GMKSA CONTROL
85      X         GMKSA
85      X         GMKSA CONTROL
SF      A         GMKSA
SF      A         GMKSA CONTROL
SF      T         GMKSA
SF      T         GMKSA CONTROL
```

In this case, because the first SELECT specified 'GMKSA CONTROL', the rows were not duplicates and were included in the answer set.

Example 3

This example demonstrates how a poorly formed query can cause truncation of the results.

```
SELECT level, param, 'GMKSA       ' (TITLE 'OWNER')
```

```
FROM gmksa
WHERE cycle = '03'
UNION
SELECT level, param,'GMKSA CONTROL'
FROM gmksa_control
WHERE cycle = '03'
ORDER BY 1, 2;
```

This query returns the following answer set:

```
***QUERY COMPLETED.10 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL   PARAM   OWNER
-----   -----   ------------
00      A       GMKSA
00      A       GMKSA CONTRO
00      T       GMKSA
00      T       GMKSA CONTRO
85      X       GMKSA
85      X       GMKSA CONTRO
SF      A       GMKSA
SF      A       GMKSA CONTRO
SF      T       GMKSA
SF      T       GMKSA CONTRO
```

This query returned the expected rows; note, however, that because of the way the name was specified in the first SELECT, there was some truncation.

Set Operators With Derived Tables

Derived tables support set operators, as demonstrated in the following example:

Example

```
SELECT x1
FROM table_1,
(SELECT x2
FROM table_2
UNION
SELECT x3
FROM table_3
) derived_table;

SELECT x1,y1
FROM table_1,
(SELECT *
FROM table_2) derived_table(column_1, column_2)
WHERE column_2 = 1 ;
```

Restrictions

You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators. The following example returns an error.

Example

The following table function "add2int" takes two integers as input and returns the two integers and their summation.

```
CREATE TABLE t1 (a1 INTEGER, b1 INTEGER);
CREATE TABLE t2 (a2 INTEGER, b2 INTEGER);

REPLACE FUNCTION add2int
   (a INTEGER,
    b INTEGER)
RETURNS TABLE
   (addend1 INTEGER,
    addend2 INTEGER,
    mysum INTEGER)
SPECIFIC add2int
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'CS!add3int!add2int.c';

/* Query Q1 */
WITH dt(a1, b1) AS
( SELECT a1, b1
  FROM t1
  UNION ALL
  SELECT a2, b2
  FROM t2
)
SELECT *
FROM TABLE (add2int(dt.a1, dt.b1)
HASH BY b1
LOCAL ORDER BY b1) tf;
```

Set Operators in Subqueries

Set operators are permitted in subqueries. The following examples demonstrate their correct use.

Example 1

```
SELECT x1
FROM table_1
WHERE (x1,y1) IN
(SELECT * FROM table_2
UNION
SELECT * FROM table_3);
```

Example 2

```
SELECT *
FROM table_1
WHERE table_1.x1 IN
```

```
(SELECT x2
FROM table_2
UNION
(SELECT x3
FROM table_3
UNION
SELECT x4
FROM table_4));
```

Example 3

```
SELECT *
FROM table_1
WHERE x1 IN
(SELECT SUM(x2)
FROM table_2
UNION
SELECT x3
FROM table_3);
```

Example 4

```
SELECT *
FROM table_1
WHERE x1 IN
(SELECT MAX(x2)
FROM table_2
UNION
SELECT MIN(x3)
FROM table_3);
```

Example 5

```
SELECT *
FROM table_1
WHERE X1 IN
(SELECT x2 FROM table_2
UNION
SELECT x3 FROM table_3
UNION
SELECT x4 FROM table_4);
```

Example 6

```
SELECT x1
FROM table_1
WHERE x1 IN ANY
(SELECT x2 FROM table_2
INTERSECT
SELECT x3 FROM table_3
MINUS
SELECT x4 FROM table_4);
```

Example 7

```
UPDATE table_1
SET x1=1
```

```
WHERE table_1.x1 IN
(SELECT x2
FROM table_2
UNION
SELECT x3
FROM table_3
UNION
SELECT x4
FROM table_4);
```

Set Operators in INSERT … SELECT Statements

Set operators are permitted in INSERT … SELECT statements. The following examples demonstrate their correct use.

Example 1

The first example demonstrates a simple INSERT … SELECT using set operators.

```
INSERT table1 (x1,y1)
SELECT *
FROM table_2
UNION
SELECT x3,y3
FROM table_3;
```

Example 2

The second example demonstrates an INSERT … SELECT from a view that uses set operators.

```
REPLACE VIEW v AS
SELECT *
FROM table_1
UNION
SELECT *
FROM table_2;

INSERT table_3(x3,y3)
SELECT *
FROM v;
```

Example 3

This example demonstrates an INSERT … SELECT from a derived table with set operators.

```
INSERT table_1
SELECT *
FROM
(SELECT x2,y2
FROM table_2
UNION
SELECT *
FROM table_3 DerivedTable
);
```

Set Operators in View Definitions

Set operators are permitted within view definitions.

For example, the following REPLACE VIEW statement uses UNION within a view definition:

```
REPLACE VIEW view_1 AS
   SELECT x1,y1
   FROM table_1
   UNION
   SELECT x2,y2
   FROM table_2;
```

Support for the GROUP BY Clause

GROUP BY can be used within views with set operators. For details, see "GROUP BY and ORDER BY Clauses" on page 134.

Restrictions

The following limitations apply to view definitions that specify set operators:

- UPDATE, DELETE, and INSERT are not applicable. The following example does *not* work:

```
REPLACE VIEW V AS
SELECT X
FROM TABLE_1
UNION
SELECT Y FROM
TABLE_1;

UPDATE V
SET X=0;
```

  An attempt to perform this sequence of statements produces the following error message:

```
***Failure 3823 VIEW 'v' may not be used for Help Index/
Constraint/Statistics, Update, Delete or Insert.
```

- WITH CHECK OPTION is not applicable. The following example does *not* work:

```
REPLACE VIEW ERRV( c ) AS
SELECT *
FROM TABLE_1
UNION
SELECT *
FROM TABLE_2
WHERE TABLE_2.X=2 WITH CHECK OPTION;
```

  An attempt to perform this statement causes the following error message:

```
***Failure 3847 Illegal use of a WITH clause.
```

- Column level access rights cannot be granted. The following example does *not* work:

```
GRANT UPDATE ( c ) ON TABLE_VIEW TO USER_NAME;
```

  An attempt to perform this statement causes the following error message:

```
   ***Failure 3499: GRANT cannot be used on views with set
operators.
```

- A view definition that uses set operators cannot specify an ORDER BY clause, but a SELECT statement applied on the view can use ORDER BY. For details, see "GROUP BY and ORDER BY Clauses" on page 134.

Examples

The following examples provide correct uses of set operators within view definitions.

Example 1

```
REPLACE VIEW v AS
   SELECT x1
   FROM TABLE_1
   UNION
   SELECT x2
   FROM TABLE_2
   UNION
```

```
   SELECT x3
   FROM TABLE_3;

SELECT * FROM v;
```

Example 2

```
REPLACE VIEW view_2 AS
   SELECT *
   FROM view_1
   UNION
   SELECT *
   FROM table_3
   UNION
   SELECT *
   FROM table_4;

SELECT *
FROM view_2
ORDER BY 1,2;
```

Example 3

```
REPLACE VIEW v AS
   SELECT x1
   FROM table_1
   WHERE x1 IN
    (SELECT x2
     FROM table_2
     UNION
     SELECT x3
     FROM table_3
    );

SELECT * FROM v;
```

Queries Connected by Set Operators

Certain rules and restrictions apply to SELECT statements connected by set operators that might not apply elsewhere.

Number of Expressions in SELECT Statements

All SELECT statements must have the same number of expressions.

If the first SELECT statement contains three expressions, all succeeding SELECT statements must contain three expressions.

You can use a null expression in a SELECT statement as a place holder for a missing expression.

In the following example, the second expression is null.

```
SELECT EmpNo, NULL (CHAR(5))
FROM Employee;
```

WITH Clause

WITH clauses cannot be used in SELECT statements connected by set operators.

GROUP BY and ORDER BY Clauses

GROUP BY clauses are allowed in individual SELECT statements of a query expression but apply only to that SELECT statement and not to the result set.

ORDER BY clauses are allowed only in the last SELECT statement of a query expression and specify the order of the result set.

ORDER BY clauses can contain only numeric literals.

For example, to order by the first column in your result set, specify ORDER BY 1.

View definitions with set operators can use GROUP BY but cannot use ORDER BY. A SELECT statement applied to a view definition with set operators can use GROUP BY and ORDER BY. The following examples are correct uses of these operations within a view definition:

```
REPLACE VIEW v AS
SELECT x1,y1
FROM table1
UNION
SELECT x2,y2
FROM table2;

SELECT *
FROM v
ORDER BY 1;

SELECT SUM(x1), y1
FROM v
GROUP BY 2;
```

You can also apply independent GROUP BY operations to each unioned SELECT. The following example demonstrates how to do this:

```
REPLACE VIEW v(column_1,column_2) AS
SELECT MIN(x1),y1
FROM table_1
GROUP BY 2
UNION ALL
SELECT MIN(x2),y2
FROM table_2
GROUP BY 2
UNION ALL
SELECT x3,y3 FROM table_3;

SELECT SUM(v.column_1) (NAMED sum_c1),column_2
GROUP BY 2
ORDER BY 2;

SELECT *
FROM table_1
```

```
WHERE (x1,y1) IN
(SELECT SUM(x2), y2
FROM table_2
GROUP BY 2
UNION
SELECT SUM(x3), y3
FROM table_3
GROUP BY 2
);
```

Table Name in SELECT Statements

Each SELECT statement must identify the table that the data is to come from even if all SELECT statements reference the same table.

Data Type Compatibility

Corresponding fields in each SELECT statement must have data types that are compatible. For example, if the first field in the first SELECT statement is a character data type, then the first field in each succeeding SELECT statement must be a character data type.

Corresponding numeric types do not have to be the same, but they must be compatible. For example, a field in one SELECT statement can be defined as INTEGER and the corresponding field in another SELECT statement can be defined as SMALLINT.

The data types in the first SELECT statement determine the data types of corresponding columns in the result set.

The following table provides details about data type compatibility.

Data Type	Details
Character	Character types in the first SELECT statement determine the length of character strings in the result set. This can lead to truncation of character strings in the result set if the length of a character type in the first SELECT statement is less than the length of corresponding character types in succeeding SELECT statements.
Numeric	Numeric types in the first SELECT statement determine the size of numeric types in the result set. All corresponding numeric fields in succeeding SELECT statements are converted to the numeric data type in the first SELECT statement. This can lead to a numeric overflow error if the size of a numeric type in the first SELECT statement is smaller than the size of corresponding numeric types in succeeding SELECT statements and the values returned by the succeeding statements do not fit into the smaller data type.

Data Type	Details
TIME TIMESTAMP PERIOD(TIME) PERIOD(TIMESTAMP)	TIME, TIMESTAMP, PERIOD(TIME), and PERIOD(TIMESTAMP) types in the first SELECT statement determine the precision of corresponding columns in the result set. All corresponding fields in succeeding SELECT statements are implicitly converted to the data type in the first SELECT statement. If a corresponding field does not have a time zone and the data type in the first SELECT statement does, the time zone is set to the current session time zone displacement. If the precision of a corresponding field is lower than the precision of the data type in the first SELECT statement, trailing zeros are appended to the fractional digits as needed. If the precision of corresponding fields in succeeding SELECT statements is higher than the precision of the data type in the first SELECT statement, an error is reported.

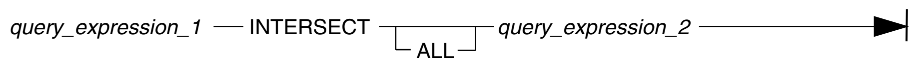
For examples that show how the length of the character type in the first SELECT statement affects the result set, see "Attributes of a Set Result" on page 125. For examples that show how the numeric data type in the first SELECT statement affects the result set, see "Example 6: Effect of the Order of SELECT Statements on Data Type" on page 148.

INTERSECT Operator

Purpose

Returns only the rows that exist in the result of both queries.

Syntax



FF07D176

where:

Syntax element ...	Specifies ...
query_expression_1	a complete SELECT statement to be INTERSECTed with *query_expression_2*. See “Syntax for query_factor” on page 121.
ALL	that duplicate rows are to be retained for the INTERSECT.
query_expression_2	a complete SELECT statement to be INTERSECTed with *query_expression_1*. See “Syntax for query_term” on page 121.

ANSI Compliance

INTERSECT is ANSI SQL:2008 compliant.

The ALL option is a Teradata extension to the ANSI standard.

Rules for INTERSECT

The following rules apply to the use of INTERSECT:

- In addition to using INTERSECT within simple queries, you can use INTERSECT within the following operations:
  - Derived tables

    Note: You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.
  - Subqueries
  - INSERT … SELECT statements
  - View definitions
- Each query connected by INTERSECT is executed to produce a result consisting of a set of rows. The intersection must include the same number of columns from each table in each

SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible.

- INTERSECT cannot be used within the following:
  - SELECT AND CONSUME statements.
  - WITH RECURSIVE clause
  - CREATE RECURSIVE VIEW statements

Attributes of a Set Result

The data type, title, and format clauses contained in the first SELECT statement in the intersection determine the data type, title, and format information that appear in the final result.

Attributes for all other SELECT statements in the query are ignored.

Data Type of Nulls

When you specify an explicit NULL for any intersection operation, its data type is INTEGER. For an example of this principle using the UNION operator, see "Example 5: Effect of Explicit NULLs on Data Type of a UNION" on page 147.

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

Duplicate Row Handling

Unless the ALL option is used, duplicate rows are eliminated from the final result.

If the ALL option is specified, duplicate rows are retained. The ALL option can be specified for as many INTERSECT operators as are used in a multistatement query.

Example

Assume that two tables contain the following rows:

SPart table

SuppNo	PartNo
100	P2
101	P1
102	P1
103	P2

SLocation table

SuppNo	SuppLoc
100	London
101	London
102	Toronto
103	Tokyo

To then select supplier number (SuppNo) for suppliers located in London (SuppLoc) who supply part number P1 (PartNo), use the following request:

```
SELECT SuppNo FROM SLocation
WHERE SuppLoc = 'London'
INTERSECT
SELECT SuppNo FROM SPart
WHERE PartNo = 'P1';
```

The result of this request is:

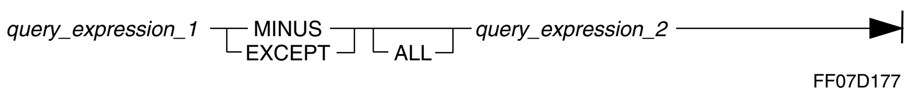
```
SuppNo
------
   101
```

MINUS/EXCEPT Operator

Purpose

Returns the results rows that appear in *query_expression_1* and *not* in *query_expression_2*.

Syntax



where:

Syntax element …	Specifies …
query_expression_1	a complete SELECT statement whose results table is to be MINUSed with *query_expression_2*.
ALL	that duplicate rows are to be retained for the MINUS operation.
query_expression_2	a complete SELECT statement to be MINUSed from *query_expression_1*.

ANSI Compliance

EXCEPT is ANSI SQL:2008 compliant.

MINUS and the ALL option are Teradata extensions to the ANSI SQL:2008 standard.

Usage Notes

Besides simple queries, MINUS or EXCEPT can be used within the following operations:

- Derived tables

  Note: You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.

- Subqueries
- INSERT … SELECT statements
- View definitions

MINUS and EXCEPT cannot be used within the following operations:

- SELECT AND CONSUME statements.
- WITH RECURSIVE clause
- CREATE RECURSIVE VIEW statements

Each query connected by MINUS or EXCEPT is executed to produce a result consisting of a set of rows. The exception must include the same number of columns from each table in each SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible. All the result sets are then combined into a single result set, which has the data types of the columns specified in the *first* SELECT statement in the exception.

MINUS/EXCEPT and NULL

When you specify an explicit NULL for any exception operation, its data type is INTEGER. For an example of this principle using the UNION operator, see "Example 5: Effect of Explicit NULLs on Data Type of a UNION" on page 147.

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

Duplicate Rows

Unless the ALL option is used, duplicate rows are eliminated from the final result.

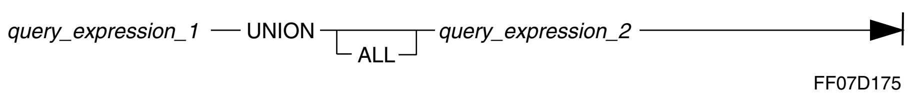
If the ALL option is specified, duplicate rows are retained. The ALL option can be specified for as many MINUS operators as are used in a multistatement query.

UNION Operator

Purpose

Combines two or more SELECT results tables into a single result.

Syntax



where:

Syntax element ...	Specifies ...
query_expression_1	a complete SELECT statement to be unioned with *query_expression_2*. For details, see "Syntax for query_expression" on page 122.
ALL	that duplicate rows are to be retained for the UNION.
query_expression_2	a complete SELECT statement to be unioned with *query_expression_1*. For details, see "Syntax for query_factor" on page 121.

ANSI Compliance

UNION is ANSI SQL:2008 compliant.

Valid UNION Operations

Besides simple queries, UNION can be used within the following operations:

- Derived tables

  Note: You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.
- Subqueries
- INSERT … SELECT statements
- Non-recursive CREATE VIEW statements

UNION ALL is the only valid set operator in a WITH RECURSIVE clause or CREATE RECURSIVE VIEW statement that defines a recursive query.

Unsupported Operations

UNION cannot be used within the following:

- SELECT AND CONSUME statements.
- WITH RECURSIVE clause (unless the ALL option is also specified)
- CREATE RECURSIVE VIEW statements (unless the ALL option is also specified)

Description of a UNION Operation

Each query connected by UNION is performed to produce a result consisting of a set of rows. The union must include the same number of columns from each table in each SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible. All the result sets are then combined into a single result set that has the data type of the columns specified in the *first* SELECT statement in the union. For an example, see "Example 6: Effect of the Order of SELECT Statements on Data Type" on page 148.

UNION and NULL

When you specify an explicit NULL for any union operation, its data type is INTEGER. For an example, see "Example 5: Effect of Explicit NULLs on Data Type of a UNION" on page 147.

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

Duplicate Rows

Unless the ALL option is used, duplicate rows are eliminated from each result set and from the final result.

If the ALL option is used, duplicate rows are retained for the applicable result set.

You can specify the ALL option for each UNION operator in the query to retain every occurrence of duplicate rows in the final result.

Unexpected Row Length Errors: Sorting Rows for UNION

Before performing the sort operation used to check for duplicates in some union operations, Teradata Database creates a sort key and appends it to the rows to be sorted. If the length of this temporary data structure exceeds the system limit of 64K bytes, the operation fails and returns an error to the requestor. Depending on the situation, the message text is one of the following:.

- A data row is too long.
- Maximum row length exceeded in *database_object_name*.

See *Messages* for explanations of these messages.

Example 1

To select the name, project, and the number of hours spent by employees assigned to project OE1-0001, plus the names of employees not assigned to a project, the following query could be used:

```
SELECT Name, Proj_Id, Hours
FROM Employee,Charges
WHERE Employee.Empno = Charges.Empno
AND Proj_Id IN ('OE1-0001')
UNION
SELECT Name, NULL (CHAR (8)), NULL (DECIMAL (4,2))
FROM Employee
WHERE Empno NOT IN
(SELECT Empno
FROM Charges);
```

This query returns the following rows:

Name	Project Id	Hours
Aguilar J	?	?
Brandle B	?	?
Chin M	?	
Clements D	?	?
Kemper R		
Marston A	?	?
Phan A	?	?
Regan R	?	?
Russell S	?	?
Smith T		
Watson L		
Inglis C	0E1-0001	30.0
Inglis C	0E1-001	30.5
Leidner P	0E1-001	10.5
Leidner P	0E1-001	23.0
Moffit H	0E1-001	12.0
Moffit H	0E1-001	35.5

In this example, null expressions are used in columns 2 and 3 of the second SELECT statement. The null expressions are used as place markers so that both SELECT statements in the query contain the same number of expressions.

Example 2

To determine the department number and names of all employees in departments 500 and 600, the UNION operator could be used as follows:

```
SELECT DeptNo, Name
FROM Employee
WHERE DeptNo = 500
UNION
SELECT DeptNo, Name
FROM Employee
WHERE DeptNo = 600 ;
```

This query returns the following rows:

DeptNo	Name
500	Carter J
500	Inglis C
500	Marston A
500	Omura H
500	Reed C
500	Smith T
500	Watson L
600	Aguilar J
600	Kemper R
600	Newman P
600	Regan R

The same results could have been returned with a simpler query, such as the following:

```
SELECT Name, DeptNo
FROM Employee
WHERE (DeptNo = 500)
OR (DeptNo = 600);
```

The advantage to formulating the query using the UNION operator is that if the DeptNo column is the primary index for the Employee table, then using the UNION operator guarantees that the basic selects are prime key operations. There is no guarantee that a query using the OR operation will make use of the primary index.

Example 3

In addition, the UNION operator is useful if you must merge lists of values taken from two or more tables.

For example, if departments 500 and 600 had their own Employee tables, the following query could be used to select data from two different tables and merge that data into a single list:

```
SELECT Name, DeptNo
FROM Employee_dept_500
UNION
SELECT Name, DeptNo
```

```
FROM Employee_dept_600 ;
```

Example 4

Suppose you want to know the number of man-hours charged by each employee who is working on a project. In addition, suppose you also wanted the result to include the names of employees who are not working on a project.

To do this, you would have to perform a union operation as illustrated in the following example.

```
SELECT Name, Proj_Id, Hours
FROM Employee, Charges
WHERE Employee.EmpNo = Charges.EmpNo
UNION
SELECT Name, Null (CHAR(8)), Null (DECIMAL(4,2)),
FROM Employee
WHERE EmpNo NOT IN
(SELECT EmpNo
FROM Charges
)
UNION
SELECT Null (VARCHAR(12)), Proj_Id, Hours
FROM Charges
WHERE EmpNo NOT IN
(SELECT EmpNo
FROM Employee
);
```

The first portion of the statement joins the Employee table with the Charges table on the EmpNo column. The second portion accounts for the employees who might be listed in the Employee table, but not the Charges table. The third portion of the statement accounts for the employees who might be listed in the Charges table and not in the Employee table. This ensures that all the information asked for is included in the response.

UNION Operator and the Outer Join

"Example 4" on page 146 does not illustrate an outer join. That operation returns all rows in the joined tables for which there is a match on the join condition and rows from the "left" join table, or the "right" join table, or both tables for which there is no match. Moreover, non-matching rows are extended with null values.

It is possible, however, to achieve an outer join using inner joins and the UNION operator, though the union of any two inner joins is not the equivalent of an outer join.

The following example shows how to achieve an outer join using two inner joins and the UNION operator. Notice how the second inner join uses null values.

```
   SELECT Offering.CourseNo, Offerings.Location, Enrollment.EmpNo
FROM Offerings, Enrollment
WHERE Offerings.CourseNo = Enrollment.CourseNo
UNION
SELECT Offerings.CourseNo, Offerings.Location, NULL
FROM Offerings, Enrollment
WHERE Offerings.CourseNo <> Enrollment.CourseNo;
```

The above UNION operation returns results equivalent to the results of the left outer join example shown above.

O.CourseNo	O.Location	E.EmpNo
C100	El Segundo	235
C100	El Segundo	668
C200	Dayton	?
C400	El Segundo	?

Example 5: Effect of Explicit NULLs on Data Type of a UNION

Set operator results evaluate to the data type of the columns defined in the first SELECT statement in the operation. When a column in the first SELECT is defined as an explicit NULL, the data type of the result is not intuitive.

Consider the following two examples, which you might intuitively think would evaluate to the same result but do not.

In the first, an explicit NULL is selected as a column value.

```
SELECT 'p', NULL
FROM TableVM
UNION
SELECT 'q', 145.87
FROM TableVM;
```

BTEQ returns the result as follows.

```
'p'          Null
---   -----------
p               ?
q             145
```

The expected value for the second row of the Null column probably differs from what you might expect—a decimal value of 145.87.

What if the order of the two SELECTs in the union is reversed?

```
SELECT 'q', 145.87
FROM TableVM
UNION
SELECT 'p', NULL
FROM TableVM;
```

BTEQ returns the result as follows.

```
'q'        145.87
---   -----------
p               ?
q          145.87
```

The value for q is now reported as its true data type—DECIMAL—and without truncation. Why the difference?

In the first union example, the explicit NULL is specified for the second column in the first SELECT statement. The second column in the second SELECT statement, though specified as a DECIMAL number, evaluates to an integer because in this context, NULL, though having no value, *does* have the data type INTEGER, and that type is retained for the result of the union.

The second union example carries the data type for the value 145.87—DECIMAL—through to the result.

You can confirm the unconverted data type for NULL and 145.87 by performing the following SELECT statement.

```
SELECT TYPE(NULL), TYPE(145.87)
```

BTEQ returns the result as follows.

```
Type(Null)          Type(145.87)
-----------------   ----------------------
INTEGER             DECIMAL(5,2)
```

Example 6: Effect of the Order of SELECT Statements on Data Type

The result of any UNION is always expressed using the data type of the selected value of the first SELECT. This means that SELECT A UNION SELECT B does not always return the same result as SELECT B UNION SELECT A unless you explicitly convert the output data type to ensure the same result in either case.

Consider the following complex unioned queries:

```
SELECT MIN(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND X8.i1 IN
(SELECT COUNT(*)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND X8.i1 = ANY
(SELECT COUNT(*)
FROM t7 X7
WHERE X7.i1 = ANY
(SELECT AVG(X1.i1)
FROM t1 X1)))
UNION
SELECT AVG(X4.i1)
FROM t4 X4
WHERE X4.i1 = ANY
(SELECT (X8.i1)
FROM t1 X1
RIGHT JOIN t8 X8 ON X8.i1=X1.i1
AND X8.i1 = IN
(SELECT MAX(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND
(SELECT (X4.i1)
FROM t6 X6
RIGHT JOIN t4 X4 ON X6.i1=i1))));
```

The result is the following report.

```
Minimum(i1)
-----------
         -2
          0
```

You might intuitively expect that reversing the order of the queries on either side of the UNION would produce the same result. Because the data types of the selected value of the first SELECT can differ, this is not always true, as the following query on the same database demonstrates.

```
SELECT AVG(X4.i1)
FROM t4 X4
WHERE X4.i1 = ANY
(SELECT (X8.i1)
FROM t1 X1
RIGHT JOIN t8 X8 ON X8.i1 = X1.i1
AND X8.i1 = ANY
(SELECT MAX(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND
(SELECT (X4.i1)
FROM t6 X6
RIGHT JOIN t4 X4 ON X6.i1 = i
)
)
)
UNION
SELECT MIN(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND X8.i1 IN
(SELECT COUNT(*)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND X8.i1 = ANY
(SELECT COUNT(*)
FROM t7 X7
WHERE X7.i1 = ANY
(SELECT AVG(X1.i1)
FROM t1 X1
)
);
```

The result is the following report.

```
Average(i1)
-----------
         -2
          1
```

The actual average is < 0.5. Why the difference when the order of SELECTs in the UNION is reversed? The following table explains the seemingly paradoxical results.

WHEN the first SELECT specifies this function …	The result data type is …	AND the value returned as the result is …
AVG	REAL	1
MIN	INTEGER	truncated to 0

CHAPTER 6 DateTime and Interval Functions and Expressions

This chapter describes functions and expressions that operate on ANSI DateTime and Interval values, and also describes functions and expressions that operate on Teradata DATE values, which are extensions to the ANSI SQL:2008 standard.

Overview

ANSI DateTime Data Types

ANSI DateTime data types include:

- DATE
- TIME
- TIME WITH TIME ZONE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Interval Data Types

There are two categories of ANSI Interval data types:

- Year-Month Intervals, which include:
  - YEAR
  - YEAR TO MONTH
  - MONTH
- Day-Time Intervals, which include:
  - DAY
  - DAY TO HOUR
  - DAY TO MINUTE
  - DAY TO SECOND
  - HOUR
  - HOUR TO MINUTE
  - HOUR TO SECOND
  - MINUTE
  - MINUTE TO SECOND
  - SECOND

ANSI DateTime and Interval Data Type Assignment Rules

Data Type Compatibility and Conversion

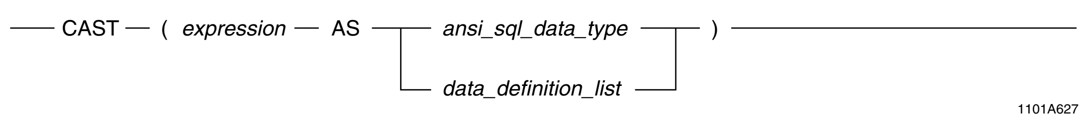
The following rules apply to assignments involving ANSI DateTime or Interval data types:

IF the source type is ...	AND the target type is ...	THEN ...
DATE	DATE	the types are compatible and assignments do not require conversion. For compatibility with existing Teradata assignments, non-ANSI operations such as assigning a DATE to an INTEGER or an INTEGER to a DATE (with validity checking) follow existing Teradata assignment rules.
TIME	TIME	the types are compatible and assignments do not require conversion. The Teradata system value TIME is encoded as a REAL and is not compatible with ANSI TIME or TIME WITH TIME ZONE.
TIMESTAMP	TIMESTAMP	the types are compatible and assignments do not require conversion.
Year-Month INTERVAL	Year-Month INTERVAL	
Day-Time INTERVAL	Day-Time INTERVAL	
Numeric	DATE	Teradata Database performs implicit type conversion before the assignment. See “Implicit Type Conversions” on page 577 for details.
DATE	• Character • Numeric • TIMESTAMP	
Character	• DATE • TIME • TIMESTAMP	
TIME	TIMESTAMP	
TIMESTAMP	• DATE • TIME	
Interval[a]	Exact Numeric	
Exact Numeric	Interval[a]	

a. The INTERVAL type must have only one field, e.g. INTERVAL YEAR.

For all other source/target data type combinations in assignments involving ANSI DateTime or Interval data types, the types must be explicitly converted.

To perform explicit conversions on ANSI DateTime or Interval data types, use the CAST function:



where:

Syntax element ...	Specifies ...
expression	an expression with known data type to be cast as a different data type.
ansi_sql_data_type	the new data type for *expression*.
data_definition_list	the new data type or data attributes or both for *expression*.

For more information, see "CAST in Explicit Data Type Conversions" on page 582.

Interval Data Type Assignment Rules

The following rules apply to Year-Month INTERVAL assignments.

WHEN ...	THEN ...
the types match	assignment is straightforward.
the source is INTERVAL YEAR and the target is INTERVAL YEAR TO MONTH	the value for MONTH in the target is set to zero.
the source is INTERVAL MONTH and the target is INTERVAL YEAR TO MONTH	the source is extended to include the YEAR field initialized to zero, and the resulting interval is normalized. For example, if the source is '15' then the extended source is '0-15', normalized to '1-03'.
the target is INTERVAL MONTH and the source is either INTERVAL YEAR or INTERVAL YEAR TO MONTH	the source is converted to INTERVAL MONTH before assignment. For example, if the source is '2-11', it is converted to '35'.
the least significant field of the source is lower than that of the target	the values of fields in the source with precision lower than the least significant field of the target are truncated. For example, if a source of INTERVAL '32' MONTH is assigned to a target column of type INTERVAL YEAR, the value stored is '2'.

The following rules apply to Day-Time INTERVAL assignments.

WHEN ...	THEN ...
the types match	assignment is straightforward.
the target is of lower significance than the least significant field of the source	values for those fields are set to zero. For example, if the source is INTERVAL '49:30' HOUR TO MINUTE and it is assigned to a target column of type INTERVAL HOUR(4) TO SECOND(2), the value stored is '49:30:00.00'.
the target has fields of higher significance than the most significant field of the source	the source type is extended to match the target type, setting the new fields to zeros, and normalizing the content as the final step. For example, if the source is INTERVAL '49:30' HOUR TO MINUTE and it is assigned to a target column of type INTERVAL DAY TO MINUTE, the value stored is '2 1:30'.
the least significant field of the source is lower than that of the target	the values of fields in the source with precision lower than the least significant field of the target are truncated. For example, if the source is INTERVAL '10:12:58' HOUR TO SECOND and it is assigned to a target column of type INTERVAL HOUR TO MINUTE, the value stored is '10:12'.

Scalar Operations on ANSI SQL:2008 DateTime and Interval Values

Teradata SQL defines a set of permissible scalar operations for ANSI DateTime and Interval values.

Scalar operations include:

Operation	Description
DateTime Expressions	Expressions providing a result that is a DateTime value. DateTime expressions have arguments that are also DateTime or Interval expressions.
Interval Expressions	Expressions providing a result that is an Interval. Interval expressions may include components that are Interval, DateTime, or Numeric expressions.

Data Type Compatibility

The Teradata Database convention of performing implicit conversions to resolve expressions of mixed data types is not supported for operations that include ANSI DateTime or Interval values.

To convert ANSI DateTime or Interval expressions, use the CAST function. See "CAST in Explicit Data Type Conversions" on page 582.

The following restrictions apply to the values appearing in all DateTime and Interval scalar operations:

IF ...	THEN ...
two DateTime values appear in the same DateTime expression	both must be DATE types ELSE both must be TIME types ELSE both must be TIMESTAMP types. You cannot mix DATE, TIME, and TIMESTAMP values across type.
a DateTime and Interval values appear in the same DateTime expression	the Interval value must contain only DateTime fields that are also contained within the DateTime value.
two Interval values appear in the same Interval expression	both must be Year-Month intervals ELSE both must be Day-Time intervals. You cannot mix Year-Month with Day-Time intervals.

ANSI DateTime Expressions

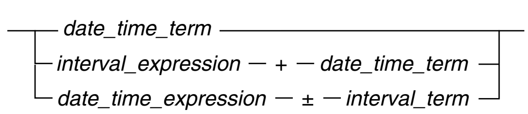
Purpose

Perform a computation on a DATE, TIME, or TIMESTAMP value (or value expression) and return a single value of the same type.

Definition

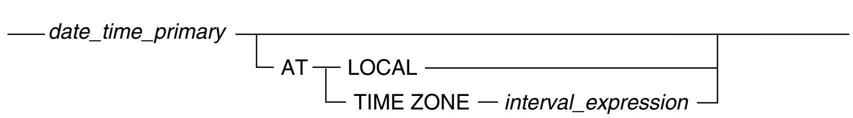
A DateTime expression is any expression that returns a result that is a DATE, TIME, or TIMESTAMP value.

date_time_expression Syntax



FF07D266

date_time_term Syntax



FF07D265

where:

Syntax element ...	Specifies ...
date_time_expression	an expression that evaluates to a DATE, TIME, or TIMESTAMP value. The form of the expression is one of the following: • a single *date_time_term* • the sum of an *interval_expression* and a *date_time_term* expression • the sum or difference of a *date_time_expression* and an *interval_term*
date_time_term	a single *date_time_primary* or a *date_time_primary* with a time zone specifier of LOCAL or TIME ZONE *displacement*.
interval_expression	one of the following: • a single *interval_term.* • an *interval_term* added to or subtracted from an *interval_expression*. • the difference between a *date_time_expression* and a *date_time_term* (enclosed by parentheses) preceding a start TO end phrase. For more information on *interval_expression* and *interval_term*, see "ANSI Interval Expressions" on page 160.
date_time_primary	one of the following elements, any of which must have the appropriate DateTime type: • Column reference • DateTime literal value For details on DateTime literals, see *SQL Data Types and Literals*. • DateTime function reference For example, the result of a CASE expression or CAST function or DateTime built-in function such as CURRENT_DATE or CURRENT_TIME. • Scalar function reference • Aggregate function reference • (*table_expression*) A scalar subquery. • (*date_time_timestamp_expression*)
AT LOCAL	the current default time zone displacement value for the session, expressed as the Interval data type used to define the local time zone offset.
AT TIME ZONE	a time zone displacement value expressed as type INTERVAL HOUR TO MINUTE.

Gregorian Calendar Rules

DateTime expressions always operate within the rules of the Gregorian calendar.

When an evaluation results in a value outside the permissible range for any contained field or results in a value impermissible according to the natural rules for DATE and TIME values, then an error is returned.

For example, the following operation returns an error because it evaluates to a date that is not valid ('1996-09-31').

```
SELECT DATE '1996-08-31' + INTERVAL '1' MONTH;
```

The desired result is obtained with a slight rephrasing of the second operand.

```
SELECT DATE '1996-08-31' + INTERVAL '30' DAY;
```

This operation returns the desired result, '1996-09-30'. No error is returned.

AT LOCAL and AT TIME ZONE Time Zone Specifiers

A *date_time_term* can include an AT LOCAL or AT TIME ZONE phrase only if the *date_time_term* evaluates to a TIME or TIMESTAMP value.

The effect is to adjust *date_time_term* to be in accordance with the specified time zone displacement value.

<table>
<tr><th>IF the type of the date_time_primary is …</th><th>THEN the time zone specifier is adjusted to …</th></tr>
<tr><td>DATE</td><td>an error.
You cannot specify a TIME ZONE with a DATE value.</td></tr>
<tr><td>TIME</td><td>TIME WITH TIME ZONE</td></tr>
<tr><td>TIMESTAMP</td><td>TIMESTAMP WITH TIME ZONE</td></tr>
<tr><td colspan="2">The time zone value is explicit in both cases.</td></tr>
<tr><td>TIME WITH TIME ZONE</td><td rowspan="2">the specified TIME ZONE, replacing the previous "current" value.</td></tr>
<tr><td>TIMESTAMP WITH TIME ZONE</td></tr>
</table>

The type of the *interval_expression* that specifies the time zone displacement value in an AT TIME ZONE phrase must be INTERVAL HOUR TO MINUTE and the limits for the range are -'12:59' to +'13:00.'

The time zone displacement value provided for AT LOCAL is always the local TIME ZONE offset with a data type having a WITH TIME ZONE specification.

Evaluation Types

Expressions involving DateTime values evaluate to a DateTime type, with DATE being the least significant type and TIMESTAMP the most significant.

DateTime expressions involving …	Evaluate to a …
Dates	date.
Times	time.
Timestamps	timestamp.

Adding and Subtracting Interval Values

DateTime expressions formed by adding an Interval to a DateTime value or by subtracting an Interval from a DateTime value are performed by adding or subtracting values of the appropriate component fields and carrying overflow from lower precision fields with the appropriate modulo to represent proper arithmetic in terms of the calendar and clock.

An *interval_expression* or *interval_term* may only contain DateTime fields that are contained in the corresponding *date_time_expression* or *date_time_term*.

When an Interval value is added to or subtracted from a TIME or TIMESTAMP value, the time zone displacement value associated with the result is identical to that associated with the TIME or TIMESTAMP value.

Computations With Time Zones

If you perform arithmetic on DateTime expressions containing time zones, the results are computed in the following way.

Call the DateTime value of the expression DV and the time zone value component (normalized to UTC) TZ.

The result is computed as DV - TZ.

Examples

The following examples illustrate various DateTime expressions using concrete instances.

Example 1: *date_time_primary*

In this example, the *date_time_primary* is a built-in time function.

```
CURRENT_TIME
```

Example 2: *date_time_term* With an Interval Column Time Zone Specifier

In this example, the *date_time_term* is a *date_time_primary* column value named f1.

TS.f1 is a value of type TIME or TIMESTAMP and intrvl.a is a column interval value of type INTERVAL HOUR TO MINUTE.

```
SELECT f1 AT TIME ZONE intrvl.a
FROM TS;
```

Example 3: *date_time_term* With an Interval Literal Time Zone Specifier

In this example, the *date_time_term* is a *date_time_primary* column value named f1.

The specified interval is an interval literal value of type INTERVAL HOUR TO MINUTE.

```
SELECT f1 AT TIME ZONE INTERVAL '01:00' HOUR TO MINUTE
FROM TS;
```

Example 4: *date_time_expression*

In this example, the *date_time_expression* is an *interval_expression* added to a *date_time_term*. Note that you can only add these terms—subtraction of a *date_time_term* from an *interval_expression* is not permitted.

```
SELECT INTERVAL '20' YEAR + CURRENT_DATE;
```

Example 5: *date_time_expression* With Addition

In this example, the *date_time_expression* is comprised of another *date_time_expression* added to an *interval_term*.

The columns *subscribe_date* and *subscription_interval* are typed DATE and INTERVAL MONTH(4), respectively.

```
SUBSCRIBE_DATE + SUBSCRIPTION_INTERVAL
```

Example 6: *date_time_expression* With Subtraction

You can also subtract an *interval_term* from a *date_time_expression*.

In this example, an *interval_term* is subtracted from the *date_time_expression*.

The columns *expiration_date* and *subscription_interval* are typed DATE and INTERVAL MONTH(4), respectively.

```
EXPIRATION_DATE - SUBSCRIPTION_INTERVAL
```

Time Zone Sort Order

Time zones are ordered chronologically, using the same time zone.

Examples

Consider the following examples using ordered SELECT statements on a table having a column with type TIMESTAMP(0) WITH TIME ZONE.

The identical ordering demonstrated in these ORDER BY SELECTs applies to all time zone comparison operations.

```
SELECT f1 TIMESTAMPFIELD
FROM timestwz
ORDER BY f1;
```

This statement returns the following results table.

```
TIMESTAMPFIELD
-------------------------
1997-10-07 15:43:00+08:00
1997-10-07 15:43:00-00:00
1997-10-07 15:47:52-08:00
```

Note how the values are displayed with the stored time zone information, but that the ordering is not immediately evident.

Now note how normalizing the time zones by means of a CAST function indicates chronological ordering explicitly.

```
SELECT CAST(f1 AS TIMESTAMP(0)) TIMESTAMP_NORMALIZED
FROM timestwz
ORDER BY f1;
```

This statement returns the following results table.

```
TIMESTAMP_NORMALIZED
-------------------
1997-10-06 23:43:00
1997-10-07 07:43:00
1997-10-07 15:45:52
```

While the ordering is the same as for the previous query, the display of TIMESTAMP values has been normalized to the time zone in effect for the session, which is '-08:00'.

A different treatment of the time zones, this time to reflect local time, indicates the same chronological ordering but from a different perspective.

```
SELECT f1 AT LOCAL LOCALIZED
FROM timestwz
ORDER BY f1;
```

This statement returns the following results table.

```
LOCALIZED
-------------------------
1997-10-06 23:43:00-08:00
1997-10-07 07:43:00-08:00
1997-10-07 15:45:52-08:00
```

ANSI Interval Expressions

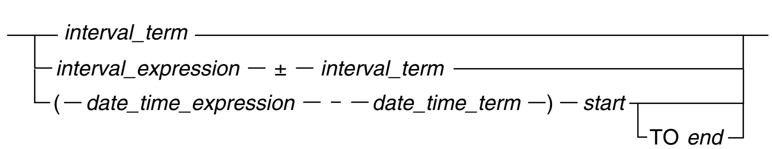
Purpose

Performs a computation on an Interval value (or value expression) and returns a single value of the same type.

Definition

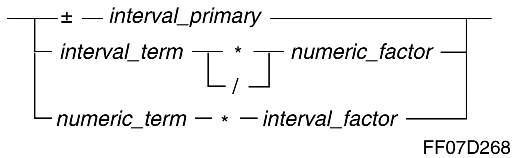
An interval expression is any expression that returns a result that is an INTERVAL value.

interval_expression Syntax

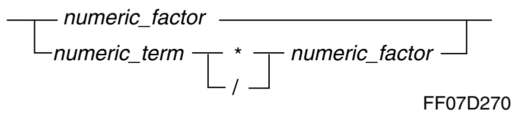


1101A010

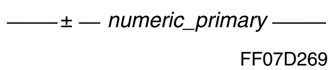
interval_term Syntax



numeric_term Syntax

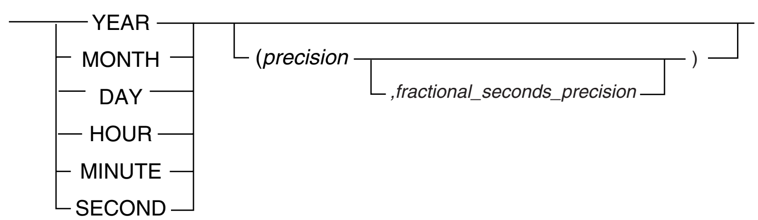
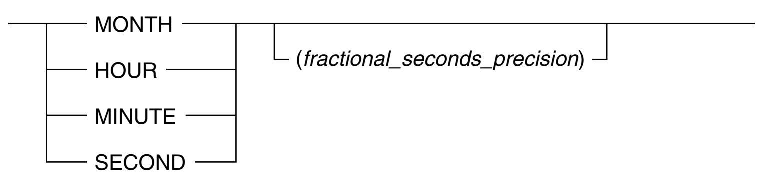


numeric_factor Syntax



where:

Syntax element …	Specifies …
interval_expression	an expression that evaluates to an INTERVAL value. The form of the expression is one of the following: • a single *interval_term* • the sum or difference of an *interval_term* and an *interval_expression* • the difference between a *date_time_expression* and a *date_time_term* (enclosed by parentheses) preceding a *start* TO *end* phrase
interval_term	one of the following expressions: • a single *interval_factor* • an *interval_term* multiplied or divided by a *numeric_factor* • the product of a *numeric_term* and an *interval_factor*
interval_factor	a signed *interval_primary*.
date_time_expression	an expression that evaluates to a DATE, TIME, or TIMESTAMP value. The form of the expression is one of the following: • a single *date_time_term* • the sum of an *interval_expression* and a *date_time_term* expression • the sum or difference of a *date_time_expression* and an *interval_term* For more information on *date_time_expression*, see “ANSI DateTime Expressions” on page 155.
date_time_term	a single *date_time_primary* or a *date_time_primary* with a time zone specifier of LOCAL or TIME ZONE *displacement*. For more information on *date_time_term*, see “ANSI DateTime Expressions” on page 155.

Syntax element ...	Specifies ...
start	a DateTime value with the following syntax that defines the beginning of a date or time interval:  1101A018 where: • *precision* specifies the permitted range of digits, ranging from one to four. The default precision is two. • *fractional_seconds_precision* specifies the fractional precision for values of SECOND, ranging from zero to six. The default is six. MONTH and SECOND values are only permitted when used without TO *end*.
TO *end*	a DateTime value with the following syntax that defines the end of a date or time interval:  1101A017 where *fractional_seconds_precision* specifies the fractional precision for values of SECOND, ranging from zero to six. The default is six. The value for *end* must be less significant than the value for *start*. If *start* is a YEAR value, then *end* must be a MONTH value.
numeric_factor	a signed *numeric_primary*.
numeric_term	a *numeric_factor* or a *numeric_term* multiplied or divided by a *numeric_factor*.

Syntax element …	Specifies …
numeric_primary	one of the following elements, any of which must have the appropriate numeric type: • Column reference • Numeric literal value • Scalar function reference • Aggregate function reference • (*table_expression*) A scalar subquery. • (*numeric_expression*)
interval_primary	one of the following elements, any of which must have the appropriate INTERVAL type: • Column reference • Interval literal value For details on Interval literals, see *SQL Data Types and Literals*. • Scalar function reference • Aggregate function reference • (*table_expression*) A scalar subquery. • (*interval_expression*)

Examples of Interval Expression Components and Their Processing

The following examples illustrate the components of an interval expression and describe how those components are processed.

Example of *interval_term*

The definition for *interval_term* can be expressed in four forms.

- *interval_factor*
- *interval_term* * *numeric_factor*
- *interval_term* / *numeric_factor*
- *numeric_term* * *interval_factor*

This example uses the second definition.

```
SELECT (INTERVAL '3-07' YEAR TO MONTH) * 4;
```

The *interval_term* in this operation is INTERVAL '3-07' YEAR TO MONTH.

The *numeric_factor* is 4.

The processing involves the following stages:

1 The interval is converted into 43 months as an INTEGER value.
2 The INTEGER value is multiplied by 4, giving the result 172 months.
3 The result is converted to '14-4'.

Example of *numeric_factor*

This example uses a *numeric_factor* with an INTERVAL YEAR TO MONTH typed value.

```
SELECT INTERVAL '10-02' YEAR TO MONTH * 12/5;
```

The *numeric_factor* in this operation is the integer 12.

The processing involves the following stages:

1 The interval is multiplied by 12, giving the result as an interval.
2 The interval result is divided by 5, giving '24-04'.

Note that very different results are obtained by using parentheses to change the order of evaluation as follows.

```
SELECT INTERVAL '10-02' YEAR TO MONTH * (12/5);
```

The *numeric_factor* in this operation is (12/5).

The processing involves the following stages:

1 The *numeric_factor* is computed, giving the result 2.4, which is truncated to 2 because the value is an integer by default.
2 The interval is multiplied by 2, giving '20-04'.

Example of *interval_term / numeric_factor*

The following example uses an *interval_term* value divided by a *numeric_factor* value.

```
SELECT INTERVAL '10-03' YEAR TO MONTH / 3;
```

The *interval_term* is INTERVAL '10-03' YEAR TO MONTH.

The *numeric_factor* is 3.

The processing involves the following stages:

1 The interval value is decomposed into a value of months.

  Ten years and three months evaluate to 123 months.
2 The interval total is divided by the *numeric_factor* 3, giving '3-05'.

The next example is similar to the first except that it shows how truncation is used in integer arithmetic.

```
SELECT INTERVAL '10-02' YEAR TO MONTH / 3;
```

The *interval_term* is INTERVAL '10-02' YEAR TO MONTH.

The *numeric_factor* is 3.

The processing involves the following stages:

1. The interval value is decomposed into a value of months.

   Ten years and two months evaluate to 122 months.

2. The interval total is divided by the *numeric_factor* 3, giving 40.67 months, which is truncated to 40 because the value is an integer.

3. The interval total is converted back to the appropriate format, giving INTERVAL '3-04'.

Example of *numeric_term * interval_primary*

In this format, the value for *numeric_term* can include instances of multiplication and division.

```
SELECT 12/5 * INTERVAL '10-02' YEAR TO MONTH;
```

The *numeric_term* is 12/5.

The *interval_primary* is INTERVAL '10-02' YEAR TO MONTH.

The processing involves the following stages:

1. The *numeric_term* 12/5 is evaluated, giving 2.4, which is truncated to 2 because the value is an integer by default.

2. The *interval_primary* is multiplied by 2, giving '20-04'.

Example of *numeric_term * ± interval_primary*

This example multiplies a negative *interval_primary* by a *numeric_term* and adds the negative result to an *interval_term*.

```
SELECT (RACE_DURATION + (2 * INTERVAL -'30' DAY));
```

The *numeric_term* in this case is the *numeric_primary* 2.

The *interval_primary* is INTERVAL -'30' DAY.

RACE_DURATION is an *interval_term*, with type INTERVAL DAY TO SECOND.

The processing involves the following stages:

1. The *interval_primary* is converted to an exact numeric, or 60 days.

2. The operations indicated in the arithmetic are performed on the operands (which are both numeric at this point), producing an exact numeric result having the appropriate scale and precision.

   In this example, 60 days are subtracted from RACE_DURATION, which is an INTERVAL type of INTERVAL DAY TO SECOND.

3. The numeric result is converted back into the indicated INTERVAL type, DAY TO SECOND.

Example of *interval_expression*

The definition for *interval_expression* can be expressed in three forms.

- *interval_term*
- *interval_expression* + *interval_term*
- (*date_time_expression* - *date_time_term*) *start* TO *end*

This example uses the second definition.

```
SELECT (CAST(INTERVAL '125' MONTH AS INTERVAL YEAR(2) TO MONTH))
+ INTERVAL '12' YEAR;
```

The *interval_expression* is INTERVAL '125' MONTH.

The *interval_term* is INTERVAL '12' YEAR.

The processing involves the following stages:

1. The CAST function converts the *interval_expression* value of 125 months to 10 years and 5 months.
2. The *interval_term* amount of 12 years is added to the *interval_expression* amount, giving 22 years and 5 months.
3. The result is converted to the appropriate data type, which is INTERVAL YEAR(2) TO MONTH, giving '22-05'.

This example uses the third definition for *interval_expression*.

You must ensure that the values for *date_time_expression* and *date_time_term* are comparable.

```
SELECT (TIME '23:59:59.99' - CURRENT_TIME(2)) HOUR(2) TO SECOND(2);
```

The *date_time_expression* is TIME '23:59:59.99'.

The *date_term* is the *date_time_primary* - CURRENT_TIME(2).

The processing involves the following stages:

1. Assume that the current system time is 18:35:37.83.
2. The HOUR(2) TO SECOND(2) time interval 18:35:37.83 is subtracted from the TIME value 23:59:59.99, giving the result '5:24:22.16'.

Here is another example that uses the third definition for *interval_expression* to find the difference in minutes between two TIMESTAMP values. First define a table:

```
CREATE TABLE BillDateTime
(start_time TIMESTAMP(0)
,end_time TIMESTAMP(0));
```

Now, determine the difference in minutes:

```
SELECT (end_time - start_time) MINUTE(4)
FROM BillDateTime;
```

The processing involves the following stages:

1 The start_time TIMESTAMP value is subtracted from the end_time TIMESTAMP value, giving an interval result.
2 The MINUTE(4) specifies an interval unit of minutes with a precision of four digits, which allows for a maximum of 9999 minutes, or approximately one week.

Rules

The following rules apply to Interval expressions.

- Expressions involving intervals are evaluated by converting the operands to integers, evaluating the resulting arithmetic expression, and then converting the result back to the appropriate interval.
- The data type of both an *interval_expression* and an *interval_primary* is INTERVAL.
- An *interval_expression* must contain either year-month interval components or day-time interval components. Mixing of INTERVAL types is not permitted.
- Expressions involving intervals always evaluate to an interval, even if the expressions contain DateTime or Numeric expressions.

IF an interval_expression contains …	THEN the result …
only one component of type INTERVAL	is of the same INTERVAL type.
a single DateTime value or a *start* TO *end* phrase	contains the DateTime fields specified for the DateTime or *start* TO *end* phrase values.
more than one component of type INTERVAL	is of an INTERVAL type including all the DateTime fields of the INTERVAL types of the component fields.

Normalization of Intervals with Multiple Fields

Because of the way the Parser normalizes multiple field INTERVAL values, the defined precision for an INTERVAL value may not be large enough to contain the value once it has been normalized.

For example, inserting a value of '99-12' into a column defined as INTERVAL YEAR(2) TO MONTH causes an overflow error because the Parser normalizes the value to '100-00'. When an attempt is made to insert that value into a column defined to have a 2-digit YEAR field, it fails because it is a 3-digit year.

Here is an example that returns an overflow error because it violates the permissible range values for the type.

First define the table.

```
CREATE TABLE BillDateTime
(column_1 INTERVAL YEAR
,column_2 INTERVAL YEAR(1) TO MONTH
,column_3 INTERVAL YEAR(2) TO MONTH
,column_4 INTERVAL YEAR(3) TO MONTH );
```

Now insert the value INTERVAL '999-12' YEAR TO MONTH using this INSERT statement.

```
INSERT BillDateTime (column_1, column_4)
VALUES ( INTERVAL '40' YEAR, INTERVAL '999-12' YEAR TO MONTH );
```

The result is an overflow error because the valid range for INTERVAL YEAR(3) TO MONTH values is -'999-11' to '999-11'.

You might expect the value '999-12' to work, but it fails because the Parser normalizes it to a value of '1000-00' YEAR TO MONTH. Because the value for year is then four digits, an overflow occurs and the operation fails.

Arithmetic Operators

Operations on ANSI DateTime and Interval values can include the scalar arithmetic operators +, -, *, and /. However, the operators are only valid on specific combinations of DateTime and Interval values.

Arithmetic Operators and Result Types

The following arithmetic operations are permitted for DateTime and Interval data types:

First Value Type	Operator	Second Value Type	Result Type
DateTime	-	DateTime	Interval
DateTime	+	Interval	DateTime
DateTime	-	Interval	DateTime
Interval	+	DateTime	DateTime
Interval	+	Interval	Interval
Interval	-	Interval	Interval
Interval	*	Number	Interval
Interval	/	Number	Interval
Number	*	Interval	Interval

Adding or Subtracting Numbers from DATE

Teradata SQL extends the ANSI SQL:2008 standard to allow the operations of adding or subtracting a number of days from an ANSI DATE value.

Teradata SQL treats the number as an INTERVAL DAY value.

For more information, see "DATE and Integer Arithmetic" on page 172.

Calculating the Difference Between Two DateTime Values

Teradata Database calculates the interval difference between two DATE, TIME or TIMESTAMP values according to the ANSI SQL standard. Units smaller than the unit of the result are ignored when calculating the interval value.

For example, when computing the difference in months for two DATE values, the day values in each of the two operands are ignored. Similarly when computing the difference in hours for two TIMESTAMP values, the minutes and the seconds values of the operands are ignored.

Example 1

The following query calculates the difference in days between the two DATE values.

```
SELECT (DATE '2007-05-10' - DATE '2007-04-28') DAY;
```

The result is the following:

```
(2007-05-10 - 2007-04-28) DAY
-----------------------------
12
```

The following query calculates the difference in months between the two DATE values.

```
SELECT (DATE '2007-05-10' - DATE '2007-04-28') MONTH;
```

The result is the following:

```
(2007-05-10 - 2007-04-28) MONTH
-------------------------------
1
```

There is a difference of 12 days between the two dates, which does not constitute one month. However, Teradata Database ignores the day values during the calculation and only considers the month values, so the result is an interval of one month indicating the difference between April and May.

Example 2: Add Interval to DATE

The following example adds an Interval value to a DateTime value:

```
CREATE TABLE Subscription
(id CHARACTER(13)
,subscribe_date DATE
,subscribe_interval INTERVAL MONTH(4));

INSERT Subscription (subscribe_date, subscribe_interval)
VALUES (CURRENT_DATE, INTERVAL ’24’ MONTH);

SELECT subscribe_date + subscribe_interval FROM Subscription;
```

The result is a DateTime value.

Aggregate Functions and ANSI DateTime and Interval Data Types

DateTime Data Types

The following aggregate functions are valid for ANSI SQL:2008 DateTime types.

<table>
<tr><th>For this function …</th><th>The result is …</th><th>For more information, see …</th></tr>
<tr><td>AVG(arg)</td><td>the type of the argument.</td><td>“AVG” on page 220.</td></tr>
<tr><td>MAX(arg)</td><td rowspan="2">the type of the argument, based on the comparison rules for DateTime types.</td><td>“MAX” on page 242.</td></tr>
<tr><td>MIN(arg)</td><td>“MIN” on page 245.</td></tr>
<tr><td rowspan="2">COUNT(arg)</td><td>INTEGER, if the mode is Teradata.</td><td rowspan="2">“COUNT” on page 226.</td></tr>
<tr><td>DECIMAL(n,0), if the mode is ANSI, where:<table><tr><th>n is …</th><th>if MaxDecimal in DBSControl is …</th></tr><tr><td>15</td><td>0, 15, or 18</td></tr><tr><td>38</td><td>38</td></tr></table></td></tr>
</table>

Interval Data Types

The following aggregate functions are valid for Interval types.

<table>
<tr><th>For this function …</th><th>The result is …</th><th>For more information, see …</th></tr>
<tr><td>AVG(arg)</td><td>the type of the argument.</td><td>“AVG” on page 220.</td></tr>
<tr><td rowspan="2">COUNT(arg)</td><td>INTEGER, if the mode is Teradata.</td><td rowspan="2">“COUNT” on page 226.</td></tr>
<tr><td>DECIMAL(n,0), if the mode is ANSI, where:<table><tr><th>n is …</th><th>if MaxDecimal in DBSControl is …</th></tr><tr><td>15</td><td>0, 15, or 18</td></tr><tr><td>38</td><td>38</td></tr></table></td></tr>
<tr><td>MAX(arg)</td><td rowspan="2">the type of the argument, based on the comparison rules for DateTime types.</td><td>“MAX” on page 242.</td></tr>
<tr><td>MIN(arg)</td><td>“MIN” on page 245.</td></tr>
<tr><td>SUM(arg)</td><td>the type of the argument.</td><td>“SUM” on page 288.</td></tr>
</table>

Scalar Operations and DateTime Functions

DateTime functions are those functions that operate on either DateTime or Interval values and provide a DateTime value as a result.

The supported DateTime functions are:

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- EXTRACT

To avoid any synchronization problems, operations among any of these functions are guaranteed to use identical definitions for DATE, TIME, or TIMESTAMP so that the following are always true:

- CURRENT_DATE = CURRENT_DATE
- CURRENT_TIME = CURRENT_TIME
- CURRENT_TIMESTAMP = CURRENT_TIMESTAMP
- CURRENT_DATE and CURRENT_TIMESTAMP always identify the same DATE
- CURRENT_TIME and CURRENT_TIMESTAMP always identify the same TIME

Example

The following example uses the CURRENT_DATE DateTime function:

```
SELECT INTERVAL '20' YEAR + CURRENT_DATE;
```

Related Topics

For more information on ...	See ...
CURRENT_DATE	"CURRENT_DATE" on page 529
CURRENT_TIME	"CURRENT_TIME" on page 533
CURRENT_TIMESTAMP	"CURRENT_TIMESTAMP" on page 535
EXTRACT	"EXTRACT" on page 180

Teradata Date and Time Expressions

Teradata SQL provides a data type for DATE values and stores TIME values as encoded numbers with type REAL. This is a Teradata extension of the ANSI SQL:2008 standard and its use is strongly deprecated.

Since both DATE and TIME are encoded values, not simple integers or real numbers, arithmetic operations on these values are restricted.

ANSI DATE and TIME values are stored using appropriate DateTime types and have their own set of rules for DateTime assignment and expressions. For information, see "ANSI DateTime and Interval Data Type Assignment Rules" on page 152 and "Scalar Operations on ANSI SQL:2008 DateTime and Interval Values" on page 154.

DATE and Integer Arithmetic

The following arithmetic functions can be performed with date and an integer (INTEGER is interpreted as a number of days):

- `DATE + INTEGER`
- `INTEGER + DATE`
- `DATE - INTEGER`

These expressions are not processed as simple addition or subtraction, but rather as explained in the following process:

1. The encoded date value is converted to an intermediate value which is the number of days since some system-defined fixed date.
2. The integer value is then added or subtracted, forming another value as number of days, since the fixed base date.
3. The result is converted back to a date, valid in the Gregorian calendar.

DATE and Date Arithmetic

The DATE - DATE expression is not processed as a simple subtraction, but rather as explained in the following process:

1. The encoded date values are converted to intermediate values which are each the number of days since a system-defined fixed date.
2. The second of these values is then subtracted from the first, giving the number of days between the two dates.
3. The result is returned as if it were in the ANSI SQL:2008 form INTERVAL DAY, though the value itself is an integer.

Other arithmetic operations on date values may provide results, but those results are not meaningful.

Example

DATE/2 provides an integer result, but the value has no meaning.

There are no simple arithmetic operations that have meaning for time values. The reason is that a time value is simply a real number with time encoded as:

```
(HOUR*10000 + MINUTE*100 + SECOND)
```

where SECOND may include a fractional value.

Scalar Operations on Teradata DATE Values

The operations of addition and subtraction are allowed as follows, where integer values represent the number of days:

Argument 1	Operation	Argument 2	Result
DATE	+	INTEGER	DATE
DATE	-	INTEGER	DATE
INTEGER	+	DATE	DATE
DATE	-	DATE	INTEGER

Adding 90 days, for example, is not identical to adding 3 months, because of the varying number of days in months.

Also, adding multiples of 365 days is not identical to adding years because of leap years.

Note that scalar operations on Teradata DATE expressions are performed using ANSI SQL:2008 data types, so an expression of the type *date_expression* - *numeric_expression* is treated as if the *numeric_expression* component were typed as INTERVAL DAY.

ANSI SQL:2008 DateTime and Interval values have their own set of scalar operations and with the exception of the scalar operations defined here for DATE, do not support the implicit conversions to resolve expressions of mixed data types.

ADD_MONTHS Function

The ADD_MONTHS function provides for adding or subtracting months or years, handling the variable number of days involved.

For details, see "ADD_MONTHS" on page 174.

EXTRACT Function

Use the EXTRACT function to get the year, month, or day from a date. The result has INTEGER data type.

For details, see "EXTRACT" on page 180.

ADD_MONTHS

Purpose

Adds an integer number of months to a DATE or TIMESTAMP expression and normalizes the result.

Date Syntax

```
—— ADD_MONTHS — (date_expression, integer_expression) —
```

FF07D202

Timestamp Syntax

```
—— ADD_MONTHS — (timestamp_expression, integer_expression) —
```

FF07D208

where:

Syntax element ...	Specifies ...
date_expression	one of the following, to which *integer_expression* months are to be added: • A quoted DATE value • A DATE literal • The CURRENT_DATE keyword • The DATE keyword • A UDT that has an implicit cast that casts between the UDT and a character or DATE type. CURRENT_DATE and DATE specify the current system DATE value.
timestamp_expression	one of the following, to which *integer_expression* months are to be added: • A TIMESTAMP literal • The CURRENT_TIMESTAMP keyword • A UDT that has an implicit cast that casts between the UDT and a character or TIMESTAMP type. CURRENT_TIMESTAMP specifies the current system TIMESTAMP value.
integer_expression	the number of integer months to be added to *date_expression* or *timestamp_expression*.

ANSI Compliance

ADD_MONTHS is a Teradata extension to the ANSI SQL:2008 standard.

Rules

ADD_MONTHS observes the following rules:

- If either argument of ADD_MONTHS is NULL, then the result is NULL.
- If the result is not in the range '0000-01-01' to '9999-12-31', then an error is reported.
- Results of an ADD_MONTHS function that are invalid dates are normalized to ensure that all reported dates are valid.

Support for UDTs

IF this argument is a UDT …	THEN Teradata Database performs implicit type conversion if the UDT has an implicit cast that casts between the UDT and any of the following predefined types …
date_expression *timestamp_expression*	• Character • Date • Timestamp
integer_expression	Numeric

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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including ADD_MONTHS, 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 *Utilities*.

For more information on implicit type conversion of UDTs, see "Implicit Type Conversions" on page 577.

Scalar Arithmetic on Months Issues

Consistent handling of a target month having fewer days than the month in the source date is an important issue for scalar arithmetic on month intervals because the concept of a month has no fixed definition.

All scalar function operations on dates use the Gregorian calendar. Peculiarities of the Gregorian calendar ensure that arithmetic operations such as adding 90 days (to represent three months) or 730 days (to represent two years) to a DATE value generally do not provide the desired result. For more information, see "Gregorian Calendar Rules" on page 156.

The ADD_MONTHS function uses an algorithm that lets you add or subtract a number of months to a *date_expression* or *timestamp_expression* and to obtain consistently valid results.

When deciding whether to use the Teradata SQL ADD_MONTHS function or ANSI SQL:2008 DateTime interval arithmetic, you are occasionally faced with choosing between returning a result that is valid, but probably neither desired nor expected, or not returning any result and receiving an error message.

A third option that does not rely on system-defined functions is to use the Teradata Database-defined Calendar view for date arithmetic. For information, see “CALENDAR View” in the *Data Dictionary* book.

Normalization Behavior of ADD_MONTHS

The standard approach to interval month arithmetic is to increment MONTH and YEAR values as appropriate and retain the source value for DAY. This is a problem for the case when the target DAY value is smaller than the source DAY value from the source date.

For example, what approach should be taken to handle the result of adding one MONTH to a source DATE value of ‘1999-01-31’? Using the standard approach, the answer would be ‘1999-02-31’, but February 31 is not a valid date.

The behavior of ADD_MONTHS is equivalent to that of the ANSI SQL:2008 compliant operations DATE ± INTERVAL ‘*n*’ MONTH and TIMESTAMP ± INTERVAL ‘*n*’ MONTH with one important difference.

The difference between these two scalar arithmetic operations is their behavior when a invalid date value is returned by the function.

- ANSI SQL:2008 arithmetic returns an error.
- ADD_MONTHS arithmetic makes normative adjustments and returns a valid date.

Definition of Normalization

The normalization process is explained more formally as follows.

When the DAY field of the source *date_expression* or *timestamp_expression* is greater than the resulting target DAY field, ADD_MONTHS sets DD equal to the last day of the month + *n* to normalize the reported date or timestamp.

Define *date_expression* as ‘YYYY-MM-DD’ for simplicity.

For a given *date_expression*, you can then express the syntax of ADD_MONTHS as follows.

```
ADD_MONTHS('YYYY-MM-DD' , n)
```

Recalling that *n* can be negative, and substituting ‘YYYY-MM-DD’ for *date_expression*, you can redefine ADD_MONTHS in terms of ANSI SQL:2008 dates and intervals as follows.

```
ADD_MONTHS('YYYY-MM-DD', n) = 'YYYY-MM-DD' ± INTERVAL 'n' MONTH
```

The equation is true unless an invalid date such as 1999-09-31 results, in which case the ANSI expression traps the invalid date exception and returns an error.

ADD_MONTHS, on the other hand, processes the exception and returns a valid, though not necessarily expected, date. The algorithm ADD_MONTHS uses to produce its normalized result is as follows, expressed as pseudocode.

```
WHEN
DD > last_day_of_the_month(MM+n)
THEN SET
DD = last_day_of_the_month(MM+n)
```

This property is also true for the date portion of any *timestamp_expression*.

Note that normalization produces valid results for leap years.

Non-Intuitive Results of ADD_MONTHS

Because of the normalization made by ADD_MONTHS, many results of the function are not intuitive, and their inversions are not always symmetrical. For example, compare the results of "Example 5" on page 178 with the results of "Example 7" on page 179.

This is because the function always produces a valid date, but not necessarily an *expected* date. Correctness in the case of interval month arithmetic is a relative term. Any definition is arbitrary and cannot be generalized, so the word 'expected' is a better choice for describing the behavior of ADD_MONTHS.

The following SELECT statements return dates that are both valid and expected:

```
SELECT ADD_MONTHS ('1999-08-15' , 1);
```

This statement returns 1999-09-15.

```
SELECT ADD_MONTHS ('1999-09-30' , -1);
```

This statement returns 1999-08-30.

The following SELECT statement returns a valid date, but its 'correctness' depends on how you choose to define the value 'one month.'

```
SELECT ADD_MONTHS ('1999-08-31' , 1);
```

This statement returns 1999-09-30, because September has only 30 days and the non-normalized answer of 1999-09-31 is not a valid date.

ADD_MONTHS Summarized

ADD_MONTHS returns a new *date_expression* or *timestamp_expression* with YEAR and MONTH fields adjusted to provide a correct date, but a DAY field adjusted only to guarantee a valid date, which might not be a date you expect intuitively.

If this behavior is not acceptable for your application, use ANSI SQL:2008 DateTime interval arithmetic instead. For more information, see "ANSI Interval Expressions" on page 160.

Remember that ADD_MONTHS changes the DAY value of the result *only* when an invalid *date_expression* or *timestamp_expression* would otherwise be reported.

For examples of this behavior, see the example set listed under "Non-Intuitive Examples" on page 178.

Intuitive Examples

"Example 1" through "Example 5" are simple, intuitive examples of the ADD_MONTHS function. All results are both valid and expected.

Example 1

This statement returns the current date plus 13 years.

```
SELECT ADD_MONTHS (CURRENT_DATE, 12*13);
```

Example 2

This statement returns the date 6 months ago.

```
SELECT ADD_MONTHS (CURRENT_DATE, -6);
```

Example 3

This statement returns the current TIMESTAMP plus four months.

```
SELECT ADD_MONTHS (CURRENT_TIMESTAMP, 4);
```

Example 4

This statement returns the TIMESTAMP nine months from January 1, 1999. Note the literal form, which includes the keyword TIMESTAMP.

```
SELECT ADD_MONTHS (TIMESTAMP '1999-01-01 23:59:59', 9);
```

Example 5

This statement adds one month to January 30, 1999.

```
SELECT ADD_MONTHS ('1999-01-30', 1);
```

The result is 1999-02-28.

Non-Intuitive Examples

"Example 6" through "Example 10" illustrate how the results of an ADD_MONTHS function are not always what you might expect them to be when the value for DAY in *date_expression* or the date component of *timestamp_expression* is 29, 30, or 31.

All examples use a *date_expression* for simplicity. In every case, the function behaves as designed.

Example 6

The result of the SELECT statement in this example is a date in February, 1996. The result would be February 31, 1996 if that were a valid date, but because February 31 is *not* a valid date, ADD_MONTHS normalizes the answer.

That answer, because the DAY value in the source date is greater than the last DAY value for the target month, is the last valid DAY value for the *target* month.

```
SELECT ADD_MONTHS ('1995-12-31', 2);
```

The result of this example is 1996-02-29.

Note that 1996 was a leap year. If the interval were 14 months rather than 2, the result would be '1997-02-28'.

Example 7

This statement performs the converse of the ADD_MONTHS function in "Example 5" on page 178.

You might expect it to return '1999-01-30', which is the source date in that example, but it does not.

```
SELECT ADD_MONTHS ('1999-02-28' , -1);
```

ADD_MONTHS returns the result 1999-01-28.

The function performs as designed and this result is not an error, though it might not be what you would expect from reading "Example 5."

Example 8

You might expect the following statement to return '1999-03-31', but it does not.

```
SELECT ADD_MONTHS ('1999-02-28' , 1);
```

ADD_MONTHS returns the result 1999-03-28.

Example 9

You might expect the following statement to return '1999-03-31', but it does not.

```
SELECT ADD_MONTHS ('1999-04-30' , -1);
```

ADD_MONTHS returns the result 1999-03-30.

Example 10

You might expect the following statement to return '1999-05-31', but it does not.

```
SELECT ADD_MONTHS ('1999-04-30' , 1);
```

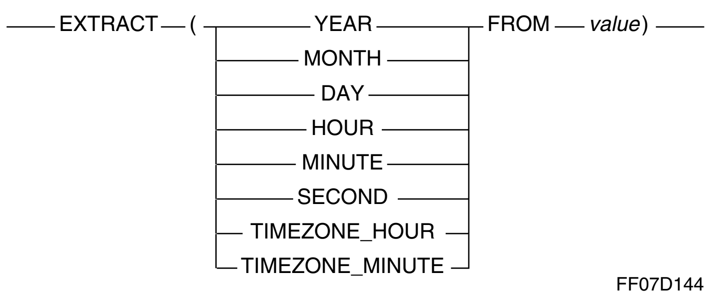
ADD_MONTHS returns the result 1999-05-30.

EXTRACT

Purpose

Extracts a single specified full ANSI SQL:2008 field from any DateTime or Interval value, converting it to an exact numeric value.

Syntax



where:

Syntax element ...	Specifies ...
YEAR	that the integer value for YEAR is to be extracted from the date represented by *value*.
MONTH	that the integer value for MONTH is to be extracted from the date represented by *value*.
DAY	that the integer value for DAY is to be extracted from the date represented by *value*.
HOUR	that the integer value for HOUR is to be extracted from the date represented by *value*.
MINUTE	that the integer value for MINUTE is to be extracted from the date represented by *value*.
TIMEZONE_HOUR	that the integer value for TIMEZONE_HOUR is to be extracted from the date represented by *value*.
TIMEZONE_MINUTE	that the integer value for TIMEZONE_MINUTE is to be extracted from the date represented by *value*.
SECOND	that the integer value for SECOND is to be extracted from the date represented by *value*.
value	an expression that results in a DateTime, Interval, or UDT value.

ANSI Compliance

EXTRACT is partially ANSI SQL:2008 compliant.

ANSI SQL:2008 EXTRACT allows extraction of any field in any DateTime or Interval value. In addition to the ANSI SQL:2008 extract function, Teradata SQL also supports HOUR, MINUTE, or SECOND extracted from a floating point value.

Arguments

IF *value* is ...	THEN ...
a character string literal that represents a date	the string must match the 'YYYY-MM-DD' format.
a character string literal that represents a time	the string must match the 'HH:MI:SS.SSSSSS' format.
a floating point type	*value* must be a time value encoded with the algorithm HOUR * 10000 + MINUTE * 100 + SECOND. Only HOUR, MINUTE, and SECOND can be extracted from a floating point value. Externally created time values can be appropriately encoded and stored in a REAL column to any desired precision if the encoding creates a value representable by REAL without precision loss. Do not store time values as REAL in any new applications. Instead, use the more rigorously defined ANSI SQL:2008 DateTime data types.
a UDT	the UDT must have an implicit cast that casts between the UDT and any of the following predefined types: • Numeric • Character • DateTime 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 *SQL Data Definition Language*. Implicit type conversion of UDTs for system operators and functions, including EXTRACT, 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 *Utilities*. For more information on implicit type conversion of UDTs, see "Implicit Type Conversions" on page 577.
not a character string literal or floating point type or UDT	the expression must evaluate to a DateTime or Interval type.

Results

EXTRACT returns an exact numeric value for ANSI SQL:2008 DateTime values.

EXTRACT returns values adjusted for the appropriate time zone, either the explicit time zone of the argument or the time zone defined for the session in which the EXTRACT is run.

<table>
<tr><th>If you extract ...</th><th>THEN ...</th></tr>
<tr><td>SECOND</td><td><table>
<tr><th>IF value has a seconds fractional precision of ...</th><th>THEN the result is ...</th></tr>
<tr><td>zero</td><td>INTEGER.</td></tr>
<tr><td>greater than zero</td><td>DECIMAL with the scaling as specified for the SECOND field in its data description.</td></tr>
</table></td></tr>
<tr><td>anything else</td><td>the result is INTEGER, with 32 bits of precision.</td></tr>
</table>

If *value* is NULL, the result is NULL.

Example 1

The following example returns the year, as an integer, from the current date.

```
SELECT EXTRACT (YEAR FROM CURRENT_DATE);
```

Example 2

Assuming PurchaseDate is a DATE field, this example returns the month of the *date value* formed by adding 90 days to PurchaseDate as an integer.

```
SELECT EXTRACT (MONTH FROM PurchaseDate+90) FROM SalesTable;
```

Example 3

The following returns 12 as an integer.

```
SELECT EXTRACT (DAY FROM '1996-12-12');
```

Example 4

This example returns an error because the character literal does not evaluate to a valid date.

```
SELECT EXTRACT (DAY FROM '1996-02-30');
```

Example 5

The following returns an error because the character string literal does not match the ANSI SQL:2008 date format.

```
SELECT EXTRACT (DAY FROM '96-02-15');
```

If the argument to EXTRACT is a value of type DATE, the value contained is warranted to be a valid date, for which EXTRACT cannot return an error.

Example 6

The following example relates to non-ANSI DateTime definitions. If the argument is a character literal formatted as a time value, it is converted to REAL and processed. In this example, 59 is returned.

```
SELECT EXTRACT (MINUTE FROM '23:59:17.3');
```

Example 7

This example returns the hour, as an integer, from the current time.

```
SELECT EXTRACT (HOUR FROM CURRENT_TIME);
```

Current time is retrieved as the system value TIME, to the indicated precision.

Example 8

The following example returns the seconds as DECIMAL(8,2). This is based on the fractional seconds precision of 2 for CURRENT_TIME.

```
SELECT EXTRACT (SECOND FROM CURRENT_TIME (2));
```

CHAPTER 7 Period Functions and Operators

This chapter describes the Period functions and operators.

BEGIN

Purpose

Bound function that returns the beginning bound of the Period argument.

Syntax

── BEGIN(*period_value_expression*) ──────

1101A595

where:

Syntax element ...	Specifies ...
period_value_expression	any expression that evaluates to a Period data type.

Return Value

The result data type of the BEGIN function is same as the element type of the Period value expression. If the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is BEGIN(*period_value_expression*).

Error Conditions

If the argument does not have a Period data type, an error is reported.

Example

In the following example, BEGIN is used in the WHERE clause.

```
SELECT * FROM employee WHERE BEGIN(period1) = DATE '2004-06-19';
```

Assume the query is executed on the following table employee where period1 is a PERIOD(DATE) column:

ename	dept	period1
Jones	Sales	('2004-01-02', '2004-01-05')

ename	dept	period1
Adams	Marketing	('2004-06-19', '2005-02-09')
Mary	Development	('2004-06-19', '2005-01-05')
Simon	Sales	('2004-06-22', '2005-01-07')

The result is as follows:

ename	dept	period1
Adams	Marketing	('2004-06-19', '2005-02-09')
Mary	Development	('2004-06-19', '2005-01-05')

END

Purpose

Bound function that returns the ending bound of the Period argument.

Syntax

```
—— END(period_value_expression) ——————
```

1101A596

where:

Syntax element ...	Specifies ...
period_value_expression	any expression that evaluates to a Period data type.

Return Value

The result type of the END function is same as the element type of the Period value expression. If the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is END(*period_value_expression*).

Error Conditions

If an argument of any data type other than a Period data type is passed, an error is reported.

Example

In the following example, END is used in the WHERE clause.

```
SELECT * FROM employee WHERE END(period1) = DATE '2005-01-07';
```

Assume the query is executed on the following table employee with PERIOD(DATE) column period1:

ename	dept	period1
Jones	Sales	('2004-01-02', '2004-01-05')

ename	dept	period1
Adams	Marketing	('2004-06-19', '2005-02-09')
Mary	Development	('2004-06-19', '2005-01-05')
Simon	Sales	('2004-06-22', '2005-01-07')

The result is as follows:

ename	dept	period1
Simon	Sales	('2004-06-22', '2005-01-07')

LAST

Purpose

Bound function that returns the last value of the Period argument (that is, the ending bound minus one granule of the element type of the argument).

Syntax

LAST(*period_value_expression*)

1101A597

where:

Syntax element ...	Specifies ...
period_value_expression	any expression that evaluates to a Period data type.

Return Value

The result type of the LAST function is same as the element type of the Period value expression. If the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the element type of the Period value expression.

The title is LAST(*period_value_expression*).

Error Conditions

If an argument has a data type other than a Period data type, an error is reported.

Example

In the following example, LAST is used in the WHERE clause.

```
SELECT * FROM employee WHERE LAST(period1) = DATE '2004-01-04';
```

Assume the query is executed on the following table employee with PERIOD(DATE) column period1:

ename	dept	period1
Jones	Sales	('2004-01-02', '2004-01-05')

ename	dept	period1
Adams	Marketing	('2004-06-19', '2005-02-09')
Mary	Development	('2004-06-19', '2005-01-05')
Simon	Sales	('2004-06-22', '2005-01-07')

The result is as follows:

ename	dept	period1
Jones	Sales	('2004-01-02', '2004-01-05')

INTERVAL

Purpose

Finds the difference between the ending and beginning bounds of a Period argument and returns this difference as the duration of the argument in terms of a specified interval qualifier.

Syntax

```
—— INTERVAL (period_expression) —— interval_qualifier ——
```

1101A577

where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type. **Note:** Implicit casting to a Period data type is not supported.
interval_qualifier	any interval qualifier appropriate for the argument's element type. The interval qualifiers are as follows: Year-Month intervals: • YEAR • YEAR TO MONTH • MONTH Day-Time Intervals: • DAY • DAY TO HOUR, MINUTE or SECOND • HOUR • HOUR TO MINUTE or SECOND • MINUTE • MINUTE to SECOND • SECOND

Return Value

The result type is the interval data type corresponding to the specified interval qualifier.

The result of the INTERVAL (p) IQ function is the value of (END(p) - BEGIN(p)) IQ, where argument p is a Period expression and IQ is an interval qualifier. The function finds the difference between the argument's ending bound and the beginning bound and returns the resulting difference as an interval value based on the specified interval qualifier.

If the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the interval data type corresponding to the specified interval qualifier.

The title is INTERVAL(*period _expression*) *interval_qualifier*.

Error Conditions

An error may be reported:

- If the argument of the INTERVAL function does not have a Period data type.
- If the argument has a PERIOD(DATE) data type and the interval qualifier is not YEAR, YEAR TO MONTH, MONTH, or DAY.
- If the argument has a PERIOD(TIME(n) [WITH TIME ZONE]) data type and the interval qualifier is not HOUR, HOUR TO MINUTE, HOUR TO SECOND, MINUTE, MINUTE TO SECOND or SECOND.
- If the result of an INTERVAL expression violates the rules specified for the precision of an interval qualifier, an existing error is reported. For example, assume p1 is a PERIOD(TIMESTAMP(0)) expression that has a value of PERIOD '(2006-01-01 12:12:12, 2007-01-01 12:12:12)'. If INTERVAL(p1) DAY is specified, the default precision for the DAY interval qualifier is 2, and, since the result is 365 days which is a three digit value that cannot fit into a DAY(2) interval qualifier, an error is reported.
- If the argument of the INTERVAL function is a period of element type DATE or TIMESTAMP(n) [WITH TIME ZONE] and the ending bound value is UNTIL_CHANGED.

Example

In the following example, INTERVAL is used in a selection list.

```
SELECT INTERVAL (period1) MONTH FROM employee;
```

Assume the query is executed on the following table employee with PERIOD(DATE) column period1:

ename	dept	period1
Jones	Sales	('2004-01-02', '2004-03-05')

The result is as follows:

INTERVAL (period1) MONTH
2

PRIOR

Purpose

Proximity function that returns the preceding value of the argument such that there is one granule of the argument type between the returned value and the argument.

Syntax

```
—— PRIOR (datetime_expression) ——————————————
                                     1101A578
```

where:

Syntax element ...	Specifies ...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.

Return Value

The return data type is the same as that of the argument; that is, a DateTime data type. If the value of the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the argument's data type.

The title is PRIOR(*proximity_argument*).

Error Conditions

If the argument does not have a DateTime data type, an error is reported.

If the result is outside the permissible range of the argument's data type, an error is reported. For example, if PRIOR(DATE '0001-01-01') is specified, an error is reported.

Example

In the following example, PRIOR is used in the WHERE clause.

```
SELECT * FROM employee WHERE PRIOR(END(period1)) = DATE '2004-03-04';
```

Assume the query is executed on the following table employee where period1 is a PERIOD(DATE) column:

ename	dept	period1
Jones	Sales	('2004-01-02', '2004-03-05')
Simon	Sales	?

The result is as follows:

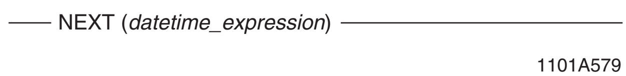
ename	dept	period1
Jones	Sales	('2004-01-02', '2004-03-05')

NEXT

Purpose

Proximity function that returns the succeeding value of the argument such that there is one granule of the argument type between the argument and the returned value.

Syntax



where:

Syntax element ...	Specifies ...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.

Return Value

The return data type is the same as that of the argument (that is, a DateTime data type). If the value of the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the proximity argument's data type.

The title is NEXT(*datetime_expression*).

Error Conditions

If the argument does not have a DateTime data type, an error is reported.

If the result is outside the permissible range of a value for the argument's data type, an error is reported. For example, if NEXT(DATE '9999-12-31') is specified, an error is reported.

Example

In the following example, NEXT is used in the WHERE clause.

```
SELECT * FROM employee WHERE NEXT(END(period1)) = DATE '2004-03-06';
```

Assume the query is executed on the following table employee where period1 is a PERIOD(DATE) column:

ename	dept	period1
Jones	Sales	('2004-01-02', '2004-03-05')
Simon	Sales	?

The result is as follows:

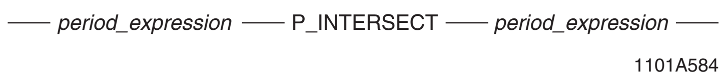
ename	dept	period1
Jones	Sales	('2004-01-02', '2004-03-05')

P_INTERSECT

Purpose

Operator that returns the portion of the Period expressions that is common between the Period expressions if they overlap. If the Period expressions do not overlap, or if either Period expression is NULL, P_INTERSECT returns NULL.

Syntax



where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type. **Note:** The Period expressions specified must be comparable. Implicit casting to a Period data type is not supported.

Return Value

If the Period expressions do not overlap, the result is NULL. If either Period expression is NULL, the result is NULL. Otherwise, the result has a Period data type that is comparable to the Period expressions.

If the Period expressions have PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) data types but different precisions, the result is a Period value of the higher precision data type. If neither Period expression has a time zone, the resulting period does not have a time zone; otherwise, the resulting period has a time zone and the value of the time zone in the result is determined using the following rules:

- If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the Period expression as defined by the Period value constructor that follows.
- If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming *p1* and *p2* are Period expressions and the result element type as determined above is *rt*, the result of p1 P_INTERSECT p2 is as follows if p1 OVERLAPS p2 is TRUE:

```
PERIOD(
```

```
CASE WHEN CAST(BEGIN(p1) AS rt) >= CAST(BEGIN(p2) AS rt)
   THEN CAST(BEGIN(p1) AS rt)
   ELSE CAST(BEGIN(p2) AS rt) END,
CASE WHEN CAST(END(p1) AS rt) <= CAST(END(p2) AS rt)
   THEN CAST(END(p1) AS rt)
   ELSE CAST(END(p2) AS rt) END)
```

Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the P_INTERSECT operation is performed if they overlap.

Format and Title

The format is the default format for the resulting Period data type.

The title is *period_expression* P-INTERSECT *period_expression*.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

In the following example, the P_INTERSECT operator is used in the selection list.

```
SELECT period2 P_INTERSECT period1 FROM employee WHERE ename = 'Adams';
```

Assume the query is executed on the following table *employee* where *period1* is a PERIOD(TIMESTAMP(1)) column and *period2* is a PERIOD(TIMESTAMP(0)) column:

ename	period1	period2
Adams	('2005-02-03 10:10:10.1', '2007-02-03 10:10:10.1')	('2004-02-03 10:10:10', '2006-02-03 10:10:10')
Paul	('2004-02-03 10:10:10.1', '2006-02-03 10:10:10.1')	('2005-02-03 10:10:10', '2007-02-03 10:10:10')
James	('2004-03-03 10:10:10.1', '2006-01-03 10:10:10.1')	('2004-02-03 10:10:10', '2006-02-03 10:10:10')
Mary	('2007-04-02 10:10:10.1', '2008-01-03 10:10:10.1')	('2005-02-03 10:10:10', '2006-02-03 10:10:10')

The result is as follows:

(period2 P_INTERSECT period1)
('2005-02-03 10:10:10.1', '2006-02-03 10:10:10.0')

LDIFF

Purpose

Operator that returns the portion of the first Period expression that exists before the beginning of the second Period expression when the Period expressions overlap. When the Period expressions overlap but there is no portion of the first Period expression before the beginning of the second Period expression or the Period expressions do not overlap, LDIFF returns NULL. If either Period expression is NULL, LDIFF returns NULL.

Syntax

```
—— period_expression —— LDIFF —— period_expression ——
```

1101A592

where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type. **Note:** The Period expressions specified must be comparable. Implicit casting to a Period data type is not supported.

Return Value

Assuming p1 and p2 are comparable Period expressions, p1 LIDFF p2 returns PERIOD(BEGIN(p1), BEGIN(p2)) if p1 OVERLAPS p2 is TRUE and BEGIN(p1) is less than BEGIN(p2). If either Period expression is NULL, p1 OVERLAPS p2 is FALSE, or BEGIN(p1) is not less than BEGIN(p2), the result is NULL.

If the Period expressions have PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) data types but have different precisions, the result has the higher of the two precisions. If one of the Period expressions contains time zones and the other does not, the result contains a time zone for each element. The result time zones are evaluated using the following rules:

- If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the expressions as defined by the Period value constructor that follows.
- If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming p1 and p2 are Period expressions and the result element type as determined above is rt, the result of p1 LDIFF p2 is as follows if p1 OVERLAPS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(BEGIN(p1) AS rt) < CAST(BEGIN(p2) AS rt)
     THEN CAST(BEGIN(p1) AS rt)
     ELSE NULL END,
  CASE WHEN CAST(BEGIN(p1) AS rt) < CAST(BEGIN(p2) AS rt)
     THEN CAST(BEGIN(p2) AS rt)
     ELSE NULL END)
```

Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the LDIFF operation is performed if they overlap.

Format and Title

The format is the default format for the resulting Period data type.

The title is *period_expression* LDIFF *period_expression*.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

In the following example, the LDIFF operator is used to find the left difference of the first Period expression with the second Period expression.

```
SELECT ename, period2 LDIFF period1 FROM employee;
```

Assume the query is executed on the following table employee where period1 and period2 are PERIOD(DATE) columns:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2002-03-05', '2004-10-07')
Randy	('2006-01-02', '2007-03-05')	('2004-03-07', '2005-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is as follows:

ename	(period2 LDIFF period1)
Adams	?
Mary	('2005-02-03', '2005-04-02')
Jones	('2002-03-05', '2004-01-02')

ename	(period2 LDIFF period1)
Randy	?
Simon	?

RDIFF

Purpose

Operator that returns the portion of the first Period expression that exists from the end of the second Period expression when the Period expressions overlap. When the Period expressions overlap but there is no portion of the first Period expression from the end of the second Period expression or if the Period expressions do not overlap, RDIFF returns NULL. If either Period expression is NULL, RDIFF returns NULL.

Syntax

```
—— period_expression —— RDIFF —— period_expression ——
```

1101A593

where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type. **Note:** The Period expressions specified must be comparable. Implicit casting to a Period data type is not supported.

Return Value

Assuming p1 and p2 are comparable Period expressions, p1 RDIFF p2 returns PERIOD(END(p2), END(p1)) if p1 OVERLAPS p2 is TRUE and END(p1) is greater than END(p2). If either Period expression is NULL, p1 OVERLAPS p2 is FALSE, or END(p1) is not greater than END(p2), the result is NULL.

If the Period expressions have PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) data types but have different precisions, the result has the higher of the two precisions. If one of the Period expressions contains time zones and the other does not, the result contains a time zone for each element. The result time zones are evaluated using the following rules:

- If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the Period expressions as defined by the Period value constructor that follows.
- If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming p1 and p2 are Period expressions and the result element type as determined above is rt, the result of p1 RDIFF p2 is as follows if p1 OVERLAPS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(END(p1) AS rt) > CAST(END(p2) AS rt)
     THEN CAST(END(p2) AS rt)
     ELSE NULL END,
  CASE WHEN CAST(END(p1) AS rt) > CAST(END(p2) AS rt)
     THEN CAST(END(p1) AS rt)
     ELSE NULL END)
```

Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the RDIFF operation is performed if they overlap.

Format and Title

The format is the default format for the resulting Period data type.

The title is *period_expression* RDIFF *period_expression*.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

In the following example, the RDIFF operator is used to find the right difference of the first Period expression with the second Period expression.

```
SELECT ename, period2 RDIFF period1 FROM employee;
```

Assume the query is executed on the following table employee where period1 and period2 are PERIOD(DATE) columns:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2001-01-02', '2003-03-05')	('2002-03-05', '2004-10-07')
Randy	('2006-01-02', '2007-03-05')	('2004-03-07', '2005-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is as follows:

ename	(period2 RDIFF period1)
Adams	?
Mary	('2006-01-03', '2006-02-03')
Jones	('2003-03-05', '2004-10-07')

ename	(period2 RDIFF period1)
Randy	?
Simon	?

P_NORMALIZE

Purpose

Operator that returns a Period value that is the combination of the two Period expressions if the Period expressions overlap or meet. If the Period expressions neither meet nor overlap, P_NORMALIZE returns NULL. If either Period expression is NULL, P_NORMALIZE returns NULL.

Syntax

```
—— period_expression —— P_NORMALIZE —— period_expression ——
```

1101A594

where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type. **Note:** The Period expressions specified must be comparable. Implicit casting to a Period data type is not supported.

Return Value

Assuming p1 and p2 are comparable Period expressions and ((BEGIN(p1) >= BEGIN(p2) AND BEGIN(p1) <= END(p2)) OR (BEGIN(p2) >= BEGIN(p1) AND BEGIN(p2) <= END(p1))) is TRUE, p1 P_NORMALIZE p2 returns PERIOD(minimum(BEGIN(p1), BEGIN(p2)), maximum(END(p1), END(p2))). If either Period expression is NULL or ((BEGIN(p1) >= BEGIN(p2) AND BEGIN(p1) <= END(p2)) OR (BEGIN(p2) >= BEGIN(p1) AND BEGIN(p2) <= END(p1))) is FALSE, the result is NULL. Note that the P_NORMALIZE operator returns a Period value if the Period expressions satisfy the MEETS or OVERLAPS condition.

If the Period expressions have PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) data type but have different precisions, the result has the higher of the two precisions. If one of the Period expressions contains a time zone, the result contains a time zone for each element. The result time zones are determined using the following rules:

- If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the Period expressions as defined by the Period value constructor that follows.

- If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming p1 and p2 are Period expressions and the result element type as determined above is rt, the result of p1 P_NORMALIZE p2 is as follows if p1 OVERLAPS p2 OR p1 MEETS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(BEGIN(p1) AS rt) <= CAST(BEGIN(p2) AS rt)
     THEN CAST(BEGIN(p1) AS rt)
     ELSE CAST(BEGIN(p2) AS rt) END,
  CASE WHEN CAST(END(p1) AS rt) >= CAST(END(p2) AS rt)
     THEN CAST(END(p1) AS rt)
     ELSE CAST(END(p2) AS rt) END)
```

Internally, Period values are saved in UTC and the OVERLAPS or MEETS operator is evaluated using these UTC represented formats and the P_NORMALIZE operation is performed if they overlap or meet.

Format and Title

The format is the default format for the resulting Period data type.

The title is *period_expression* P-NORMALIZE *period_expression*.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

In the following example, the P_NORMALIZE operator is used to collapse two Period columns.

```
SELECT period2 P_NORMALIZE period1 FROM employee WHERE ename = 'Adams';
```

Assume the query is executed on the following table employee where period1 is PERIOD(TIMESTAMP(1)) column and period2 is PERIOD(TIMESTAMP(0)) column:

ename	period1	period2
Adams	('2005-02-03 10:10:10.1', '2007-02-03 10:10:10.1')	('2004-02-03 10:10:10', '2006-02-03 10:10:10')
Paul	('2004-02-03 10:10:10.1', '2006-02-03 10:10:10.1')	('2005-02-03 10:10:10', '2007-02-03 10:10:10')
James	('2004-03-03 10:10:10.1', '2006-01-03 10:10:10.1')	('2004-02-03 10:10:10', '2006-02-03 10:10:10')
Mary	('2007-04-02 10:10:10.1', '2008-01-03 10:10:10.1')	('2005-02-03 10:10:10', '2006-02-03 10:10:10')

The result is as follows:

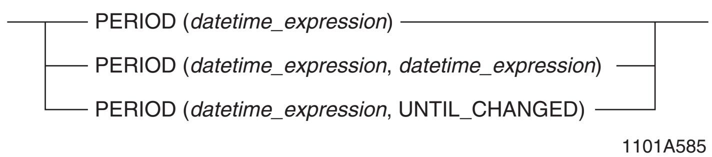
period2 P_NORMALIZE period1
('2004-02-03 10:10:10.0', '2007-02-03 10:10:10.1')

Period Value Constructor

Purpose

Initializes an instance of a Period data type.

Syntax



where:

Syntax element ...	Specifies ...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
UNTIL_CHANGED	a DATE or TIMESTAMP value that is considered to be forever or until it is changed. For PERIOD(DATE) types, UNTIL_CHANGED has a value of DATE '9999-12-31' and for PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) types, UNTIL_CHANGED has a value of TIMESTAMP '9999-12-31 23:59:59.999999 00:00'(with the precision truncated to the precision of the beginning bound and the time zone omitted if the beginning bound does not have a time zone).

Usage Rules

The following rules apply to the Period Value Constructor function:

1. The beginning bound must have a DateTime data type and, if an ending bound is specified, the data types of the beginning and ending bounds must be comparable. Otherwise, an error is reported.
2. If only the beginning bound is specified, the result ending bound is the beginning bound plus one granule of the result element type. If the result ending bound exceeds or becomes equal to the maximum allowed DATE or TIMESTAMP value for result data type of PERIOD(DATE) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), respectively, an error is reported.
3. If the result beginning bound is greater than or equal to the result ending bound, an error is reported.

4 If the beginning or ending bound is NULL, or both the bounds are NULL, the result is NULL.

5 If the beginning bound is NULL and the ending bound is UNTIL_CHANGED, the result is NULL.

6 If the beginning and ending bounds are NULL or if the beginning bound is NULL and the ending bound is UNTIL_CHANGED, then the type of the period defaults to PERIOD(TIMESTAMP(0)).

7 If an ending bound is specified as a value expression and the beginning bound and ending bound have different precisions, the result precision is the higher of the two precisions. Otherwise, the result precision is the precision of the beginning bound.

8 If the beginning bound and/or the ending bound include a time zone value (if specified as a value expression and not as UNTIL_CHANGED), the result data type is WITH TIME ZONE. If only one of them includes a time zone value, the time zone field of the other is set to the current session time zone displacement. If both include time zone values, the result bounds include the corresponding time zone value.

9 The result Period data type has an element type that is the same as the DateTime data type of the beginning bound except with the precision and time zone as defined above.

10 The ending bound where the beginning bound's data type is DATE or TIMESTAMP can be set to UNTIL_CHANGED. UNTIL_CHANGED sets the result ending element to a maximum DATE or TIMESTAMP value depending on the beginning bound's data type. If the beginning bound's data type is TIMESTAMP(n) WITH TIME ZONE, the result ending element is set to the maximum TIMESTAMP(n) WITH TIME ZONE value at UTC (that is, the time zone displacement for the ending bound is INTERVAL '00:00' HOUR TO MINUTE). If UNTIL_CHANGED is specified for the ending bound and the beginning bound's data type is TIME(n) [WITH TIME ZONE], an error is reported. If UNTIL_CHANGED is specified for the beginning bound, an error is reported.

11 The handling of leap seconds for Period data types with TIME and TIMESTAMP element types is as follows. If the value for the beginning or ending bound contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the result precision. During this process, if the beginning and ending bounds are the same, an error is reported.

Example

In the following example, assume t1 is a table with an INTEGER column c1 and a PERIOD(DATE) column c2 and t2 is a table with an INTEGER column a and two DATE columns b and c.

This example shows the Period value constructor used in two INSERT statements.

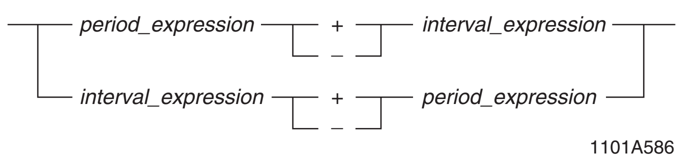
```
INSERT INTO t1
  VALUES (1, PERIOD(DATE '2005-02-03', DATE '2006-02-04'));
INSERT INTO t1 SELECT a, PERIOD(b, c) FROM t2;
```

Arithmetic Operators

Purpose

Adds or subtracts an Interval value to or from a Period value, or adds a Period value to an Interval value.

Syntax



where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type.
interval_expression	an expression that evaluates to an INTERVAL data type. For information on INTERVAL data types, see *SQL Data Types and Literals*.

Usage Notes

Assuming that p is a Period expression of element type DATE or TIMESTAMP and v is an Interval value expression:

- p + v and v + p are both equivalent to:

  ```
  PERIOD(BEGIN(p) + v, CASE WHEN END(p) IS UNTIL_CHANGED THEN END(p)
  ELSE (END(p) + v) END)
  ```

- p - v is equivalent to:

  ```
  PERIOD(BEGIN (p) - v, CASE WHEN END(p) IS UNTIL_CHANGED THEN END(p)
  ELSE (END(p) - v) END)
  ```

Assuming that p is a Period expression of element type TIME and v is an interval value expression:

- p + v and v + p are both equivalent to:

  ```
  PERIOD(BEGIN(p) + v, END(p) + v)
  ```

- p - v is equivalent to:

  ```
  PERIOD(BEGIN (p) - v, END(p) - v)
  ```

Usage Rules

The following rules apply to arithmetic operators and Period data types:

1. The interval value expression must be a valid interval expression and must follow the rules of an Interval expression (see "ANSI Interval Expressions" on page 160). Otherwise, an error is reported. For example, the interval expression (DATE '2006-02-03' - DATE '2005-02-03') DAY, results in a value of 365 days which cannot fit into the default precision 2 of the interval qualifier DAY; therefore, an error is reported.
2. The period arithmetic operations of adding or subtracting an Interval to or from a period or adding a period to an Interval follow the rules of DateTime expressions. Otherwise, errors are reported. See "ANSI DateTime Expressions" on page 155 for details on DateTime expression rules.
3. An interval value expression can be subtracted from a Period expression but not vice versa. If a period expression is subtracted from an interval value expression, an error is reported.
4. For a Period expression with an element type of TIME, if the Period arithmetic operation results in a beginning bound less than the ending bound, an error is reported.
5. For a period of element type DATE or TIMESTAMP, if the ending bound is UNTIL_CHANGED, the ending bound in the result ending bound is UNTIL_CHANGED. If the ending bound is not UNTIL_CHANGED and the ending bound in the result evaluates to an UNTIL_CHANGED value, an error is reported.
6. For a period arithmetic operation, one of the operands must be an INTERVAL data type. Otherwise, an error is reported.

CHAPTER 8 Aggregate Functions

This chapter describes the following SQL aggregate functions.

For window aggregate functions and their Teradata-specific equivalents, see Chapter 9: "Ordered Analytical Functions."

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 Constant Expressions in the Select List

Constant expressions in the select list may optionally appear in the GROUP BY clause. For example, the following statement uses an aggregate function and a constant 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 constant expression.

<table>
<tr><th>IF the constant expression …</th><th>THEN the result of the constant expression in the query result is …</th></tr>
<tr><td>does not contain a column reference</td><td rowspan="2">the value of the constant 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)
FROM Sales_Table;

Count(*) Substring(Current Time(0) From 1 For 8)
-------- ---------------------------------------
       0 09:01:43</td></tr>
<tr><td>is a non-deterministic function, such as RANDOM</td></tr>
<tr><td>contains a column reference</td><td rowspan="2">NULL.
Here is an example:
SELECT COUNT(*), UDF_CALC(1,2)
FROM Sales_Table;

   Count(*) UDF_CALC(1,2)
----------- -------------
          0   ?</td></tr>
<tr><td>is a UDF</td></tr>
</table>

Nesting Aggregates

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

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

But 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;
```

For details on aggregate window functions, see Chapter 9: "Ordered Analytical Functions."

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
aggregate_operator(*expression*) … GROUP BY …	No Record Found
aggregate_operator(*expression*) … HAVING …	

Aggregates and Nulls

Aggregates (with the exception of COUNT(*)) ignore nulls[1] in all computations.

This behavior 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 2" on page 228 and "Example 3" on page 228. Note that the aggregates are behaving exactly as they should—the results are *not* mathematically anomalous.

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

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) + 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 Teradata Database 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.

For more information on potential problems associated with floating point values in computations, see *SQL Data Types and Literals*.

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:

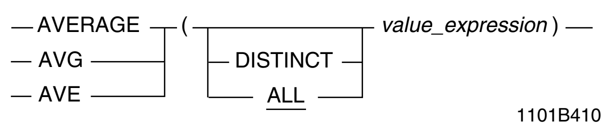
```
SELECT SUM(DISTINCT x), SUM(DISTINCT y) FROM XYTable;
```

AVG

Purpose

Returns the arithmetic average of all values in the specified expression for each row in the group.

Syntax



where:

Syntax element …	Specifies …
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the average computation for the group. This is the default.
DISTINCT	that null and duplicate values specified by *value_expression* are eliminated from the average computation for the group.
value_expression	a constant or column expression for which an average is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

AVG is ANSI SQL:2008 compliant.

AVERAGE and AVE are Teradata extensions to the ANSI standard.

Result Type and Attributes

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

Attribute	Value
Data Type	REAL
Title	Average(x)

Attribute	Value	
Format	**IF the operand is …**	**THEN the format is the …**
	• numeric • date • interval	same format as x.
	character	default format for FLOAT.
	UDT	format for the data type to which the UDT is implicitly cast.

For an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including AVG, 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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

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))
```

Restrictions

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

AVG is valid only for numeric data.

Nulls are not included in the result computation. For more information, see *SQL Fundamentals* and "Aggregates and Nulls" on page 217.

Example

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
```

AVG Window Function

For the AVG window function that computes a group, cumulative, or moving average, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

CORR

Purpose

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

Syntax

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

1101B217

where:

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>value_expression_2</td><td rowspan="2">a numeric expression to be correlated with a second numeric expression.
The expressions cannot contain any ordered analytical or aggregate functions.</td></tr>
<tr><td>value_expression_1</td></tr>
</table>

ANSI Compliance

CORR is ANSI SQL:2008 compliant.

Definition

The 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.

IF the correlation coefficient has this value ...	THEN the association between the variables ...
NULL	cannot be measured because there are no non-null data point pairs in the data used for the computation.

Computation

The equation for computing CORR is defined as follows:

$$CORR = \frac{COVAR\_SAMP(x,y)}{STDDEV\_SAMP(x)STDDEV\_SAMP(y)}$$

where:

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

Division by zero results in NULL rather than an error.

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 *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

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 Chapter 9: "Ordered Analytical Functions."

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

Example

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.

CORR Window Function

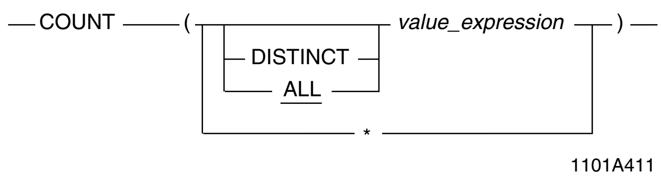
For the CORR window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

COUNT

Purpose

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

Syntax



where:

Syntax element …	Specifies …
ALL	that all non-null values of *value_expression*, including duplicates, are included in the total count. This is the default.
DISTINCT	that a *value_expression* that evaluates to a null value or to a duplicate value does not contribute to the total count.
value_expression	a constant or column expression for which the total count is computed. The expression cannot contain any ordered analytical or aggregate functions.
*	to count all rows in the group of rows on which COUNT operates.

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.
COUNT(*)	in the group of rows on which COUNT operates.

For COUNT functions that return the group, cumulative, or moving count, see “Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)” on page 320.

COUNT is valid for any data type.

With the exception of COUNT(*), the computation does not include nulls. For more information, see *SQL Fundamentals* and "Aggregates and Nulls" on page 217.

For an example that uses COUNT(*) and nulls, see "Example 2" on page 228.

Result Type and Attributes

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

<table>
<tr><th>Mode</th><th>Data Type</th></tr>
<tr><td>ANSI</td><td><table><tr><th>IF MaxDecimal in DBSControl is …</th><th>THEN the result type is …</th></tr><tr><td>0, 15, or 18</td><td>DECIMAL(15,0)</td></tr><tr><td>38</td><td>DECIMAL(38,0)</td></tr></table></td></tr>
<tr><td>Teradata</td><td>INTEGER</td></tr>
</table>

ANSI mode uses DECIMAL because tables frequently have a cardinality exceeding the range of INTEGER.

Teradata mode uses INTEGER to avoid regression problems.

When in Teradata mode, 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;
```

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

Operation	Format	Title
COUNT(x)	Default format for result data type	Count(x)
COUNT(*)	Default format for result data type	Count(*)

For information on data type default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example 1

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 2

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 3

If you ran the report in Example 2 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 2 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 4

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 5

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 6

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 7

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
```

COVAR_POP

Purpose

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

Syntax

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

1101B216

where:

Syntax element …	Specifies …
value_expression_2	a numeric expression to be paired with a second numeric expression to determine their covariance. The expressions cannot contain any ordered analytical or aggregate functions.
value_expression_1	

ANSI Compliance

COVAR_POP is ANSI SQL:2008 compliant.

Definition

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.

Combination With Other Functions

COVAR_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 Chapter 9: "Ordered Analytical Functions."

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

Computation

The equation for computing COVAR_POP is defined as follows:

$$\mathrm{COVAR\_POP} = \frac{\mathrm{SUM}((x - \mathrm{AVG}(x))(y - \mathrm{AVG}(y)))}{\mathrm{COUNT}(x)}$$

where:

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

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td><table>
<tr><th>IF the operand is ...</th><th>THEN the format is ...</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table></td><td>COVAR_POP(y,x)</td></tr>
</table>

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 *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

COVAR_POP Window Function

For the COVAR_POP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

COVAR_SAMP

Purpose

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

Syntax

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

1101A456

where:

Syntax element ...	Specifies ...
value_expression_2	a numeric expression to be paired with a second numeric expression to determine their covariance. The expressions cannot contain any ordered analytical or aggregate functions.
value_expression_1	

ANSI Compliance

COVAR_SAMP is ANSI SQL:2008 compliant.

Definition

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.

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 Chapter 9: “Ordered Analytical Functions.”

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

Computation

The equation for computing COVAR_SAMP is defined as follows:

$$\mathrm{COVAR\_SAMP} = \frac{\mathrm{SUM}((x - \mathrm{AVG}(x))(y - \mathrm{AVG}(y)))}{\mathrm{COUNT}(x) - 1}$$

where:

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

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td><table>
<tr><th>IF the operand is …</th><th>THEN the format is …</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table></td><td>COVAR_SAMP(y,x)</td></tr>
</table>

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 *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

COVAR_SAMP Window Function

For the COVAR_SAMP window function that performs a group, cumulative, or moving computation, see “Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)” on page 320.

Example

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 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

Purpose

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

Syntax

—— GROUPING —— (*expression*) ——

1101A461

where:

Syntax element ...	Specifies ...
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

GROUPING is ANSI SQL:2008 compliant.

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.

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)

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Result Value

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

Example

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 Topics

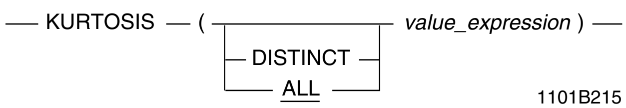
For more information on GROUP BY, GROUPING SETS, ROLLUP, and CUBE, see *SQL Data Manipulation Language*.

KURTOSIS

Purpose

Returns the kurtosis of the distribution of *value_expression*.

Syntax



where:

Syntax element …	Specifies …
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates specified by *value_expression* from the computation.
value_expression	a constant or column expression for which the kurtosis of the distribution of its values is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

KURTOSIS is a Teradata extension to the ANSI SQL:2008 standard.

Definition

Kurtosis is the fourth moment of a distribution. It is a measure of the relative peakedness or flatness compared with the normal, Gaussian distribution.

The normal distribution has a kurtosis of 0.

Positive kurtosis indicates a relative peakedness of the distribution, while negative kurtosis indicates a relative flatness.

Result Type and Attributes

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

Data Type	Format	Title
REAL	Default format of the REAL data type	Kurtosis(x)

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

Computation

The equation for computing KURTOSIS is defined as follows:

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

where:

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

Conditions That Produce a NULL Return Value

The following conditions produce a null return value:

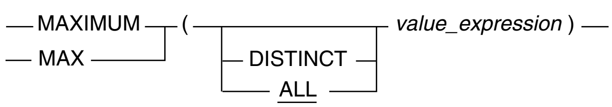
- Fewer than four non-null data points in the data used for the computation
- STDDEV_SAMP(x) = 0
- Division by zero

MAX

Purpose

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

Syntax



1101B412

where:

Syntax element …	Specifies …
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the maximum value computation for the group. This is the default.
DISTINCT	that duplicate and non-null values specified by *value_expression* are eliminated from the maximum value computation for the group.
value_expression	a constant or column expression for which the maximum value is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

MAX is ANSI SQL:2008 compliant.

MAXIMUM is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

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

<table>
<tr><th>Attribute</th><th>Value</th></tr>
<tr><td>Data Type</td><td>
<table>
<tr><th>IF operand x is …</th><th>THEN the result data type is the data type …</th></tr>
<tr><td>not a UDT</td><td>of operand x.</td></tr>
<tr><td>a UDT</td><td>to which the UDT is implicitly cast.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Attribute</th><th>Value</th></tr>
<tr><td>Format</td><td><table><tr><th>IF operand x is …</th><th>THEN the result format is the format of …</th></tr><tr><td>not a UDT</td><td>operand x.</td></tr><tr><td>a UDT</td><td>the type to which the UDT is implicitly cast.</td></tr></table></td></tr>
<tr><td>Title</td><td>Maximum(x)</td></tr>
</table>

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

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 *SQL Fundamentals* and "Aggregates and Nulls" on page 217.

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.

MAX Window Function

For the MAX window function that computes a group, cumulative, or moving maximum value, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example 1: CHARACTER Data

The following SELECT returns the immediately following result.

```
SELECT MAX(Name)
FROM Employee;

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

Example 2: Column Expressions

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

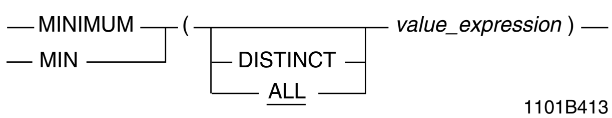
```
SELECT MAX(CostOfSales) AS m, ProdID
FROM Inventory
GROUP BY ProdID
ORDER BY m DESC;

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

MIN

Purpose

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

Syntax



where:

Syntax element …	Specifies …
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the minimum value computation for the group. This is the default.
DISTINCT	that duplicate and non-null values specified by *value_expression* are eliminated from the minimum value computation for the group.
value_expression	a constant or column expression for which the minimum value is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

MIN is ANSI SQL:2008 compliant.

MINIMUM is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

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

<table>
<tr><th>Attribute</th><th>Value</th></tr>
<tr><td>Data Type</td><td><table><tr><th>IF operand x is …</th><th>THEN the result data type is the data type …</th></tr><tr><td>not a UDT</td><td>of operand x.</td></tr><tr><td>a UDT</td><td>to which the UDT is implicitly cast.</td></tr></table></td></tr>
<tr><td>Title</td><td>Minimum(x)</td></tr>
</table>

Attribute	Value
Format	IF operand x is … → THEN the result format is the format of … not a UDT → operand x. a UDT → the type to which the UDT is implicitly cast.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

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 *SQL Fundamentals* and “Aggregates and Nulls” on page 217.

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.

MIN Window Function

For the MIN window function that computes a group, cumulative, or moving minimum value, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example 1: MINIMUM Used With CHARACTER Data

The following SELECT returns the immediately following result.

```
SELECT MINIMUM(Name)
FROM Employee;

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

Example 2: 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
```

REGR_AVGX

Purpose

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

Syntax

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

1101B414

where:

Syntax element ...	Specifies ...
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

REGR_AVGX is ANSI SQL:2008 compliant.

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 Chapter 9: “Ordered Analytical Functions.”

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

Computation

The equation for computing REGR_AVGX is:

$$REGR\_AVGX = \frac{SUM(x)}{n}$$

where:

This variable ...	Represents ...
x	*independent_variable_expression* x is the independent, or predictor, variable expression.
n	COUNT(x)

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is ...</th><th>THEN the format is ...</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>REGR_AVGX(y,x)</td></tr>
</table>

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 *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

REGR_AVGX Window Function

For the REGR_AVGX window function that performs a group, cumulative, or moving computation, see “Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)” on page 320.

Example

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

Purpose

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

Syntax

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

1101B415

where:

Syntax element ...	Specifies ...
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

REGR_AVGY is ANSI SQL:2008 compliant.

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 Chapter 9: "Ordered Analytical Functions."

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

Computation

The equation for computing REGR_AVGY is:

$$\mathrm{REGR\_AVGY} = \frac{\mathrm{SUM}(y)}{n}$$

where:

This variable ...	Represents ...
y	*dependent_variable_expression* y is the dependent, or response, variable expression.
n	COUNT(y)

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is ...</th><th>THEN the format is ...</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>REGR_AVGY(y,x)</td></tr>
</table>

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 *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

REGR_AVGY Window Function

For the REGR_AVGY window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example

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
```

REGR_COUNT

Purpose

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

Syntax

— REGR_COUNT — (*dependent_variable_expression, independent_variable_expression*) —

1101B416

where:

Syntax element …	Specifies …
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

REGR_COUNT is ANSI SQL:2008 compliant.

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 Chapter 9: "Ordered Analytical Functions."

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

Result Type and Attributes

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

<table>
<tr><th>Mode</th><th>Data Type</th></tr>
<tr><td>ANSI</td><td><table>
<tr><th>IF MaxDecimal in DBSControl is ...</th><th>THEN the result type is ...</th></tr>
<tr><td>0, 15, or 18</td><td>DECIMAL(15,0)</td></tr>
<tr><td>38</td><td>DECIMAL(38,0)</td></tr>
</table></td></tr>
<tr><td>Teradata</td><td>INTEGER</td></tr>
</table>

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;
```

The following table lists the default format and title for the result of REGR_COUNT(y, x).

<table>
<tr><th>Format</th><th>Title</th></tr>
<tr><td><table>
<tr><th>IF operand y is ...</th><th>THEN the format is ...</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>numeric</td><td>the same format as y.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table></td><td>REGR_COUNT(y,x)</td></tr>
</table>

For information on data type default formats, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

REGR_COUNT Window Function

For the REGR_COUNT window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example

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;

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

REGR_INTERCEPT

Purpose

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

Syntax

── REGR_INTERCEPT ── (*dependent_variable_expression, independent_variable_expression*) ──

1101B417

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_INTERCEPT is ANSI SQL:2008 compliant.

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.

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 Chapter 9: "Ordered Analytical Functions."

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

Computation

The equation for computing REGR_INTERCEPT is defined as follows:

$$\text{REGR\_INTERCEPT} = \text{AVG(y)} - \text{REGR\_SLOPE(y,x)AVG(x)}$$

where:

This variable …	Represents …
x	*independent_variable_expression*
y	*dependent_variable_expression*

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.

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)

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 *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For details on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

REGR_INTERCEPT Window Function

For the REGR_INTERCEPT window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example

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
```

REGR_R2

Purpose

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

Syntax

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

1101B418

where:

Syntax element ...	Specifies ...
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

REGR_R2 is ANSI SQL:2008 compliant.

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 Chapter 9: "Ordered Analytical Functions."

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

Computation

The coefficient of determination for two variables is the square of their Pearson product-moment correlation.

The equation for computing REGR_R2 is defined as follows:

$$\text{REGR\_R2} = \frac{\text{POWER(COUNT(xy)} \bullet \text{SUM(xy)} - \text{SUM(x)} \bullet \text{SUM(y) , 2)}}{\text{((COUNT(xy)} \bullet \text{SUM(x**2)} - \text{SUM(x)} \bullet \text{SUM(x))} \bullet \text{(COUNT(xy)} \bullet \text{SUM(y**2)} - \text{SUM(y)} \bullet \text{SUM(y)))}}$$

where:

This variable …	Represents …
x	*independent_variable_expression* x is the independent, or predictor, variable expression.
y	*dependent_variable_expression* y is the dependent, or response, variable expression.

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is …</th><th>THEN the format is …</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>numeric</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>REGR_R2(y,x)</td></tr>
</table>

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 *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

REGR_R2 Window Function

For the REGR_R2 window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example

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
```

REGR_SLOPE

Purpose

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

Syntax

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

1101B419

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_SLOPE is ANSI SQL:2008 compliant.

Definition

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.

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 Chapter 9: "Ordered Analytical Functions."

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

Computation

The equation for computing REGR_SLOPE is defined as follows:

$$\text{REGR\_SLOPE} = \frac{(\text{COUNT}(x)\text{SUM}(x*y)) - (\text{SUM}(x)\text{SUM}(y))}{(\text{COUNT}(x)\text{SUM}(x**2)) - (\text{SUM}(x)**2)}$$

where:

This variable ...	Represents ...
x	*independent_variable_expression*
y	*dependent_variable_expression*

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.

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)

For information on the default format of data types and the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

REGR_SLOPE Window Function

For the REGR_SLOPE window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example

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
```

REGR_SXX

Purpose

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

Syntax

——— REGR_SXX — (*dependent_variable_expression, independent_variable_expression*) ——

1101B420

where:

Syntax element ...	Specifies ...
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

REGR_SXX is ANSI SQL:2008 compliant.

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 Chapter 9: "Ordered Analytical Functions."

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

Computation

The equation for computing REGR_SXX is defined as follows:

$$\text{REGR\_SXX} = (\text{SUM}(x^{**}2)) - \left(\text{SUM}(x) \bullet \left(\frac{\text{SUM}(x)}{n}\right)\right)$$

where:

This variable ...	Represents ...
x	*independent_variable_expression* x is the independent, or predictor, variable expression.
n	COUNT(x)

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is ...</th><th>THEN the format is ...</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>REGR_SXX(y,x)</td></tr>
</table>

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 *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

REGR_SXX Window Function

For the REGR_SXX window function that performs a group, cumulative, or moving computation, see “Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)” on page 320.

Example

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
```

REGR_SXY

Purpose

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.

Syntax

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

1101B421

where:

Syntax element ...	Specifies ...
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

REGR_SXY is ANSI SQL:2008 compliant.

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 Chapter 9: "Ordered Analytical Functions."

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

Computation

The equation for computing REGR_SXY is defined as follows:

$$\text{REGR\_SXY} = (\text{SUM}(x*y)) - \left((\text{SUM}(x)) \bullet \left(\frac{\text{SUM}(y)}{n}\right)\right)$$

This variable …	Represents …
x	*independent_variable_expression* x is the independent, or predictor, variable expression.
y	*dependent_variable_expression* y is the dependent, or response, variable expression.
n	COUNT(x,y)

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is …</th><th>THEN the format is …</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>REGR_SXY(y,x)</td></tr>
</table>

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

REGR_SXY Window Function

For the REGR_SXY window function that performs a group, cumulative, or moving computation, see “Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)” on page 320.

Example

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
```

REGR_SYY

Purpose

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

Syntax

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

1101B422

where:

Syntax element ...	Specifies ...
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

REGR_SYY is ANSI SQL:2008 compliant.

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 Chapter 9: "Ordered Analytical Functions."

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

Computation

The equation for computing REGR_SYY is defined as follows:

$$\text{REGR\_SYY} = (\text{SUM}(y^{**}2)) - \left(\text{SUM}(y) \bullet \left(\frac{\text{SUM}(y)}{n}\right)\right)$$

where:

This variable ...	Represents ...
y	*dependent_variable_expression* y is the dependent, or response, variable expression.
n	COUNT(y)

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is ...</th><th>THEN the format is ...</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>REGR_SYY(y,x)</td></tr>
</table>

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

REGR_SYY Window Function

For the REGR_SYY window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example

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.

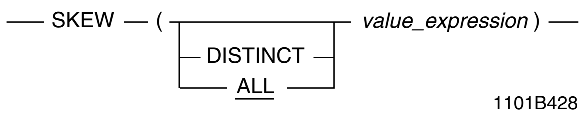
```
SELECT REGR_SYY(weight,height)
FROM regrtbl;

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

SKEW

Purpose

Returns the skewness of the distribution of *value_expression*.

Syntax



where:

Syntax element …	Specifies …
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the computation for the group. This is the default.
DISTINCT	that null and duplicate values specified by *value_expression* are eliminated from the computation for the group.
value_expression	a constant or column expression for which the skewness of the distribution of its values is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

SKEW is ANSI SQL:2008 compliant.

Definition

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.

Result Type and Attributes

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

Data Type	Format	Title
REAL	Default format of the REAL data type	SKEW(x)

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including SKEW, 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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

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:

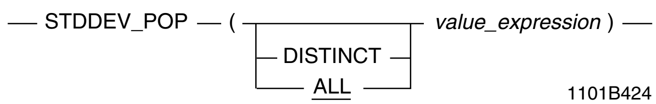
- Fewer than three non-null data points in the data used for the computation
- STDDEV_SAMP(x) = 0
- Division by zero

STDDEV_POP

Purpose

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

Syntax



where:

Syntax element …	Specifies …
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates of *value_expression* from the computation.
value_expression	a numeric constant or column expression whose population standard deviation is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

STDDEV_POP is ANSI SQL:2008 compliant.

Definition

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.

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 Chapter 9: "Ordered Analytical 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" on page 285.

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.

Computation

The equation for computing STDDEV_POP is as follows:

$$\text{STDDEV\_POP} = \text{SQRT}\left(\frac{\text{COUNT(x)SUM(x**2)} - \text{(SUM(x)**2)}}{\text{(COUNT(x)**2)}}\right)$$

where:

This variable …	Represents …
x	*value_expression*

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is ...</th><th>THEN the format is ...</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>STDDEV_POP(x)</td></tr>
</table>

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including STDDEV_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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

STDDEV_POP Window Function

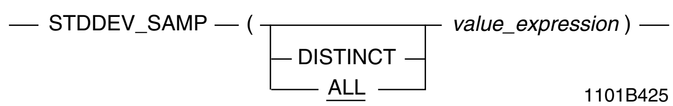
For the STDDEV_POP window function that performs a group, cumulative, or moving computation, see “Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)” on page 320.

STDDEV_SAMP

Purpose

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

Syntax



where:

Syntax element ...	Specifies ...
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates of *value_expression* from the computation.
value_expression	a numeric constant or column expression whose sample standard deviation is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

STDDEV_SAMP is ANSI SQL:2008 compliant.

Definition

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

The equation for computing STDDEV_SAMP is as follows:

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

where:

This variable …	Represents …
x	*value_expression*

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is …</th><th>THEN the format is …</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>STDDEV_SAMP(x)</td></tr>
</table>

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including STDDEV_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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

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 Chapter 9: “Ordered Analytical 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” on page 282.

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.

STDDEV_SAMP Window Function

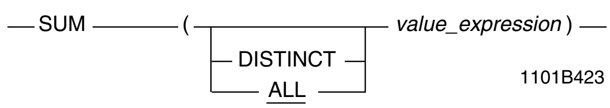
For the STDDEV_SAMP window function that performs a group, cumulative, or moving computation, see “Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)” on page 320.

SUM

Purpose

Returns a column value that is the arithmetic sum for a specified expression for a group.

Syntax



where:

Syntax element ...	Specifies ...
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the sum computation for the group. This is the default.
DISTINCT	that duplicate and non-null values specified by *value_expression* are eliminated from the sum computation for the group.
value_expression	a constant or column expression for which the sum is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

SUM is ANSI SQL:2008 compliant.

Result Type and Attributes

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	Same as the operand	Default format of the INTEGER data type	Sum(x)
character	Same as the operand	Default format for FLOAT	
UDT	Same as the operand	Format for the data type to which the UDT is implicitly cast	

<table>
<tr><th>Data Type of Operand</th><th>Data Type of Result</th><th>Format</th><th>Title</th></tr>
<tr><td>DECIMAL(n,m)</td><td>DECIMAL(p,m), where p is determined by the rules in the following table.
<table>
<tr><th>IF MaxDecimal in DBSControl is …</th><th>AND …</th><th>THEN p is …</th></tr>
<tr><td rowspan="3">0 or 15</td><td>$n \le 15$</td><td>15.</td></tr>
<tr><td>$15 < n \le 18$</td><td>18.</td></tr>
<tr><td>$n > 18$</td><td>38.</td></tr>
<tr><td rowspan="2">18</td><td>$n \le 18$</td><td>18.</td></tr>
<tr><td>$n > 18$</td><td>38.</td></tr>
<tr><td>38</td><td>n = any value</td><td>38.</td></tr>
</table></td><td>Default format for the data type of the operand</td><td rowspan="2">Sum(x)</td></tr>
<tr><td>Other than UDT, SMALLINT, BYTEINT, DECIMAL, or character</td><td>Same as the operand</td><td>Default format for the data type of the operand</td></tr>
</table>

For an explanation of the formatting characters in the format, and information on data type default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character

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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including SUM, 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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

Usage Notes

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

SUM is valid only for numeric data.

Nulls are not included in the result computation. For details, see "Manipulating Nulls" in *SQL Fundamentals* and "Aggregates and Nulls" on page 217.

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))
```

SUM Window Function

For the SUM function that returns the cumulative, group, or moving sum, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Example 1: 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 2: Face Value of Inventory

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

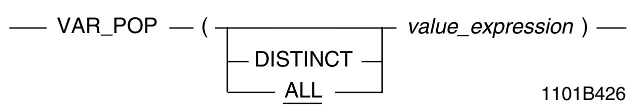
```
SELECT SUM(QUANTITY * Price)
FROM Inventory;

Sum((QUANTITY * Price))
-----------------------
          38,525,151.91
```

VAR_POP

Purpose

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

Syntax



where:

Syntax element ...	Specifies ...
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates of *value_expression* from the computation.
value_expression	a numeric constant or column expression whose population variance is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

VAR_POP is ANSI SQL:2008 compliant.

Definition

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

The equation for computing VAR_POP is as follows:

$$\text{VAR\_POP} = \frac{\text{COUNT(x)SUM(x**2)} - \text{(SUM(x)**2)}}{\text{(COUNT(x)**2)}}$$

where:

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

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is …</th><th>THEN the format is …</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
</td><td>VAR_POP(x)</td></tr>
</table>

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including VAR_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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

Combination With Other Functions

VAR_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 Chapter 9: "Ordered Analytical Functions."

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" on page 294.

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.

VAR_POP Window Function

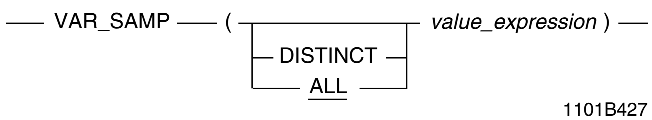
For the VAR_POP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

VAR_SAMP

Purpose

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

Syntax



where:

Syntax element ...	Specifies ...
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates of *value_expression* from the computation.
value_expression	a numeric constant or column expression whose sample variance is to be computed. The expression cannot contain ordered analytical or aggregate functions.

ANSI Compliance

VAR_SAMP is ANSI SQL:2008 compliant.

Definition

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

The equation for computing VAR_SAMP is as follows:

$$\text{VAR\_SAMP} = \frac{\text{COUNT}(x)\text{SUM}(x^{**}2) - (\text{SUM}(x)^{**}2)}{(\text{COUNT}(x))(\text{COUNT}(x) - 1)}$$

where:

This variable …	Represents …
x	*value_expression*

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.

Combination With Other Functions

VAR_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 Chapter 9: "Ordered Analytical Functions."

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. For information, see "VAR_POP" on page 291.

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.

Result Type and Attributes

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

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>REAL</td><td>
<table>
<tr><th>IF the operand is …</th><th>THEN the format is …</th></tr>
<tr><td>character</td><td>the default format for FLOAT.</td></tr>
<tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr>
<tr><td>UDT</td><td>the format for the data type to which the UDT is implicitly cast.</td></tr>
</table>
For details on the default format of data types, see SQL Data Types and Literals.
</td><td>VAR_SAMP(x)</td></tr>
</table>

Support for UDTs

By default, Teradata Database 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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including VAR_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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

VAR_SAMP Window Function

For the VAR_SAMP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

CHAPTER 9 Ordered Analytical Functions

This chapter describes *ordered analytical functions* that enable and expedite the processing of queries containing On Line Analytical Processing (OLAP) style decision support requests.

Ordered analytical functions include ANSI SQL:2008 compliant window functions, as well as Teradata SQL-specific functions.

Ordered Analytical Functions

Ordered analytical functions provide support for many common operations in analytical processing and data mining environments that require an ordered set of results rows or depend on values in a previous row.

For example, computing a seven-day running sum requires:

- First, that rows be ordered by date.
- Then, the value for the running sum must be computed,
  - 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.

Ordered Analytical Functions Benefits

Ordered analytical functions extend the Teradata Database query execution engine with the concept of an ordered set and with the ability to use the values from multiple rows in computing 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 Teradata Database 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
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

Syntax Alternatives for Ordered Analytical Functions

Teradata SQL supports two syntax alternatives for ordered analytical functions:

- ANSI SQL:2008 compliant

- Teradata

Window aggregate, rank, distribution, and row number functions are ANSI SQL:2008 compliant, while Teradata-specific functions are not.

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.

Relationship Between Teradata-Specific Functions and Window Functions

The following table identifies equivalent ANSI SQL:2008 window functions for Teradata-specific functions:

Teradata-Specific Functions	Equivalent ANSI SQL:2008 Window Functions
CSUM	SUM
MAVG	AVG
MDIFF(x, w, y)	composable from SUM
MLINREG	composable from SUM and COUNT
QUANTILE	composable from RANK and COUNT
RANK	RANK
MSUM	SUM

Window Feature

The ANSI SQL:2008 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

To see the syntax for the OVER() phrase and the associated clauses, refer to "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

The window feature can be applied to the following functions:

- AVG
- CORR
- COUNT
- COVAR_POP
- COVAR_SAMP
- MAX
- MIN
- PERCENT_RANK
- RANK
- REGR_AVGX
- REGR_AVGY
- REGR_COUNT
- REGR_INTERCEPT
- REGR_R2
- REGR_SLOPE
- REGR_SXX
- REGR_SXY
- REGR_SYY
- ROW_NUMBER
- STDDEV_POP
- STDDEV_SAMP
- SUM
- VAR_POP
- VAR_SAMP

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. See "RESET WHEN Phrase" on page 303 for details.

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
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. See "PARTITION BY Phrase" on page 301 for details.

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. See "Rules For Using a DEFAULT Function As Part of a RESET WHEN Condition" on page 304 for details.

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. Once the system has evaluated the DEFAULT function, it treats it like a constant 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.

See “DEFAULT” on page 503 for more information about the DEFAULT function.

Example 1

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 2

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 3

This example illustrates a window function with a nested aggregate.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 4

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 can be specified with the ANSI SQL:2008 compliant window aggregate functions:

- AVG
- CORR
- COUNT
- COVAR_POP
- COVAR_SAMP
- MAX
- MIN
- REGR_AVGX
- REGR_AVGY
- REGR_COUNT
- REGR_INTERCEPT
- REGR_R2
- REGR_SLOPE
- REGR_SXX
- REGR_SXY
- REGR_SYY
- STDDEV_POP
- STDDEV_SAMP
- SUM
- VAR_POP
- VAR_SAMP

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;
```

Applying Windows to Aggregate Functions

A window specification can be applied to the following ANSI SQL:2008 compliant aggregate functions:

- AVG
- CORR
- COUNT
- COVAR_POP
- COVAR_SAMP
- MAX
- MIN
- REGR_AVGX
- REGR_AVGY
- REGR_COUNT
- REGR_INTERCEPT
- REGR_R2
- REGR_SLOPE
- REGR_SXX
- REGR_SXY
- REGR_SYY
- STDDEV_POP
- STDDEV_SAMP
- SUM
- VAR_POP
- VAR_SAMP

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
```

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;
```

For details on the QUALIFY clause, see *SQL Data Manipulation Language*.

DISTINCT Clause Restriction

The DISTINCT clause is not permitted in windowed 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

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.

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, Teradata Database 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 Teradata Database 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;
```

Nesting Aggregates in Ordered Analytical Functions

You can nest aggregates in window functions, including the select list, HAVING, QUALIFY, and ORDER BY clauses. However, 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
…	…	…	…	

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.

Combination				Semantics
Window Function	Teradata-Specific Function	Aggregate Function	GROUP BY Clause	
	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 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.

Using Ordered Analytical Functions Examples

Example 1: 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
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 2: 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

Item	SMonth	Sales	Rank(Sales)	Moving Avg(Sales)
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 3: 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

Store	ProdID	Sales	Rank(Sales)
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
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).

Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)

Purpose

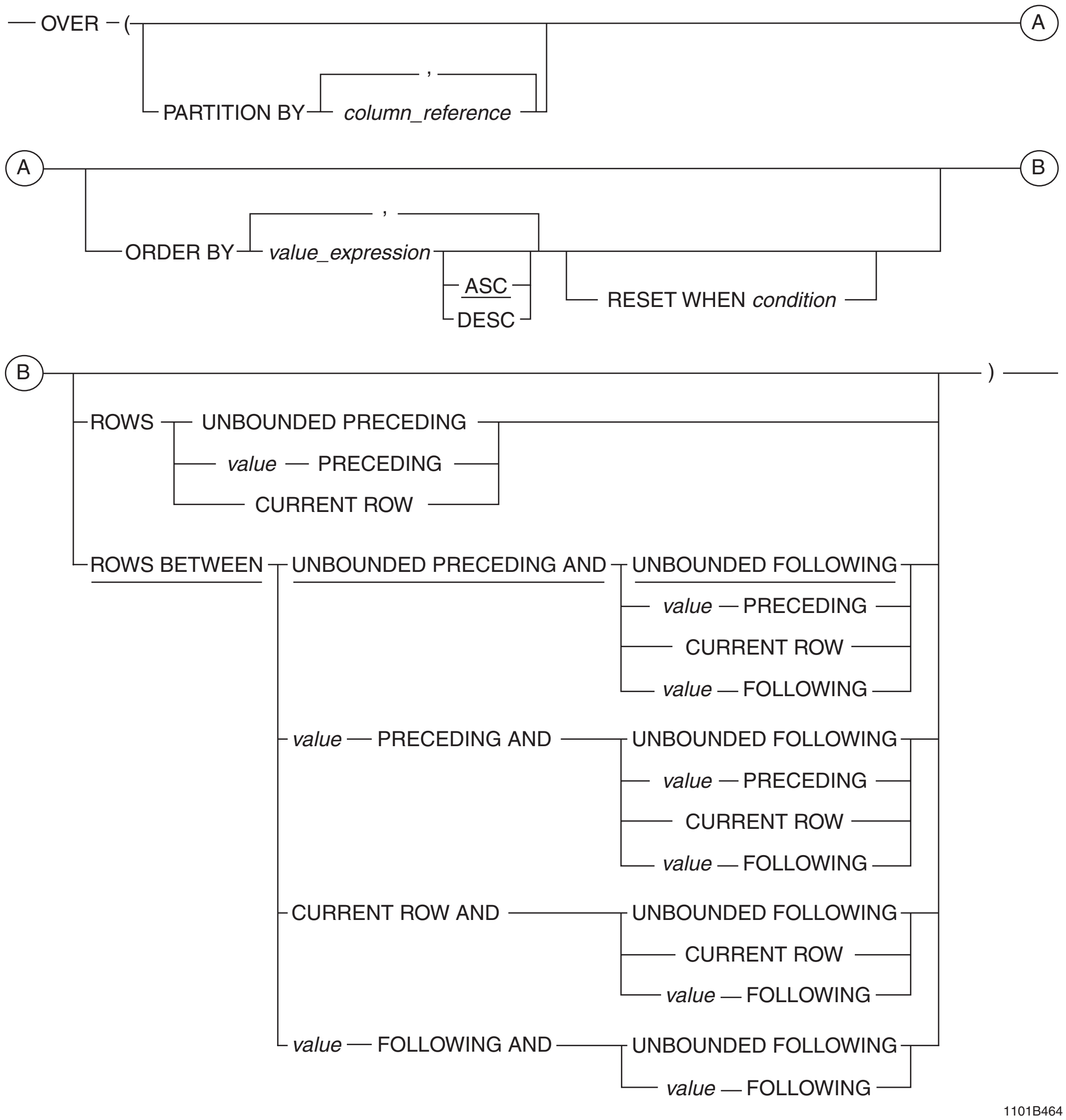
Cumulative, group, moving, or remaining computation of an aggregate function.

Type

ANSI SQL:2008 window aggregate function.

Syntax


AVG (value_expression)
A
COUNT (value_expression)
*
COVAR_POP (value_expression_1, value_expression_2)
COVAR_SAMP (value_expression_1, value_expression_2)
CORR (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)
SUM (value_expression)
VAR_POP (value_expression)
VAR_SAMP (value_expression)
A
window
1101A465


OVER
(
A
PARTITION BY
,
column_reference
A
B
ORDER BY
,
value_expression
ASC
DESC
RESET WHEN condition
B
)
ROWS
UNBOUNDED PRECEDING
value
PRECEDING
CURRENT ROW
ROWS BETWEEN
UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING
value
PRECEDING
CURRENT ROW
value
FOLLOWING
value
PRECEDING AND
UNBOUNDED FOLLOWING
value
PRECEDING
CURRENT ROW
value
FOLLOWING
CURRENT ROW AND
UNBOUNDED FOLLOWING
CURRENT ROW
value
FOLLOWING
value
FOLLOWING AND
UNBOUNDED FOLLOWING
value
FOLLOWING
1101B464

where:

Syntax element ...	Specifies ...
AVG CORR COUNT COVAR_POP COVAR_SAMP MAX MIN REGR_AVGX REGR_AVGY REGR_COUNT REGR_INTERCEPT REGR_R2 REGR_SLOPE REGR_SXX REGR_SXY REGR_SYY STDDEV_POP STDDEV_SAMP SUM VAR_POP VAR_SAMP	the aggregate function and arguments on which the window specification is applied. For descriptions of aggregate functions and arguments, see Chapter 8: "Aggregate Functions."
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.
PARTITION BY	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 is 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	in its *value_expression* the order in which the values in a group, or partition, are sorted.
ASC	ascending sort order. The default is ASC.
DESC	descending sort order.
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 is 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.

Syntax element ...	Specifies ...
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" on page 303 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.
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 is no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.
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 is 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 constant. 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 constant. The maximum number of rows in an aggregation group is 4096 when *value* FOLLOWING appears as the group start or group end.

ANSI Compliance

Window aggregate functions are partially ANSI SQL:2008 compliant.

In the presence of an ORDER BY clause and the absence of a ROWS or ROWS BETWEEN clause, ANSI SQL:2008 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.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

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 • 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 constants, constant 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 also 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(*x*,*y*) REGR_AVGY(*x*,*y*) REGR_COUNT(*x*,*y*) 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 a character type	FLOAT	Default format for FLOAT

<table>
<tr><th>Function</th><th>Result Type</th><th>Format</th></tr>
<tr><td>CORR(x,y)
COVAR_POP(x,y)
COVAR_SAMP(x,y)
REGR_AVGX(x,y)
REGR_AVGY(x,y)
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</td><td>Same data type as operand x.</td><td>Default format for the data type of operand x</td></tr>
<tr><td>COUNT(x)
COUNT(*)
REGR_COUNT(x,y)

where the transaction mode is ANSI</td><td>DECIMAL(p,0)
<table>
<tr><th>IF MaxDecimal in DBSControl is …</th><th>THEN p is …</th></tr>
<tr><td>0, 15, or 18</td><td>15.</td></tr>
<tr><td>38</td><td>38.</td></tr>
</table>
ANSI transaction mode uses DECIMAL because tables frequently have a cardinality exceeding the range of INTEGER.</td><td>Default format for resulting data type</td></tr>
<tr><td>COUNT(x)
COUNT(*)
REGR_COUNT(x,y)

where the transaction mode is Teradata</td><td>INTEGER

Teradata transaction mode uses INTEGER to avoid regression problems.</td><td>Default format for resulting data type</td></tr>
<tr><td>MAX(x), MIN(x)</td><td>Same data type as operand x.</td><td>Same format as operand x</td></tr>
<tr><td>SUM(x)

where x is a character type</td><td>Same as the operand x.</td><td>Default format for FLOAT</td></tr>
</table>

<table>
<tr><th>Function</th><th>Result Type</th><th>Format</th></tr>
<tr><td>SUM(x)
where x is a DECIMAL(n,m) type</td><td>DECIMAL(p,m), where p is determined by the rules in the following table.
<table>
<tr><th>IF MaxDecimal in DBSControl is …</th><th>AND …</th><th>THEN p is …</th></tr>
<tr><td>0 or 15</td><td>$n \le 15$</td><td>15.</td></tr>
<tr><td></td><td>$15 < n \le 18$</td><td>18.</td></tr>
<tr><td></td><td>$n > 18$</td><td>38.</td></tr>
<tr><td>18</td><td>$n \le 18$</td><td>18.</td></tr>
<tr><td></td><td>$n > 18$</td><td>38.</td></tr>
<tr><td>38</td><td>n = any value</td><td>38.</td></tr>
</table></td><td>Default format for DECIMAL</td></tr>
<tr><td>SUM(x)
where x is any numeric type other than DECIMAL</td><td>Same as the operand x.</td><td>Default format for the data type of the operand</td></tr>
</table>

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 *SQL Data Types and Literals*.

Result Title

The default title that appears in the heading for displayed or printed results depends on the type of computation performed.

<table>
<tr><th>IF the type of computation is …</th><th>THEN the result title is …</th></tr>
<tr><td>cumulative</td><td>Cumulative Function_name (argument_list)
For example, consider the following computation:
<pre>SELECT AVG(sales) OVER (PARTITION BY region
   ORDER BY smonth ROWS UNBOUNDED PRECEDING)
FROM sales_history;</pre>
The title that appears in the result heading is:
<pre>Cumulative Avg(sales)
---------------------</pre></td></tr>
</table>

IF the type of computation is …	THEN the result title is …

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

Ensure 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.

Using Window Aggregate Functions Instead of Teradata Functions

Be sure to use the ANSI-compliant window functions for any new applications you develop. Avoid using Teradata-specific functions such as MAVG, CSUM, and MSUM for applications intended to be ANSI-compliant and portable.

<table>
<tr><th>ANSI Function</th><th>Teradata Function</th><th>Relationship</th></tr>
<tr><td>AVG</td><td>MAVG</td><td> 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). </td></tr>
<tr><td>SUM</td><td>CSUM
MSUM</td><td> 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, 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.
</td></tr>
</table>

Example 1: Moving Average

Determine, for a business with several sales territories, the sales in each territory averaged over the current month and the preceding 2 months.

The following query might return the results found in the table that follows it.

```
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 meanings of the phrases in the example query are as follows:

Phrase	Meaning
PARTITION BY	Indicates that the rows delivered by the FROM clause, the rows of sales_history, should be assigned to groups, or partitions, based on their territory. If no PARTITION clause is specified, then the entire result set constitutes a single group, or partition.
ORDER BY	Indicates that rows are sorted in ascending order of month within each group, or partition. Ascending is the default sort order.
ROWS 2 PRECEDING	Defines the number of rows used to compute the moving average. In this case, the computation uses the current row and the 2 preceding rows of the group, or partition, as available.

Thus, the moving average for the first row of the partition East (199810), which has no preceding rows, is 10. That is, the value of the first row, the current row (10)/ the number of rows (1) = 10.

The moving average for the second row of the partition East (199811), which has only 1 preceding row, is 7. That is, the value of the second row, the current row, and the preceding row (10 + 4) / the number of rows (2) = 7.

The moving average for the third row of the partition East (199812), which has 2 preceding rows, is 8. That is, the value of the third row, the current row, and the 2 preceding rows (10 + 4 + 10) / the number of rows (3) = 8. And so on.

Month is specified as a six-digit numeric in the YYYYMM format.

Example 2: 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 3: 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 4: 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 5: 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 6: 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 7: 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 8: 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
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 9: 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
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 10: 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       ?
```

CSUM

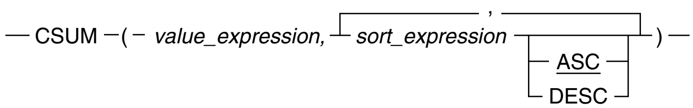
Purpose

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.

Type

Teradata-specific function.

Syntax



1101A398

where:

Syntax element …	Specifies …
value_expression	a numeric constant 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 constant or column expression or comma-separated list of constant or column expressions to be used to sort the values. For example, CSUM(Sale, Region ASC, Store DESC), where Sale is the *value_expression,* and Region ASC, Store DESC is the *sort_expression* list. The expression cannot contain any ordered analytical or aggregate functions.
ASC	ascending sort order. The default sort direction is ASC.
DESC	descending sort order.

ANSI Compliance

CSUM is a Teradata extension to the ANSI SQL:2008 standard.

Using SUM Instead of CSUM

The use of CSUM is strongly discouraged. It is a Teradata extension to the ANSI SQL:2008 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.

For more information on the SUM window function, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Meaning of Cumulative Sums

CSUM accumulates a sum over an ordered set of rows, providing the current value of the SUM on each row.

Result Type and Attributes

The data type, format, and title for CSUM(*x*, *y direction*) are as follows:

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>Same as operand x</td><td><table><tr><th>IF operand x is ...</th><th>THEN the format is ...</th></tr><tr><td>character</td><td>the default format for FLOAT.</td></tr><tr><td>numeric</td><td>the same format as x.</td></tr></table></td><td>CSum(x, y direction)</td></tr>
</table>

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 *SQL Data Types and Literals*.

Example 1

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 2

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

MAVG

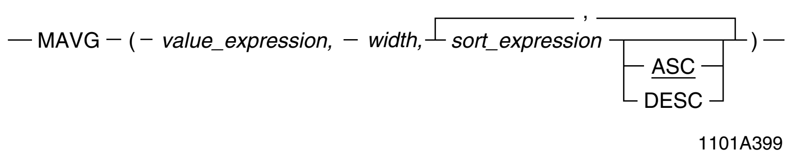
Purpose

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.

Syntax



where:

Syntax element ...	Specifies ...
value_expression	a numeric constant or column expression for which a moving average is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
width	number of previous rows to be used in computing the moving average. The value is always a positive integer constant. The maximum is 4096.
sort_expression	a constant or column expression or comma-separated list of constant 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	ascending sort order. The default sort direction is ASC.
DESC	descending sort order.

ANSI Compliance

MAVG is a Teradata extension to the ANSI SQL:2008 standard.

Using AVG Instead of MAVG

The use of MAVG is strongly discouraged. It is a Teradata extension to the ANSI SQL:2008 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.

For more information on the AVG window function, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Result Type and Attributes

The data type, format, and title for MAVG(*x*, *w*, *y direction*) are as follows:

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>Same as operand x</td><td><table><tr><th>IF operand x is …</th><th>THEN the format is …</th></tr><tr><td>character</td><td>the default format for FLOAT.</td></tr><tr><td>numeric
date
interval</td><td>the same format as x.</td></tr></table></td><td>MAvg(x, w, y direction)</td></tr>
</table>

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

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.

Example 1

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 2

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

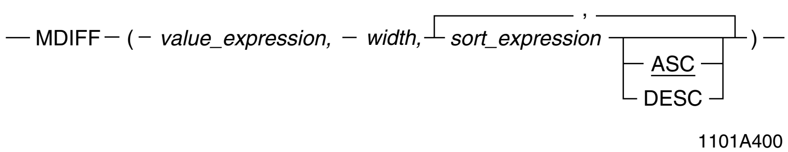
Purpose

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.

Syntax



where:

Syntax element ...	Specifies ...
value_expression	a numeric column or constant 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 computing the moving difference. The value is always a positive integer constant. The maximum is 4096.
sort_expression	a constant or column expression or comma-separated list of constant or column expressions to be used to sort the values. For example, MDIFF(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	ascending sort order. The default sort direction is ASC.
DESC	descending sort order.

ANSI Compliance

MDIFF is a Teradata extension to the ANSI SQL:2008 standard.

Meaning of Moving Difference

A common business metric is to compare activity for some variable in a current time period to the activity for the same variable in another time period a fixed distance in the past. For example, you might want to compare current sales volume against sales volume for preceding quarters. This is a moving difference calculation where *value_expression* would be the quarterly sales volume, width is 4, and *sort_expression* might be the quarter_of_calendar column from the SYS_CALENDAR.Calendar system view.

Using SUM Instead of MDIFF

The use of MDIFF is strongly discouraged. It is a Teradata extension to the ANSI SQL:2008 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)
```

For more information on the SUM window function, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Result Type and Attributes

The data type, format, and title for MDIFF(*x*, *w*, *y direction*) are as follows:

Data Type and Format			Title
IF operand x is …	**THEN the data type is …**	**AND the format is …**	MDiff(x, w, y direction)
character	the same as x.	the default format for FLOAT.	
numeric	the same as x.	the same format as x.	
date	INTEGER	the default format for INTEGER.	

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

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.

Example 1

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 2

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

MLINREG

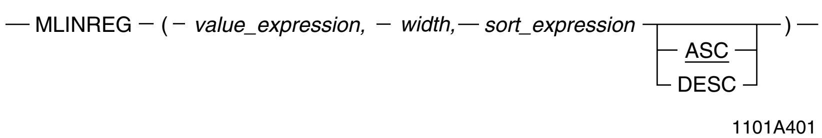
Purpose

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.

Type

Teradata-specific function.

Syntax



where:

Syntax element ...	Specifies ...
value_expression	a numeric constant or column expression for which a predicted value is to be computed. The expression cannot contain any ordered analytical or aggregate functions. The data type of the expression must be numeric or a data type that Teradata Database can successfully convert implicitly to numeric.
width	the number of rows to use to compute the function. *width*-1 previous rows are used to compute the linear regression and the row value itself is used for calculating the predicted value. The value is always a positive integer constant greater than 2. 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 Teradata Database can successfully convert implicitly to numeric.
ASC	ascending sort order. The default sort direction is ASC.
DESC	descending sort order.

ANSI Compliance

MLINREG is Teradata extension to the ANSI SQL:2008 standard.

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:2008 standard, and is retained only for backward compatibility with existing applications.

Result Type and Attributes

The data type, format, and title for MLINREG(*x*, *w*, *y direction*) are as follows:

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>Same as operand x</td><td><table><tr><th>IF operand x is …</th><th>THEN the format is …</th></tr><tr><td>character</td><td>the default format for FLOAT.</td></tr><tr><td>• numeric
• date
• interval</td><td>the same format as x.</td></tr></table></td><td>MLinReg(x, w, y direction)</td></tr>
</table>

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 *SQL Data Types and Literals*.

Default Independent Variable

MLINREG assumes that the independent variable is described by *sort_expression*.

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.

MLINREG Report Structure

All rows in the results table except the first two, which are always null, display the predicted value.

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 value 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  ?
...
```

MSUM

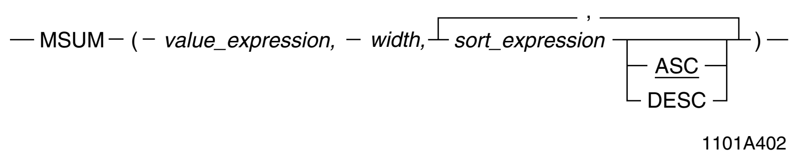
Purpose

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.

Syntax



where:

Syntax element ...	Specifies ...
value_expression	a numeric constant or column expression for which a moving sum is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
width	the number of previous rows to be used in computing the moving sum. The value is always a positive integer constant. The maximum is 4096.
sort_expression	a constant or column expression or comma-separated list of constant or column expressions to be used to sort the values. For example, MSUM(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.
ASC	ascending sort order. The default sort direction is ASC.
DESC	descending sort order.

ANSI Compliance

MSUM is a Teradata extension to the ANSI SQL:2008 standard.

Using SUM Instead of MSUM

The use of MSUM is strongly discouraged. It is a Teradata extension to the ANSI SQL:2008 standard, and is equivalent to the ANSI-compliant SUM window function. MSUM is retained only for backward compatibility with existing applications.

For more information on the SUM window function, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

Result Type and Attributes

The data type, format, and title for MSUM(*x*, *w*, *y direction*) are as follows:

<table>
<tr><th>Data Type</th><th>Format</th><th>Title</th></tr>
<tr><td>Same as operand x</td><td><table><tr><th>IF operand x is …</th><th>THEN the format is …</th></tr><tr><td>character</td><td>the default format for FLOAT.</td></tr><tr><td>numeric</td><td>the same format as x.</td></tr></table></td><td>MSum(x, w, y direction)</td></tr>
</table>

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

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*.

PERCENT_RANK

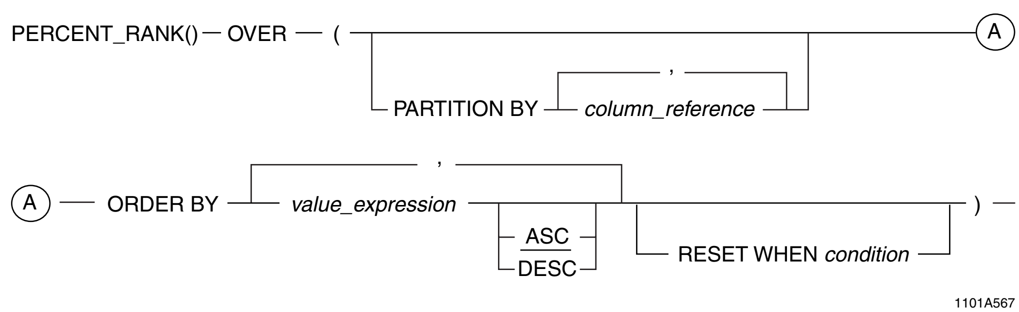
Purpose

Returns the relative rank of rows for a *value_expression*.

Type

ANSI SQL:2008 window function.

Syntax



where:

Syntax element …	Specifies …
OVER	how the values, grouped according to the PARTITION BY and RESET WHEN clauses and named by *value_expression* in the ORDER BY clause, are ranked.
PARTITION BY	in its *column_reference* the column, or columns, according to which ranking resets. PARTITION BY is optional. If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, specified by the ORDER BY clause, constitutes a single group or partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the column, or columns, being ranked.
ASC	ascending sort order. The default order is ASC.
DESC	descending sort order.

Syntax element ...	Specifies ...
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 is no RESET WHEN or PARTITION BY clauses, then the entire result set 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. For more information, see "RESET WHEN Condition Rules" on page 303and the "QUALIFY Clause" in *SQL Data Manipulation Language*.

ANSI Compliance

The PERCENT_RANK window function, which uses ANSI-specific syntax, is ANSI SQL:2008 compliant.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

Computation

The formula for PERCENT_RANK is:

$$\text{PERCENT\_RANK} = \frac{(\text{RK} - 1)}{(\text{NR} - 1)}$$

where:

This variable ...	Represents the ...
RK	rank of the row
NR	number of rows in the window partition

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.

PERCENT_RANK has this value …	FOR the result row assigned this rank …
1.0	highest in the result.

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)

For an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example 1

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 2

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
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

QUANTILE

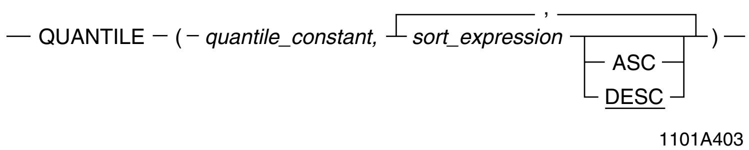
Purpose

Computes the quantile scores for the values in a group.

Type

Teradata-specific function.

Syntax



where:

Syntax element ...	Specifies ...
quantile_constant	a positive integer constant used to define the number of quantile partitions to be used.
sort_expression	a constant or column expression or comma-separated list of constant or column expressions to be used to sort the values. For example, QUANTILE(10, Region ASC, Store DESC), where 10 is the *quantile_constant* and Region ASC, Store DESC is the *sort_expression* list.
ASC	ascending sort order.
DESC	descending sort order. The default sort direction is DESC.

ANSI Compliance

QUANTILE is a Teradata extension to the ANSI SQL:2008 standard.

Definition

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.

Using ANSI Window Functions Instead of QUANTILE

The use of QUANTILE is strongly discouraged. It is a Teradata extension to the ANSI SQL:2008 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()
```

QUANTILE Report

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.

Quantile Value Range

Quantile values range from 0 through (Q-1), where Q is the number of quantile partitions specified by *quantile_constant*.

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)

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example 1

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 2

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 3

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.

RANK

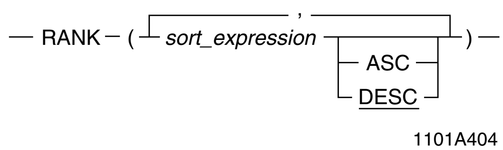
Purpose

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.

Type

Teradata-specific function.

Syntax



where:

Syntax element …	Specifies …
sort_expression	a constant or column expression or comma-separated list of constant or column expressions to be used to sort the values. For example, RANK(Region ASC, Store DESC), where Region ASC, Store DESC is the *sort_expression* list. The expression cannot contain any ordered analytical or aggregate functions.
ASC	ascending sort order.
DESC	descending sort order. The default sort direction is DESC.

ANSI Compliance

RANK is a Teradata extension to the ANSI SQL:2008 standard.

Using ANSI RANK Instead of Teradata RANK

The use of Teradata RANK is strongly discouraged. It is a Teradata extension to the ANSI SQL:2008 standard, and is equivalent to the ANSI-compliant RANK window function. Teradata RANK is retained only for backward compatibility with existing applications.

For more information on the RANK window function, see "RANK" on page 362.

Meaning of 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.

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*.

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)

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example 1

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 2

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 3

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

RANK

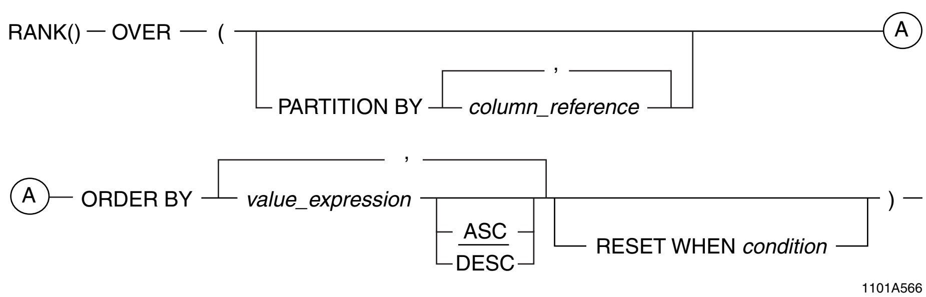
Purpose

Returns an ordered ranking of rows based on the *value_expression* in the ORDER BY clause.

Type

ANSI SQL:2008 window function.

Syntax



where:

Syntax element ...	Specifies ...
OVER	how the values, grouped according to the PARTITION BY and RESET WHEN clauses and named by *value_expression* in the ORDER BY clause, are ranked.
PARTITION BY	in its *column_reference* the column, or columns, according to which ranking resets. PARTITION BY is optional. If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, specified by the ORDER BY clause, constitutes a single group, or partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the column, or columns, being ranked.
ASC	ascending rank, or sort order. The default order is ASC.
DESC	descending rank, or sort order.

Syntax element ...	Specifies ...
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 is no RESET WHEN or PARTITION BY clauses, then the entire result set 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. For more information, see "RESET WHEN Condition Rules" on page 303 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.

ANSI Compliance

The RANK window function is ANSI SQL:2008 compliant.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

Meaning of Rank

RANK 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.

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)

For an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example

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

Notice that 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.

ROW_NUMBER

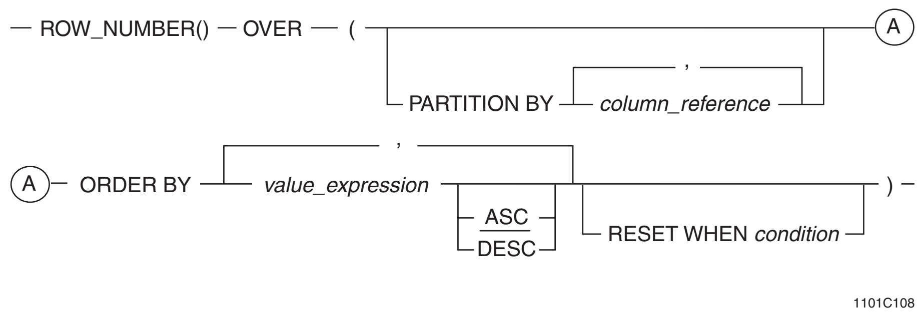
Purpose

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:2008 window function.

Syntax



where:

Syntax element ...	Specifies ...
OVER	the window partition and ordering.
PARTITION BY	the column, or columns, according to which the result set is partitioned. PARTITION BY is optional. If there is no PARTITION BY or RESET WHEN clauses, then the entire result set, specified by the ORDER BY clause, constitutes a partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the order in which to sort the values in the partition.
ASC	ascending sort order. The default order is ASC.
DESC	descending sort order.

Syntax element ...	Specifies ...
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 is no RESET WHEN or PARTITION BY clauses, then the entire result set 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. For more information, see "RESET WHEN Condition Rules" on page 303 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.

ANSI Compliance

The ROW_NUMBER window function is ANSI SQL:2008 compliant.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

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).
```

For more information on COUNT, see "Window Aggregate Functions (AVG, CORR, COUNT, COVAR_POP, COVAR_SAMP, MAX, MIN, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, STDDEV_POP, STDDEV_SAMP, SUM, VAR_POP, VAR_SAMP)" on page 320.

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
```

CHAPTER 10 String Operator and Functions

This chapter describes the concatenation operator and functions that operate on character, byte, and numeric strings.

String Functions

SQL provides a concatenation operator and string functions to translate, concatenate, and perform other operations on strings.

IF you want to ...	THEN use ...
concatenate strings	concatenation operator
convert a character string to hexadecimal representation	CHAR2HEXINT
get the starting position of a substring within another string	• INDEX • POSITION
convert a character string to lowercase	LOWER
get the Soundex code for a character string	SOUNDEX
extract a substring from another string	• SUBSTRING • SUBSTR
translate a character string to another server character set	TRANSLATE
determine if TRANSLATE can successfully translate a character string to a specified server character set	TRANSLATE_CHK
trim specified pad characters or bytes from a character or byte string	TRIM
convert a character string to uppercase	UPPER
convert a character string to VARGRAPHIC representation	VARGRAPHIC

String Definition

The functions documented in this chapter are designed primarily to work with strings of characters. Because many of them can also process byte and numeric constant 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:2008 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
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 in this chapter.

FOR more information on …	SEE …
attribute functions that return descriptive information about strings, such as: • BYTE • CHARACTER_LENGTH/ CHAR_LENGTH • OCTET_LENGTH	Chapter 12: "Attribute Functions."
comparison operators	Chapter 4: "Comparison Operators."
the LIKE predicate	Chapter 11: "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

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 with regards to multibyte characters and some single-byte characters.

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.

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.

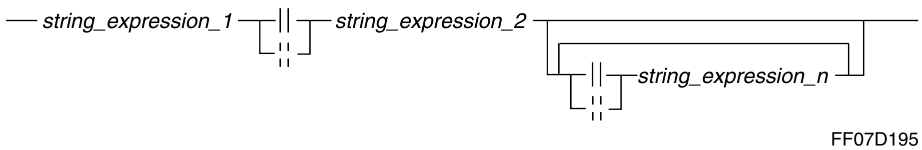
For details, see "Implicit Character-to-Character Translation" on page 595.

Concatenation Operator

Purpose

Concatenates string expressions.

Syntax



where:

Syntax element ...	Specifies ...
string_expression_1	a byte, numeric, or character string or string expression.
string_expression_2	
string_expression_n	

ANSI Compliance

EXCLAMATION POINT character pairs (!!) are Teradata extensions to the ANSI SQL:2008 standard. Do not use them as concatenation operators.

Solid and broken VERTICAL LINE character pairs (||) are ANSI SQL:2008 compliant forms of the concatenation operator.

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" on page 655
- 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" on page 595.

- 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 *SQL Data Definition Language*.

  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 *Utilities*.

  For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

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*:

<table>
<tr><th>Data Type</th><th>Heading</th></tr>
<tr><td>
<table>
<tr><th>IF the arguments are …</th><th>THEN the result is a …</th></tr>
<tr><td>byte strings</td><td>byte string.</td></tr>
<tr><td>numeric or character strings or UDTs that are implicitly cast to character strings</td><td>character string.</td></tr>
</table>
</td><td>(arg1||arg2)</td></tr>
</table>

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*)

IF this argument …	Is this data type or attribute …	THEN the result is this data type or attribute …
both	BYTE	BYTE(*n*)
	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.

Example 1: Using Concatenation to Create More Readable Results

Constants, 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 2: 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 3: 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 4: 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.

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. For details, see "Implicit Character-to-Character Translation" on page 595.

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	'abcΔabc'	VARCHAR(7) CHARACTER SET UNICODE
clatin || csjis	'abcabc'	VARCHAR(6) CHARACTER SET UNICODE
cunicode || csjis	'abcΔabc'	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 for Japanese Character Sets

The following tables show the results of concatenating string expressions under each of the Kanji character sets supported by Teradata Database.

These examples assume that the string expressions follow the rules defined in the chapter "SQL Data Definition" in *SQL Data Types and Literals.*

For an explanation of symbols and other notation in the examples, see "Character Shorthand Notation Used In This Book" on page 748.

Example 1: KanjiEBCDIC

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
<**ABC**>	<**DEF**>G	<**ABCDEF**>G
<**ABC**>	<>	<**ABC**>

string_expression_1	string_expression_2	Result
<**ABC**>a	<**DEF**>	<**ABC**>a<**DEF**>

Example 2: KanjiEUC

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
ABCm	**DEF**g	**ABC**m**DEF**g
ss_3**A** ss_2**B** m	ss_3**C**	ss_3**A** ss_2**B** m ss_3**C**

Example 3: KanjiShift-JIS

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
mn**ABCX**	**B**	mn**ABCXB**
mn**ABCX**	g	mn**ABCX**g

CHAR2HEXINT

Purpose

Returns the hexadecimal representation for a character string.

Syntax

```
── CHAR2HEXINT── ( character_string_expression ) ──
```

1101E173

where:

Syntax element ...	Specifies ...
character_string_expression	a character string or character string expression for which the hexadecimal representation is to be returned.

ANSI Compliance

CHAR2HEXINT is a Teradata extension to the ANSI SQL:2008 standard.

Argument Types

Use CHAR2HEXINT on character strings or character string expressions.

By default, Teradata Database 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. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including CHAR2HEXINT, 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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

CHAR2HEXINT is not supported for CLOBs.

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 depends on whether Japanese language support was enabled during sysinit.

IF the system uses this type of language support …	THEN the result specifies this server character set …
standard	LATIN
Japanese	KANJI1

CHAR2HEXINT and Constant Strings

You can apply CHAR2HEXINT to a quoted character string to determine its hexadecimal equivalent.

Character constants are treated as VARCHAR(*n*) CHARACTER SET UNICODE, where *n* is the length of the constant.

The following statement and results illustrate how CHAR2HEXINT operates on constant strings:

```
SELECT CHAR2HEXINT('123');

Char2HexInt('123')
-----------------------
003100320033
```

Example 1

Assume that the system was enabled with Japanese language support during 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 2

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
```

INDEX

Purpose

Returns the position in *string_expression_1* where *string_expression_2* starts.

Syntax

```
—— INDEX —— ( string_expression_1 ,string_expression_2 ) ——
```

FF07D253

where:

Syntax element …	Specifies …
string_expression_1	a full string to be searched.
string_expression_2	a substring to be searched for its position within the full string.

ANSI Compliance

INDEX is a Teradata extension to the ANSI SQL:2008 standard.

Use POSITION instead of INDEX for ANSI SQL:2008 compliance.

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 *SQL Data Definition Language*.

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 *Utilities*.

INDEX does not support CLOBs or BLOBs.

For more information on implicit type conversion, see Chapter 17: "Data Type Conversions."

Result Type and Attributes

Here are the default result type and attributes for INDEX(*arg1*, *arg2*):

Data Type	Heading
INTEGER	Index(*arg1*, *arg2*)

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.

  For details, see "Rules for KANJI1 Server Character Set" on page 384.

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. For a description of implicit character translation rules, see "Implicit Character-to-Character Translation" on page 595.

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.

<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>either argument has a CASESPECIFIC attribute (either by default or specified explicitly)</td><td>simple Latin letters are considered to be matching only if they are the same letters and the same case.</td></tr>
<tr><td>both arguments have a NOT CASESPECIFIC attribute (either by default or specified explicitly)</td><td>before the operation begins, some characters are converted to uppercase.
<table>
<tr><th>IF the character is a ...</th><th>THEN the character is ...</th></tr>
<tr><td>lowercase simple Latin letter</td><td>converted to uppercase before the operation begins.</td></tr>
<tr><td>non-Latin single byte character</td><td rowspan="3">not converted to uppercase.</td></tr>
<tr><td>multibyte character</td></tr>
<tr><td>byte indicating a transition between single-byte and multibyte character data</td></tr>
</table>
</td></tr>
</table>

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

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.

For an example of when the two functions return different results for the same data, see “How POSITION and INDEX Differ” on page 392.

Example 1

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

Example 2

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 3

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

IF *string_1* contains ...	AND *string_2* contains ...	THEN the result is ...
MN<**AB**>P	P	9
MXN<**AB**>P	<**B**>	7

Example 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 KanjiEUC.

IF *string_1* contains ...	AND *string_2* contains ...	THEN the result is ...
a b ss$_3$**A**	ss$_3$**A**	3
a b ss$_2$**B**	ss$_2$**B**	3
CS1_DATA	**A**	6
a b ss$_2$**D** ss$_3$**E** ss$_2$**F**	ss$_2$**F**	5
a b **C** ss$_2$**D** ss$_3$**E** ss$_2$**F**	ss$_2$**F**	6
CS1_Dm**ATA**	**A**	7

Example 5

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 ...
mn**ABCX**	**B**	4
mn**ABCX**	**X**	6

Example 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).

Example 7

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 8

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
```

LOWER

Purpose

Returns a character string identical to *character_string_expression*, except that all uppercase letters are replaced by their lowercase equivalents.

Syntax

```
— LOWER — (character_string_expression) ————————
```

FF07D091

where:

Syntax element ...	Specifies ...
character_string_expression	a character string or character string expression for which all uppercase characters are to be replaced by their lowercase equivalents.

ANSI Compliance

LOWER is ANSI SQL:2008 compliant.

Argument Types

Use LOWER on character strings or character string expressions, except for CLOBs.

By default, Teradata Database performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and a predefined character type, except for CLOB.

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 *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

Result Type and Attributes

Here are the default result type and attributes for LOWER(*arg*):

Data Type	Heading
Same type as *arg*	Lower(*arg*)

Usage Notes

The LOWER function allows users who want ANSI portability to have case blind comparisons with ANSI-compliant syntax.

You can also replace characters with uppercase equivalents. For more information, see "UPPER" on page 424.

Restrictions

The LOWER function operates with the LATIN server character set. If the type of argument for LOWER is anything other than LATIN, LOWER attempts to translate the non-LATIN string to LATIN before evaluation. If the string cannot be converted successfully, an error is returned.

Note that a constant string is an acceptable argument because it is implicitly converted from UNICODE to LATIN before it is evaluated.

Examples

In the following examples, columns charfield_1 and charfield_2 have CASESPECIFIC comparison attributes.

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.

Example 1

The following example compares the strings on a case blind basis.

```
SELECT id
FROM names
WHERE LOWER(charfield_1) = LOWER(charfield_2);
```

Example 2

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));
```

POSITION

Purpose

Returns the position in *string_expression_2* where *string_expression_1* starts.

Syntax

— POSITION —(*string_expression_1* — IN — *string_expression_2*) —

FF07D090

where:

Syntax element ...	Specifies ...
string_expression_1	a substring to be searched for its position within the full string.
string_expression_2	a full string to be searched.

ANSI Compliance

POSITION is ANSI SQL:2008 compliant.

Use POSITION instead of INDEX for ANSI SQL:2008 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.)

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 *SQL Data Definition Language*.

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 *Utilities*.

For more information on implicit type conversion, see Chapter 17: "Data Type Conversions."

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*)

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.

SOUNDEX

Purpose

Returns a character string that represents the Soundex code for *string_expression*.

Syntax

SOUNDEX — (— *string_expression* —) —

KO01A060

where:

Syntax element …	Specifies …
string_expression	a character string or expression that contains a surname to be evaluated in simple Latin characters. Soundex is case insensitive. Embedded or trailing pad characters within *character_string* return an error to the requestor.

ANSI Compliance

SOUNDEX is a Teradata extension to the ANSI SQL:2008 standard.

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, Teradata Database 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 *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

Definition: 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.

Definition: 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 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.

Example 1

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 2

“Example 2” and “Example 3” on page 397 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 3

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 Invalid 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.

STRING_CS

Purpose

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*.

The result is not guaranteed correct, but should work for most strings likely to be encountered.

Syntax

— STRING_CS — (*string_expression*) —

1101A515

where:

Syntax element ...	Specifies ...
string_expression	a CHAR or VARCHAR character string or expression.

ANSI Compliance

STRING_CS is a Teradata extension to the ANSI SQL:2008 standard.

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.

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.

<table>
<tr><th>IF the result value is …</th><th>THEN the heuristic found that string_expression …</th></tr>
<tr><td>0</td><td>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.
<table>
<tr><th>IF string_expression contains …</th><th>AND you want it to be interpreted as …</th><th>THEN use …</th></tr>
<tr><td>0x5C</td><td>REVERSE SOLIDUS</td><td rowspan="2">a single-byte character set.</td></tr>
<tr><td>0x7E</td><td>TILDE</td></tr>
<tr><td>0x5C</td><td>YEN SIGN</td><td rowspan="2">any of the following:
• KANJISJIS_0S
• KANJIEBCDIC5026_0I
• KANJIEBCDIC5035_0I
• KATAKANAEBCDIC
• KANJIEUC_0U</td></tr>
<tr><td>0x7E</td><td>OVERLINE</td></tr>
</table></td></tr>
<tr><td>1</td><td>uses the encoding of one of the following:
• KANJIEBCDIC5026_0I
• KANJIEBCDIC5035_0I
• KATAKANAEBCDIC</td></tr>
<tr><td>2</td><td>uses the encoding of KANJIEUC_0U.</td></tr>
<tr><td>3</td><td>uses the encoding of KANJISJIS_0S.</td></tr>
</table>

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.

For more information on TRANSLATE, see "TRANSLATE" on page 407.

Example 1: 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 2: Using STRING_CS to Translate a KANJI1 String to UNICODE

Consider the SysNames table from the preceding example, “Example 1: 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:

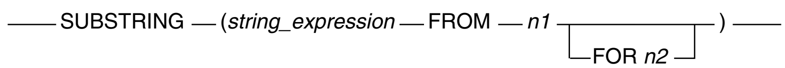
```
SELECT CASE STRING_CS(SysName)
   WHEN 0 THEN TRANSLATE(SysName USING KANJI1_SBC_TO_UNICODE)
   WHEN 1 THEN TRANSLATE(SysName USING KANJI1_KANJIEBCDIC_TO_UNICODE)
   WHEN 2 THEN TRANSLATE(SysName USING KANJI1_KANJIEUC_TO_UNICODE)
   WHEN 3 THEN TRANSLATE(SysName USING KANJI1_KANJISJIS_TO_UNICODE)
   ELSE TRANSLATE(SysName USING KANJI1_SBC_TO_UNICODE)
   END
FROM SysNames;
```

SUBSTRING/SUBSTR

Purpose

Extracts a substring from a named string based on position.

ANSI Syntax

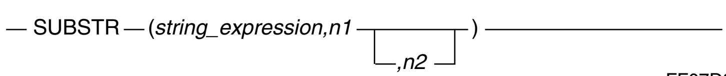


FF07D256

where:

Syntax Element ...	Specifies ...
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*.
FOR	a keyword indicating that the searched substring is bounded on the right by the value *n2*. If you omit FOR *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 FOR *n2*, trailing binary zeros or pad characters are trimmed.
n2	the length of the substring to extract from *string_expression*. If *n2* < 0, the function returns an error.

Teradata Syntax



FF07D257

where:

Syntax Element ...	Specifies ...
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 be extracted from *string_expression*. If *string_expression* is a BYTE or CHAR type and you omit *n2*, trailing binary zeros or pad characters are trimmed. If *n2* < 0, the function returns an error.

ANSI Compliance

SUBSTRING is ANSI SQL:2008 compliant.

SUBSTR is a Teradata extension to the ANSI SQL:2008 standard.

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. For more information on CREATE CAST, see *SQL Data Definition Language*.

  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. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 17: “Data Type Conversions.”

Result Type and Attributes

Here are the default result type and attributes for SUBSTR(*string*, *n1*, *n2*) and SUBSTRING(*string* FROM *n1* FOR *n2*):

<table>
<tr><th>Data Type</th><th>Heading</th></tr>
<tr><td>
<table>
<tr><th>IF the string argument is a …</th><th>THEN the result type is …</th></tr>
<tr><td>BLOB</td><td>BLOB(n).</td></tr>
<tr><td>byte string other than BLOB</td><td>VARBYTE(n).</td></tr>
<tr><td>CLOB</td><td>CLOB(n).</td></tr>
<tr><td>numeric, or character string other than CLOB</td><td>VARCHAR(n).</td></tr>
</table>
</td><td>Substring(string From n1 For n2)
Substr(string, n1, n2)</td></tr>
</table>

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.

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 $(0 \leq n2)$
- $(n1 < 1)$ AND $(0 \leq n2)$ AND $((n2 + n1 - 1) \leq 0)$

Usage Rules for SUBSTRING and SUBSTR

SUBSTRING is the ANSI SQL:2008 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.)

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.

Example 1

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 2

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 3

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 4

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 5

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');
```

Here are some examples of how to use SUBSTR to extract substrings from the KanjiEUC client character set.

Function	Result
`SELECT SUBSTR(c2, 2, 3) FROM cstr;`	'2年a'
`SELECT SUBSTR(c1, 2, 2) FROM cstr;`	'bc'

Example 6

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
	`SELECT SUBSTRING(c1 FROM 3 FOR 8) FROM ctable1;`	<**ABC**>P
	`SELECT SUBSTRING(c1 FROM 4) FROM ctable1;`	<**BC**>P

Example 7

The following table shows examples for the KanjiEUC client character set, where ctable1 is the table defined in Example 6.

IF c1 contains ...	THEN this query ...	Returns ...
A ss$_2$**B** **CD**	`SELECT SUBSTR(c1,2) FROM ctable1;`	ss$_2$**B** **CD**
ss$_3$**A** ss$_2$**B** ss$_3$**C** ss$_2$**D**	`SELECT SUBSTR(c1,2,2) FROM ctable1;`	ss$_2$**B** ss$_3$**C**

Example 8

The following table shows examples for KanjiShift-JIS client character set, where ctable1 is the table defined in Example 6.

IF c1 contains ...	THEN this query ...	Returns ...
mn**ABC<u>X</u>**	`SELECT SUBSTR(c1, 6, 1) FROM ctable1;`	**<u>X</u>**
	`SELECT SUBSTR(c1,4) FROM ctable1;`	**BC<u>X</u>**

Example 9

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.

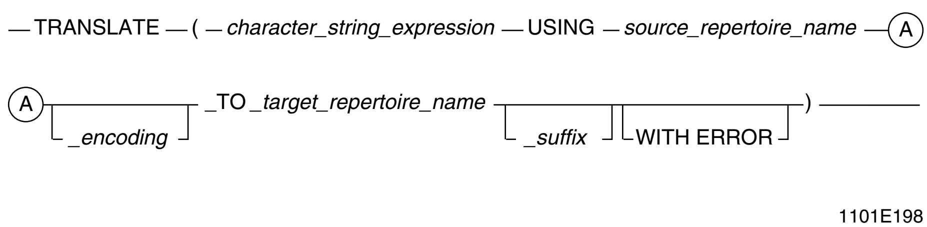
```
INSERT sub_text (text)
SELECT SUBSTRING (text FROM 9 FOR 128000)
FROM full_text;
```

TRANSLATE

Purpose

Converts a character string or character string expression from one server character set to another server character set.

Syntax



where:

Syntax element …	Specifies …
character_string_expression	a character string to translate to another server character set. If the string or string expression is not a character type, an error is returned.
source_repertoire_name	the source character set of the string to translate. For supported values, see “Supported Translations Between Character Sets” on page 410. 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. For details, see “Supported Translations Between Character Sets” on page 410.
_encoding	an optional 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*.

<table>
<tr><th>Syntax element ...</th><th>Specifies ...</th></tr>
<tr><td>_encoding
(continued)</td><td>
<table>
<tr><th>IF the translation is from this character set ...</th><th>THEN use this value for _encoding ...</th></tr>
<tr><td>• KatakanaEBCDIC
• KanjiEBCDIC5026_0I
• KanjiEBCDIC5038_0I</td><td>_KanjiEBCDIC</td></tr>
<tr><td>KanjiEUC_0U</td><td>_KanjiEUC</td></tr>
<tr><td>KanjiShiftJIS_0S</td><td>_KANJISJIS</td></tr>
<tr><td>ASCII or EBCDIC</td><td>_SBC</td></tr>
</table>
</td></tr>
<tr><td>target_repertoire_name</td><td>the target character set of the string to translate. For supported values, see “Supported Translations Between Character Sets” on page 410.

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. For details, see “Supported Translations Between Character Sets” on page 410.</td></tr>
<tr><td>_suffix</td><td>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.</td></tr>
<tr><td>WITH ERROR</td><td>that the translation replaces offending characters in the string with a designated error character, instead of reporting an error.

For details, see “Error Characters Assigned by the WITH ERROR Option” on page 413).</td></tr>
</table>

ANSI Compliance

TRANSLATE is ANSI SQL:2008 compliant.

Argument Types

Use TRANSLATE on character strings or character string expressions.

By default, Teradata Database 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 *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

Result Type and Attributes

The default attributes for TRANSLATE (*string* USING *source*_TO_*target*) are as follows.

<table>
<tr><th>Data Type</th><th>Heading</th></tr>
<tr><td>
<table>
<tr><th>IF the argument is ...</th><th>THEN the result is ...</th></tr>
<tr><td>• CHAR
• VARCHAR</td><td>VARCHAR(n)
CHARACTER SET target</td></tr>
<tr><td>CLOB</td><td>CLOB(n)
CHARACTER SET target</td></tr>
</table>
where source_TO_target determines the character set value of target, according to the supported translations in “Supported Translations Between Character Sets” on page 410.
</td><td>Translate(string USING source_to_target)</td></tr>
</table>

Supported Translations for CLOB Strings

The following translations are supported for CLOB strings:

- LATIN_TO_UNICODE
- UNICODE_TO_LATIN

Supported Translations Between Character Sets

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
UNICODE_TO_LOCALE	UNICODE	KANJI1
		LATIN
UNICODE_TO_UNICODE_FoldSpace	UNICODE	UNICODE

Value of *source*_TO_*target*	Source Character Set	Target Character Set
UNICODE_TO_UNICODE_Fullwidth	UNICODE	UNICODE
UNICODE_TO_UNICODE_Halfwidth	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 • 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	KANJI1_SBC
	• 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
	• 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 • UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is LATIN or KANJI1_SBC	non-LATIN

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. For a list of the encodings, see the definition of _encoding in "Syntax" on page 407.

This form of translation is also useful for migrating object names. For information, see "Migration" on page 415.

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_Fullwidth	This translation maps any character with a fullwidth variant to that variant. 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.

*source*_TO_*target*	Meaning
UNICODE_TO_UNICODE_Halfwidth	This translation maps any character with a halfwidth variant to that variant, and all fullwidth variants to their non-fullwidth counterparts. Other characters remain unchanged by the translation.
UNICODE_TO_GRAPHIC_VarGraphic	This translation is an ANSI equivalent to the VARGRAPHIC function.

Note that these translations are useful for maintaining more information as a step in translating GRAPHIC to LATIN and vice versa.

For details on the mappings, see *International Character Set Support*.

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.

For details on which characters are converted to U+0020, see *International Character Set Support*.

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 Teradata Database treats constants 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年' 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年' 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年' USING UNICODE_TO_LATIN WITH ERROR) where ε represents the designated error character for LATIN (0x1A).	'abcε'	VARCHAR(4) CHARACTER SET LATIN

Related Topics

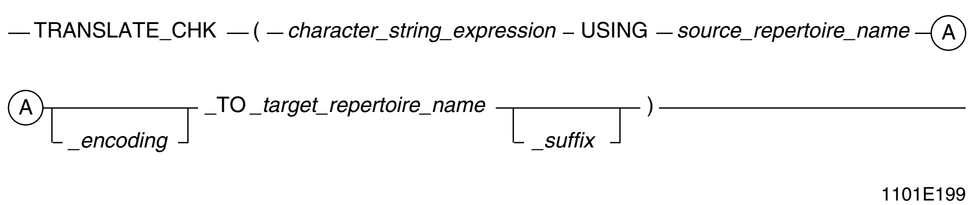
For details on the mappings that Teradata Database uses for the TRANSLATE function, see *International Character Set Support*.

TRANSLATE_CHK

Purpose

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.

Syntax



where:

Syntax element ...	Specifies ...
character_string_expression	a character string to be translated to another server character set. If the string or string expression is not a character type, an error is returned.
source_repertoire_name	the source character set of the string to be translated. For supported values, see "Supported Translations Between Character Sets" on page 410. 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. For details, see "Supported Translations Between Character Sets" on page 410.
_encoding	an optional 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*.

<table>
<tr><th>Syntax element …</th><th>Specifies …</th></tr>
<tr><td>_encoding
(continued)</td><td>
<table>
<tr><th>IF the translation is from this character set …</th><th>THEN use this value for _encoding …</th></tr>
<tr><td>• KatakanaEBCDIC
• KanjiEBCDIC5026_0I
• KanjiEBCDIC5038_0I</td><td>_KanjiEBCDIC</td></tr>
<tr><td>KanjiEUC_0U</td><td>_KanjiEUC</td></tr>
<tr><td>KanjiShiftJIS_0S</td><td>_KANJISJIS</td></tr>
<tr><td>ASCII or EBCDIC</td><td>_SBC</td></tr>
</table>
</td></tr>
<tr><td>target_repertoire_name</td><td>the target character set of the string to translate. For supported values, see “Supported Translations Between Character Sets” on page 410.

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. For details, see “Supported Translations Between Character Sets” on page 410.</td></tr>
<tr><td>_suffix</td><td>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.</td></tr>
</table>

ANSI Compliance

TRANSLATE_CHK is a Teradata extension to the ANSI SQL:2008 standard.

Argument Types

Use TRANSLATE_CHK on character strings and character string expressions.

By default, Teradata Database 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 *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

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*)

Result Values

Value	Meaning
0	The string can be translated without error.
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.

Example 1

Function	Result
TRANSLATE_CHK('abc' USING UNICODE_TO_LATIN)	0
TRANSLATE_CHK('abc年' USING UNICODE_TO_LATIN)	4

Example 2

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 3

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 4

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.

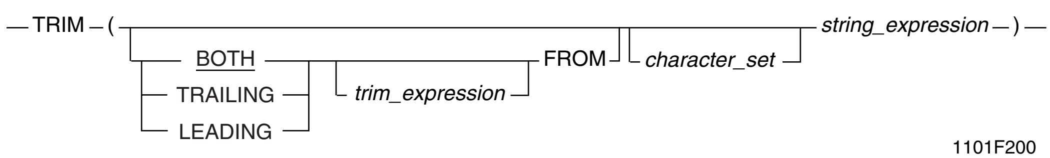
```
SELECT ckanji1
FROM table_1
WHERE TRANSLATE_CHK(ckanji1 USING Kanji1_KanjiEUC_TO_Unicode) <> 0;
```

TRIM

Purpose

Takes a character or byte *string_expression* argument, trims the specified pad characters or bytes, and returns the trimmed *string_expression*.

Syntax



where:

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td>BOTH
TRAILING
LEADING</td><td>how to trim the specified trim character or byte from string_expression.
The keywords and their meanings appear in the following table.
<table>
<tr><th>Keyword</th><th>Meaning</th></tr>
<tr><td>BOTH</td><td>Trim both trailing and leading characters or bytes.</td></tr>
<tr><td>TRAILING</td><td>Trim only trailing characters or bytes.</td></tr>
<tr><td>LEADING</td><td>Trim only leading characters or bytes.</td></tr>
</table>
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.</td></tr>
<tr><td>trim_expression</td><td>the 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.</td></tr>
<tr><td>FROM</td><td>a keyword required when BOTH, TRAILING, or LEADING are specified.</td></tr>
<tr><td>character_set</td><td>the name of the server character set to associate with the string expression.</td></tr>
</table>

<table>
<tr><th>Syntax Element ...</th><th>Specifies ...</th></tr>
<tr><td>character_set (continued)</td><td>Possible values appear in the following table.
<table>
<tr><th>Value</th><th>Server Character Set</th></tr>
<tr><td>_Latin</td><td>LATIN</td></tr>
<tr><td>_Unicode</td><td>UNICODE</td></tr>
<tr><td>_KanjiSJIS</td><td>KANJISJIS</td></tr>
<tr><td>_Graphic</td><td>GRAPHIC</td></tr>
</table></td></tr>
<tr><td>string_expression</td><td>a byte or character string or string expression to be trimmed.</td></tr>
</table>

ANSI Compliance

TRIM is ANSI SQL:2008 compliant.

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. For more information on CREATE CAST, see *SQL Data Definition Language*.

  Implicit type conversion of UDTs for system operators and functions, including TRIM, 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 *Utilities*.

For more information on implicit type conversion, see Chapter 17: “Data Type Conversions.”

Result Type and Attributes

Here are the default result type and attributes for TRIM(*string_expression*):

<table>
<tr><th>Data Type</th><th>Heading</th></tr>
<tr><td>
<table>
<tr><th>IF string_expression is …</th><th>THEN the result type is …</th></tr>
<tr><td>a byte string</td><td>VARBYTE.</td></tr>
<tr><td>a numeric expression or character string</td><td>VARCHAR.</td></tr>
</table>
</td><td>Trim(BOTH FROM string_expression)</td></tr>
</table>

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.

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.

Example 1

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 2

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 3

The following statement trims trailing SEMICOLON characters from the specified string.

```
SELECT TRIM( TRAILING ';' FROM textfield) FROM texttable;
```

Example 4

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 'ΔΔabc年ΔΔΔ');	'abc年'
SELECT TRIM(_Unicode 'ΔΔabc年ΔΔΔ');	'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.

UPPER

Purpose

Returns a character string identical to *character_string_expression*, except that all lowercase letters are replaced by their uppercase equivalents.

Syntax

```
—— UPPER —( character_string_expression) ———
```

FF07D258

where:

Syntax element …	Specifies …
character_string_expression	a character string or character string expression for which all lowercase characters are to be replaced by their uppercase equivalents.

ANSI Compliance

UPPER is ANSI SQL:2008 compliant.

Argument Types

UPPER is valid only for character strings and character string expressions, except for CLOBs.

By default, Teradata Database 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 *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including UPPER, 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 *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

Result Type and Attributes

Here are the default result type and attributes for UPPER(*arg*):

Data Type	Heading
Same type as *arg*	Upper(*arg*)

Usage Notes

The UPPER 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" on page 388.

Restrictions

UPPER does not convert multibyte characters to uppercase in the KANJI1 server character set.

Example 1

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 2

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 3

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'.

VARGRAPHIC

Purpose

Returns the VARGRAPHIC representation of the character data in *character_string_expression*.

Syntax

```
—— VARGRAPHIC —— ( character_string_expression ) ——
```

1101E197

where:

Syntax element ...	Specifies ...
character_string_expression	a character string or character string expression for which the VARGRAPHIC representation is to be returned.

ANSI Compliance

VARGRAPHIC is a Teradata extension to the ANSI SQL:2008 standard.

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. For more information on CREATE CAST, see *SQL Data Definition Language*.

  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. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 17: “Data Type Conversions.”

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*)

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 details, see "VARGRAPHIC Function Conversion Tables" on page 430.

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.

Example 1

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abcΔ**'
VARGRAPHIC('abc')	'**abc**'

Example 2

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 ('def', '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**.)

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. (Also see the notes following the table.)

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
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[a]		N/A	N/A
0F[b]		FEFE	FEFE
10		FEFE	FEFE
11[c]	£	424A	424A
12[d]	¬	425F	FEFE
13	\	43E0	FEFE
14	~	43A1	FEFE
15		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
16		FEFE	FEFE
17		FEFE	FEFE
18		FEFE	FEFE
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

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
33	3	42F3	43F3
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
4F	O	42D6	42D6

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
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
6A	j	4291	FEFE
6B	k	4292	FEFE
6C	l	4293	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	**JIS X 0201 Printable Character**	**KanjiEBCDIC 5026/ 5035**	**Katakana EBCDIC**
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	-[e]	42A1	42A1
7F		FEFE	FEFE
80		FEFE	FEFE
81		FEFE	FEFE
82		FEFE	FEFE
83		FEFE	FEFE
84		FEFE	FEFE
85		FEFE	FEFE
86		FEFE	FEFE
87		FEFE	FEFE
88		FEFE	FEFE
89		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
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
A0		FEFE	FEFE
A1	f	4341	4341
A2	g	4342	4342
A3	h	4343	4343
A4	i	4344	4344
A5	j	4345	4345
A6	k	4346	4346

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
A7	l	4347	4347
A8	m	4348	4348
A9	n	4349	4349
AA	o	4351	4351
AB	p	4352	4352
AC	q	4353	4353
AD	r	5454	4354
AE	s	4355	4355
AF	t	4356	4356
B0	u	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
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

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
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
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	v	43BE	43BE
DF	w	43BF	43BF
E0		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
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
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

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
FE		FEFE	FEFE
FF		FEFE	FEFE

a. 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

b. 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.

c. Pound Sterling sign

d. Logical NOT

e. Overline

f. Ideographic period

g. Left corner bracket

h. Right corner bracket

i. Ideographic comma

j. Katakana middle dot

k. Katakana letter WO

l. Katakana letter A

m. Katakana letter small I

n. Katakana letter small U

o. Katakana letter small E

p. Katakana letter small O

q. Katakana letter small YA

r. Katakana letter small YU

s. Katakana letter small YO

t. Katakana letter small WO

u. Katakana-Hiragana prolonged sound mark

v. Katakana-Hiragana voiced sound mark

w. Katakana-Hiragana semi-voice sound mark

CHAPTER 11 Logical Predicates

This chapter describes SQL logical predicates.

For information on comparison operators, see Chapter 4: "Comparison Operators."

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 constant
- An arithmetic expression
- A Period value expression
- The DEFAULT function
- A built-in function such as CURRENT_DATE or USER that evaluates to a system variable

Logical predicates are also referred to as conditional expressions. The ANSI SQL standard refers to them as search conditions.

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" on page 102.	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.
Overlaps	OVERLAPS	Tests whether two time periods overlap.
Period predicates	CONTAINS MEETS PRECEDES SUCCEEDS	Operates on two Period expressions or one Period expression and one DateTime expression and evaluates to TRUE, FALSE, or UNKNOWN.
	IS UNTIL_CHANGED IS NOT UNTIL_CHANGED	Tests whether the ending bound of a Period value expression 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, Period, and UDT types.

Data Type	Restrictions
BLOB	Predicates do not support BLOB or CLOB data types.
CLOB	You can explicitly cast BLOBs to BYTE and VARBYTE types and CLOBs to CHARACTER and VARCHAR types, and use the results in a predicate.

<table>
<tr><th>Data Type</th><th>Restrictions</th></tr>
<tr><td>PERIOD</td><td>Predicates are only supported for CONTAINS, MEETS, PRECEDES, SUCCEEDS, and IS [NOT] UNTIL_CHANGED.</td></tr>
<tr><td>UDT</td><td>
<table>
<tr><th>Predicate</th><th>Restrictions</th></tr>
<tr><td>LIKE</td><td rowspan="2">The LIKE and OVERLAPS logical predicates do not support UDTs.</td></tr>
<tr><td>OVERLAPS</td></tr>
<tr><td>EXISTS/
NOT EXISTS</td><td>Multiple UDTs involved as predicate operands must be identical types because Teradata Database 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 SQL Data Definition Language.</td></tr>
<tr><td>BETWEEN/
NOT BETWEEN</td><td rowspan="2">• Multiple UDTs involved as predicate operands must be identical types because Teradata Database 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.
Teradata Database 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 SQL Data Definition Language.</td></tr>
<tr><td>IN/NOT IN</td></tr>
</table>
</td></tr>
</table>

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, Teradata Database can derive the column context of the DEFAULT function even though the column name is omitted.

```
SELECT * FROM Employee WHERE Dept_No < DEFAULT;
```

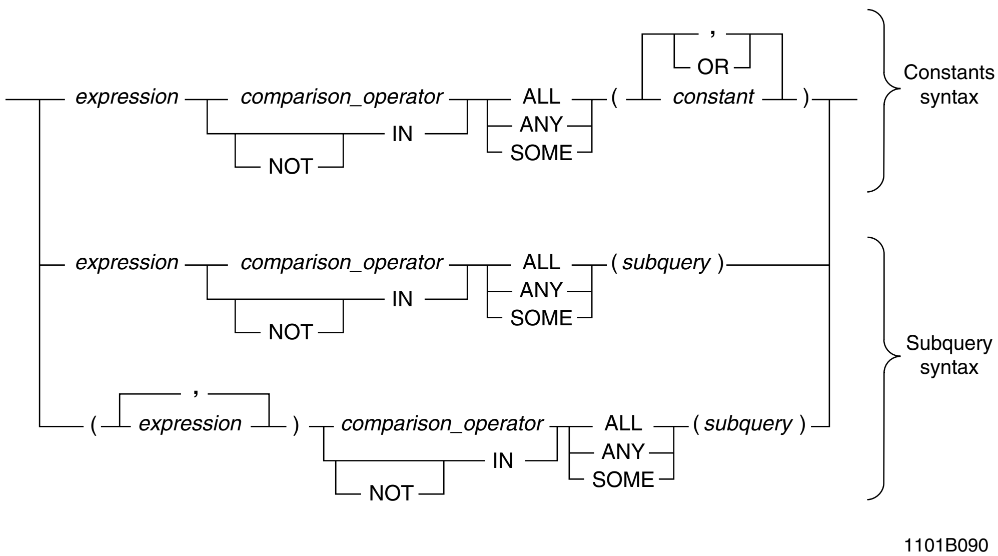
Note that if the DEFAULT function evaluates to null, the predicate is unknown and the WHERE condition is false.

ANY/ALL/SOME Quantifiers

Purpose

Enables quantification in a comparison operation or IN/NOT IN predicate.

Syntax



where:

Syntax element ...	Specifies ...
expression	an expression that specifies a value.
comparison_operator	a comparison operator that compares the expression or list of expressions and the constants in the list (Constants syntax) or the subquery (Subquery syntax) to produce a TRUE, FALSE or UNKNOWN result. For more information on comparison operators, see Chapter 4: “Comparison Operators.”
[NOT] IN	a predicate that tests the existence of the expression or list of expressions in the list of constants (Constants syntax) or the subquery (Subquery syntax) to produce a TRUE, FALSE, or UNKNOWN result. For more information on IN/NOT IN, see “IN/NOT IN” on page 459.
constant	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.

ANSI Compliance

ANY, SOME, and ALL are ANSI SQL:2008 compliant quantifiers.

ANY/ALL/SOME Quantifiers and Constant Syntax

When a list of constants 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 constant in the list produces true results.
	ANY	the comparison of *expression* and any constant in the list is true.
	SOME	
IN	ALL	*expression* is equal to every constant in the list.
	ANY	*expression* is equal to any constant in the list.
	SOME	
NOT IN	ALL	*expression* is not equal to any constant in the list.
	ANY	*expression* is not equal to every constant 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*.

IF this quantifier is specified ...	AND the predicate is ...	THEN the result is ...	WHEN ...
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 ...[a]
`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))`	

a. 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)`

Example 1

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`
`WHERE DeptNo = ANY (100,300,500) ;`
	`SELECT EmpNo, Name, DeptNo`
	`FROM Employee`
	`WHERE (DeptNo = 100)`
	`OR (DeptNo = 300)`
	`OR (DeptNo = 500) ;`
	and
	`SELECT EmpNo, Name, DeptNo`
	`FROM Employee`
	`WHERE DeptNo IN (100,300,500) ;`

Example 2

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 3

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);
```

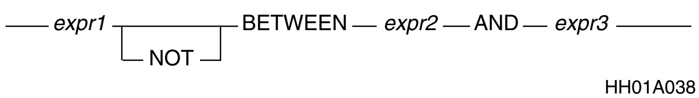
IF you use this query …	THEN the result is …
`SELECT * FROM t WHERE x IN ANY (1,2)`	1, 2
`SELECT * FROM t WHERE x = SOME (1,2)`	1, 2
`SELECT * FROM t WHERE x NOT IN ALL (1,2)`	3, 4, 5
`SELECT * FROM t WHERE NOT (x IN ANY (1,2))`	3, 4, 5
`SELECT * FROM t WHERE NOT (x = SOME (1,2))`	3, 4, 5
`SELECT * FROM t WHERE x NOT IN SOME (1, 2)`	1, 2, 3, 4, 5
`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

Purpose

Tests whether an expression value is between two other expression values.

Syntax



ANSI Compliance

BETWEEN and NOT BETWEEN are ANSI SQL:2008 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 ;
```

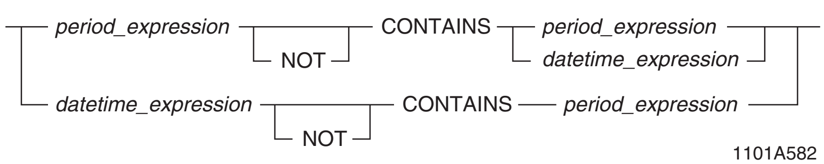
CONTAINS

Purpose

Predicate that operates on two Period expressions or one Period expression and one DateTime expression and evaluates to TRUE, FALSE, or UNKNOWN.

If both expressions have a Period data type, returns TRUE if the beginning bound of the first expression is less than or equal to the beginning bound of the second expression and the ending bound of the first expression is greater than or equal to the ending bound of the second expression; otherwise, returns FALSE. If the first expression is a Period expression and the second expression is a DateTime expression, returns TRUE if the beginning bound of the Period expression is less than or equal to the DateTime expression and the ending bound of the Period expression is greater than the DateTime expression; otherwise, returns FALSE. If the first expression is a DateTime expression and the second expression is a Period expression, returns TRUE if the DateTime expression is less than or equal to beginning bound of the Period expression and the DateTime expression plus one granule is greater than or equal to the ending bound of the Period expression; otherwise, returns FALSE. If either expression is NULL, the operator returns UNKNOWN.

Syntax



where:

Syntax element...	Specifies...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
period_expression	any expression that evaluates to a Period data type. **Note:** The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.

Error Conditions

If either expression evaluates to a data type that is other than a Period or DateTime, an error is reported.

If the expressions do not have comparable data types, an error is reported.

Example

In the following example, the CONTAINS operator is used in the WHERE clause.

```
SELECT * FROM employee WHERE period2 CONTAINS period1;
```

Assume the query is executed on the following table employee where period1 and period2 are PERIOD(DATE) columns:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is as follows:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')

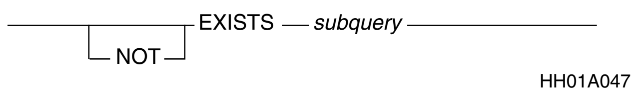
EXISTS/NOT EXISTS

Purpose

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.

Syntax



ANSI Compliance

EXISTS and NOT EXISTS are ANSI SQL:2008 compliant.

Usage Notes

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.

Note that use of 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.

Subquery Restrictions

The subquery cannot specify a SELECT AND CONSUME statement.

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" on page 455.

For information on IN and NOT IN, see "IN/NOT IN" on page 459.

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.

Example 1: 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
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
```

For a full explanation of correlated subqueries, see “Correlated Subqueries” in *SQL Data Manipulation Language*.

Example 2: 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 1: EXISTS with Correlated Subqueries" on page 455, the following rows are returned:

```
SName          SNo
-----          ---
Helen Chu      1
```

IN/NOT IN

Purpose

Tests the existence of the value of an expression or expression list in a comparable set in one of two ways:

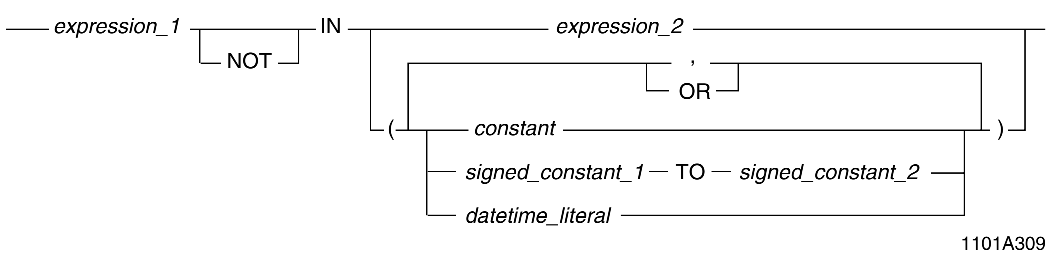
- Compares the value of an expression with values in an explicit list of constants.
- Compares values in a list of expressions with values and in a set of corresponding expressions in a subquery.

ANSI Compliance

IN and NOT IN are ANSI SQL:2008 compliant.

Using TO in a list of constants is a Teradata extension to the ANSI standard.

Syntax 1: *expression* IN and NOT IN *expression* or constants



where:

Syntax element ...	Specifies ...
expression_1	the value of the expression whose existence is to be tested in *expression_2* or in an explicit list of constants named by *constant*, *signed_constant* TO *signed_constant*, or *datetime_literal*.

<table>
<tr><th>Syntax element ...</th><th>Specifies ...</th></tr>
<tr><td>IN
NOT IN</td><td>whether the test is inclusive or exclusive.
<table>
<tr><th>You can substitute ...</th><th>FOR ...</th></tr>
<tr><td>IN ANY
IN SOME
= ANY
= SOME</td><td>IN, unless a list of constants is specified and includes signed_constant_1 TO signed_constant_2</td></tr>
<tr><td><> ALL
NOT IN ALL</td><td>NOT IN, unless a list of constants is specified and includes signed_constant_1 TO signed_constant_2</td></tr>
</table></td></tr>
<tr><td>expression_2</td><td>the value in which the existence of expression_1 is to be tested.</td></tr>
<tr><td>constant</td><td>constant
macro parameter
built-in value such as TIME or DATE</td></tr>
<tr><td>signed_constant_1 TO signed_constant_2</td><td>a range of constants.</td></tr>
<tr><td>datetime_literal</td><td>an ANSI DateTime literal.</td></tr>
</table>

Result

If IN is used with a list of constants, the result is true if the value of *expression_1* is:

- equal to any *constant* in the list,
- between *signed_constant_1* and *signed_constant_2*, inclusively, when *signed_constant_1* is less than or equal to *signed_constant_2*, or
- between *signed_constant_2* and *signed_constant_1*, inclusively, when *signed_constant_2* is less than *signed_constant_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 constants; the NOT IN condition is satisfied if none of the values in the list of constants 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*

THE condition is true for this form …	WHEN …
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 = 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);`

Usage Notes

If IN is used with a single-term operator, that operator can be a constant or an expression. If a multiple-term operator is used, that operator must consist of constants; expressions are not allowed.

The *expression_1* data type and the *constant* values must be compatible. Implicit conversion rules are the same as for the comparison operators.

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" on page 453.

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 ...[a]
`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)`

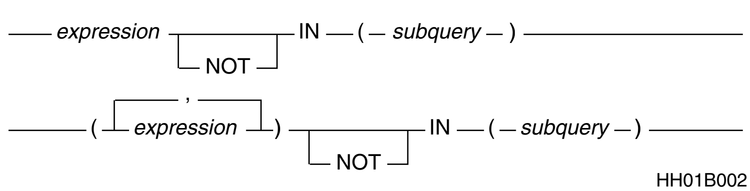
a. In the equivalences, if *op* is NOT IN, then NOT *op* is IN, not NOT NOT IN.

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)`
`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:



where:

Syntax element ...	Specifies ...
expression	the value of the expression whose existence is to be tested in *subquery*.

Syntax element …	Specifies …
subquery	a SELECT statement that returns values that satisfy the stated search criterion. The *subquery* must: • Be enclosed in parentheses. • Not end with a semicolon. • Select the same number of expressions as are defined in the expression list. • Not specify a SELECT AND CONSUME statement.

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 T1 WHERE *expression_list_1* NOT IN (SELECT *expression_list_2* FROM T2 WHERE *search_condition*);	the *search_condition* on T2 returns no rows	all T1 rows, including those containing a null value in the first field of *expression_list_1*, are returned.

[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);
```

Example 1

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 2

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 3

This example shows the behavior of IN/NOT IN with a list of constants.

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);
```

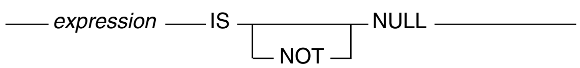
IF you use this query ...	THEN the result is ...
`SELECT * FROM t WHERE x IN (1,2)`	1, 2
`SELECT * FROM t WHERE x IN ANY (1,2)`	1, 2
`SELECT * FROM t WHERE NOT (x NOT IN (1,2))`	1, 2
`SELECT * FROM t WHERE x NOT IN (1,2)`	3, 4, 5
`SELECT * FROM t WHERE x NOT IN ALL (1,2)`	3, 4, 5
`SELECT * FROM t WHERE 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

Purpose

Searches for or excludes nulls in an expression.

Syntax



HH01A042

where:

Syntax element ...	Specifies ...
expression	an expression that specifies a value that is tested for nulls.

ANSI Compliance

IS NULL and IS NOT NULL are ANSI SQL:2008 compliant.

Example 1

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 2

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 non-null value in the DeptNo field.

Example 3: Searching for NULL and NOT-NULL in the Same Statement

If you are searching for nulls and non-null values in the same statement, the search condition for null values 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 4: 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.

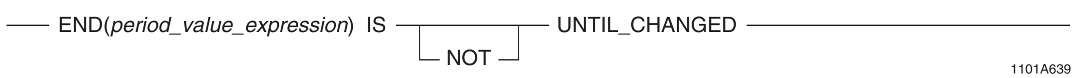
```
SELECT Name, EdLev
FROM Employee
WHERE (EdLev < 16)
OR (EdLev IS NULL) ;
```

IS UNTIL_CHANGED/IS NOT UNTIL_CHANGED

Purpose

Tests whether the ending bound of a Period value expression is (or is not) UNTIL_CHANGED.

Syntax



where:

Syntax element ...	Specifies ...
period_value_expression	any expression that evaluates to a PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE) type.
UNTIL_CHANGED	that the ending bound has a value of forever, or until it is changed. For a Period value with an element type of DATE, UNTIL_CHANGED specifies a value of DATE '9999-12-31'. For a Period value with an element type of TIMESTAMP, UNTIL_CHANGED specifies a value of TIMESTAMP '9999-12-31 23:59:59.999999' in UTC. For a Period value with an element type of TIMESTAMP WITH TIME ZONE, UNTIL_CHANGED specifies a value of TIMESTAMP '9999-12-31 23:59:59.999999+00:00'. The time zone displacement is INTERVAL '00:00' HOUR TO MINUTE.

Usage Notes

You can only compare UNTIL_CHANGED to the ending bound of a Period value with an element type of DATE or TIMESTAMP [WITH TIME ZONE]. Therefore, the result type of the END function must be DATE or TIMESTAMP [WITH TIME ZONE]. For information about the END function, see “END” on page 188.

In comparisons, the precision of the UNTIL_CHANGED value is truncated to the precision of the ending bound value being compared. That is, the number of digits after the decimal point for UNTIL_CHANGED depends upon the precision of the ending bound to which it is compared. The time zone is omitted if the ending bound value has no time zone.

If the ending bound value is NULL, IS [NOT] UNTIL_CHANGED returns UNKNOWN.

Example

Consider the following employee table, where the column *eduration* is defined as a PERIOD(DATE) type:

```
ename               eid  eduration
----------  -----------  ------------------------
Adams            210677  ('05/03/01', '06/05/21')
Gunther          199347  ('04/06/06', '99/12/31')
Montoya          199340  ('04/06/02', '99/12/31')
Chan             210427  ('04/09/24', '99/12/31')
Fuller           197899  ('03/05/27', '03/11/30')
```

The following query uses IS UNTIL_CHANGED to compare the ending bound value of the eduration column to UNTIL_CHANGED:

```
SELECT ename, eid
FROM employee
WHERE END(eduration) IS UNTIL_CHANGED;
```

The result is the following:

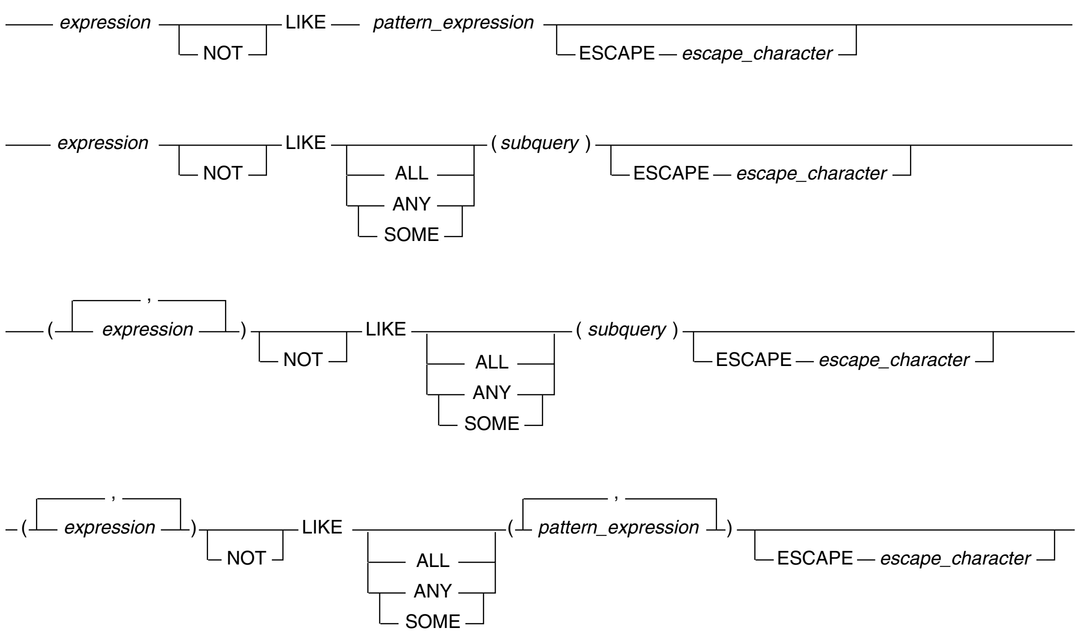
```
ename               eid
----------  -----------
Gunther          199347
Montoya          199340
Chan             210427
```

LIKE

Purpose

Searches for a character string pattern within another character string or character string expression.

Syntax



FF07D196

where:

Syntax Element ...	Specifies ...
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.
ANY ALL 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. SOME is a synonym for ANY.
subquery	a SELECT statement argument. A subquery cannot specify a SELECT AND CONSUME statement.
ESCAPE *escape_character*	keyword/variable combination specifying a single escape character (single or multibyte).

ANSI Compliance

LIKE is ANSI SQL:2008 compliant.

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 *Database Design*.

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" on page 473.

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.
	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.

Example

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%'.

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.

Example 1

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 Teradata Database 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 2

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 3

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.

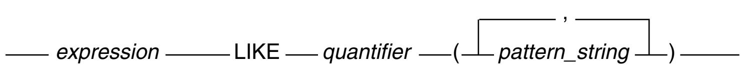
Both leading and trailing pad characters in a pattern are significant to the matching rules.

Example 4

LIKE ’ΔΔZ%’ locates only those fields that start with two pad characters followed by Z.

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.



FF07D273

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%'`

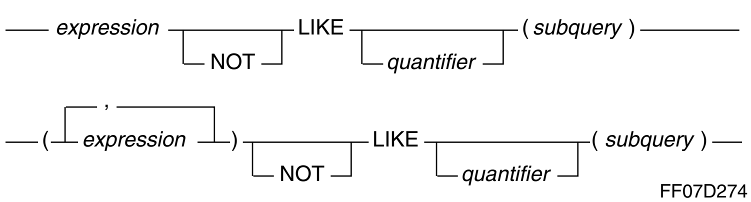
The following statement selects from the employee table the row of any employee whose job title includes the characters “Pres” or begins with the characters “Man”:

```
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.

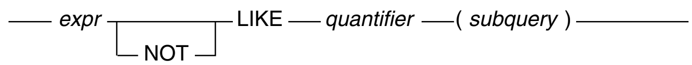
Example

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%')) ;
```

subquery

If the following form is used, the subquery might return none, one, or several values.



HH01A045

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.

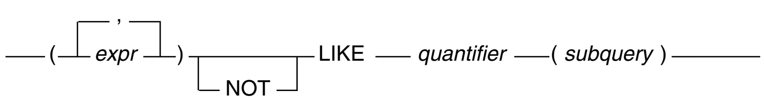
Department_Proj has two columns: Proj_pattern and Department. The rows in this table look like the following.

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.



HH01A046

For example:

```
(x,y) LIKE ALL (SELECT a,b FROM c)
```

is equivalent to:

```
(x LIKE c.a) AND (y LIKE c.b)
```

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” on page 595.

Miscellaneous Examples

Function	Result
_KanjiSJIS ‘92年abc’ LIKE _Unicode ‘%abc’	TRUE
_KanjiSJIS ‘92年abc’ LIKE _Unicode ‘**％**abc’	FALSE[a]
‘c%’ LIKE ‘c%%’ ESCAPE ‘%’	TRUE
‘c**％**’ LIKE ‘c%%’ ESCAPE ‘%’	FALSE[b]

a. ％ (FULLWIDTH PERCENT SIGN) is not a metacharacter in either KanjiSJIS or Unicode.

b. ％ (FULLWIDTH PERCENT SIGN) does not match % (PERCENT SIGN).

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

expression	*pattern_expression*	Server Character Set	Result
MN<**AB**>P	%<__**C**>%	KANJI1	FALSE

__ 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
ss$_3$**A** ss$_2$**B** ss$_3$**C** ss$_2$**D**	% ss$_2$**B**%	UNICODE	TRUE
M ss$_2$**B** **N** ss$_2$**D**	**M** __%	GRAPHIC	TRUE
ss$_3$**A** ss$_2$**B** ss$_3$**C** ss$_2$**D**	__%	KANJISJIS	TRUE
ss$_3$**A** ss$_2$**B** ss$_3$**C** ss$_2$**D**	_ %	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
mn**ABCI**	%B%	UNICODE	TRUE	TRUE
mn**ABCI**	%**I**	UNICODE	TRUE	TRUE
mn**ABCI**	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.
mn**ABCI**	mn__%**I**	KANJI1	TRUE	TRUE

__ represents a FULLWIDTH UNDERSCORE.

_ represents a SPACING UNDERSCORE.

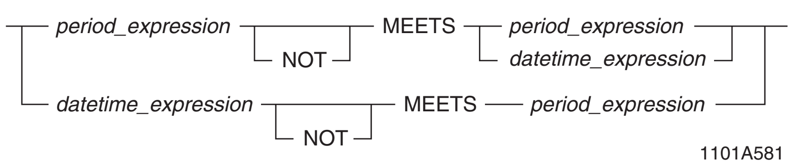
MEETS

Purpose

Predicate that operates on two Period expressions or one Period expression and one DateTime expression and evaluates to TRUE, FALSE, or UNKNOWN.

If both expressions have a Period data type, returns TRUE if the ending bound of the first expression is equal to the beginning bound of the expression or the ending bound of the second expression is equal to the beginning bound of the first expression; otherwise, returns FALSE. If one expression is a Period expression and the other expression is a DateTime expression, returns TRUE if the ending bound of the Period expression is equal to the DateTime expression or if the DateTime expression plus one granule is equal to the beginning bound of the Period expression; otherwise, returns FALSE. If either expression is NULL, the operator returns UNKNOWN.

Syntax



where:

Syntax element...	Specifies...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
period_expression	any expression that evaluates to a Period data type. **Note:** The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.

Error Conditions

If either expression evaluates to a data type other than a Period or DateTime, an error is reported.

If the expressions are not comparable, an error is reported.

Example

In the following example, the MEETS operator is used in the WHERE clause.

```
SELECT * FROM employee WHERE period2 MEETS period1;
```

Assume the query is executed on the following table employee where period1 and period2 are PERIOD(DATE) columns:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is as follows:

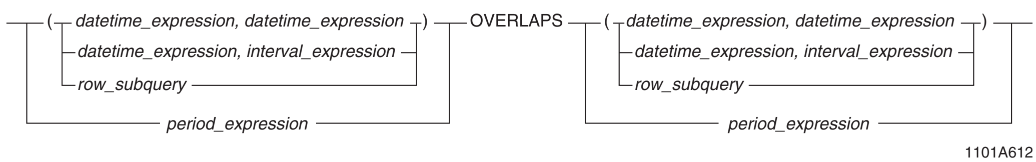
ename	period1	period2
Jones	('2004-01-02','2004-03-05')	('2004-03-05', '2004-10-07')

OVERLAPS

Purpose

Tests whether two time periods overlap one another.

Syntax



where:

Syntax element ...	Specifies ...
datetime_expression	a start and end DateTime.
interval_expression	an end DateTime.
row_subquery	an element of a row subquery in a SELECT statement. The subquery cannot specify a SELECT AND CONSUME statement.
period_expression	any expression that evaluates to a Period data type.

ANSI Compliance

OVERLAPS is ANSI SQL:2008 compliant.

Time Periods

Each time period to the left and right of the OVERLAPS keyword is one of the following expression types:

- DateTime, DateTime
- DateTime, Interval
- Row subquery
- Period

Each time period represents a start and end DateTime, using an explicit Period value, DateTime values or a DateTime and an Interval.

If the start and end DateTime values in a time period are not ordered chronologically, they are manipulated to make them so prior to making the comparison, using the rule that *end_DateTime* >= *start_DateTime* for all cases.

If a time period contains a null *start_DateTime* and a non-null *end_DateTime*, then the values are switched to indicate a non-null *start_DateTime* and a null *end_DateTime*.

If both time periods have a Period data type, the data types must be comparable. If only one time period is a Period type, the other time period must evaluate to a DateTime type that is comparable to the element type of the Period.

Note: Implicit casting to a Period data type is not supported.

Results

Consider the general case of an OVERLAPS comparison, stated as follows.

```
(S1, E1) OVERLAPS (S2, E2)
```

The result of OVERLAPS is as follows.

```
(S1 > S2 AND NOT (S1 >= E2 AND E1 >= E2))
OR
(S2 > S1 AND NOT (S2 >= E1 AND E2 >= E1))
OR
(S1 = S2 AND (E1 = E2 OR E1 <> E2))
```

For Period data types, where p1 is the first Period expression and p2 is the second Period expression, the values of S1, E1, S2, and E2 are as follows:

```
S1 = BEGIN(p1)
E1 = END(p1)
S2 = BEGIN(p2)
E2 = END(p2)
```

Rules

The following rules apply to the OVERLAPS comparison.

- When you specify two DateTime types, they must be comparable.
- When you specify two Period types, they must be comparable.
- If the first columns of each left and right time periods are DateTime types, they must have the same data type: both DATE, both TIME, or both TIMESTAMP.
- If only one time period is a Period type, the first column of the other time period must have the same data type as the element type of the Period.
- If neither time period is a Period type, then the second column of each left and right time period must either be the same DateTime type as its corresponding first column (that is, the two types must be compatible) or it must be an Interval type that involves only DateTime fields where the precision is such that its value can be added to that of the corresponding DateTime type.

Example 1

The following example compares two time spans that share a single common point, CURRENT_TIME.

The result returned is FALSE because when two time spans share a single point, they do not overlap by definition.

```
SELECT 'OVERLAPS'
WHERE (CURRENT_TIME(0), INTERVAL '1' HOUR)
OVERLAPS (CURRENT_TIME(0), INTERVAL -'1' HOUR);
```

Example 2

The following example is nearly identical to the previous one, except that the arguments have been adjusted to overlap by one second. The result is TRUE and the value 'OVERLAPS' is returned.

```
SELECT 'OVERLAPS'
WHERE (CURRENT_TIME(0), INTERVAL '1' HOUR)
OVERLAPS (CURRENT_TIME(0) + INTERVAL '1' SECOND,INTERVAL -'1' HOUR);
```

Example 3

Here is an example that uses the *datetime_expression, datetime_expression* form of OVERLAPS. The two DATE periods overlap each other, so the result is TRUE.

```
SELECT 'OVERLAPS'
WHERE (DATE '2000-01-15',DATE '2002-12-15')
OVERLAPS (DATE '2001-06-15',DATE '2005-06-15');
```

Example 4

The following example is the same as the previous one, but in *row_subquery* form:

```
SELECT 'OVERLAPS'
WHERE (SELECT DATE '2000-01-15', DATE '2002-12-15')
OVERLAPS (SELECT DATE '2001-06-15', DATE '2005-06-15');
```

Example 5

The null value in the following example means the second *datetime_expression* has a start time of 2001-06-13 15:00:00 and a null end time.

```
   SELECT 'OVERLAPS'
   WHERE (TIMESTAMP '2001-06-12 10:00:00', TIMESTAMP '2001-06-15
08:00:00')
   OVERLAPS (TIMESTAMP '2001-06-13 15:00:00', NULL);
```

Because the start time for the second expression falls within the TIMESTAMP interval defined by the first expression, the result is TRUE.

Example 6

In the following example, the OVERLAPS predicate operates on PERIOD(DATE) columns.

```
SELECT * FROM employee WHERE period2 OVERLAPS period1;
```

Assume the query is executed on the following table employee; where *period1* and *period2* are PERIOD(DATE) columns:

Ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is as follows:

Ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')

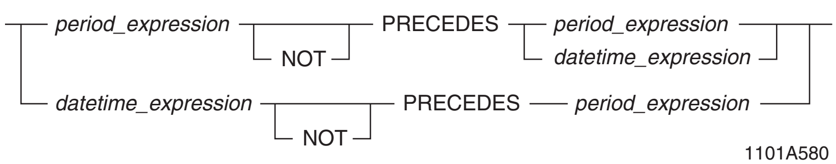
PRECEDES

Purpose

Predicate that operates on two Period expressions or one Period expression and one DateTime expression and evaluates to TRUE, FALSE, or UNKNOWN.

If both expressions have a Period data type, returns TRUE if the ending bound of the first expression is less than or equal to the beginning bound of the second expression; otherwise, returns FALSE. If the first expression is a Period expression and the second expression is a DateTime expression, returns TRUE if the ending bound of the first expression is less than or equal to the second expression; otherwise, returns FALSE. If the first expression is a DateTime value expression and the second expression has a Period data type, returns TRUE if the first expression is less than the beginning bound of the second expression; otherwise, returns FALSE. If either expression is NULL, the operator returns UNKNOWN.

Syntax



where:

Syntax element...	Specifies...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
period_expression	any expression that evaluates to a Period data type. **Note:** The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.

Error Conditions

If either expression is other than a Period data type or a DateTime value expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

In the following example, the PRECEDES operator is used in the WHERE clause.

```
SELECT * FROM employee WHERE period1 PRECEDES period2;
```

Assume the query is executed on the following table employee where period1 and period2 are PERIOD(DATE) columns:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is as follows:

ename	period1	period2
Jones	('2004-01-02','2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02','2004-03-05')	('2004-03-07', '2004-10-07')

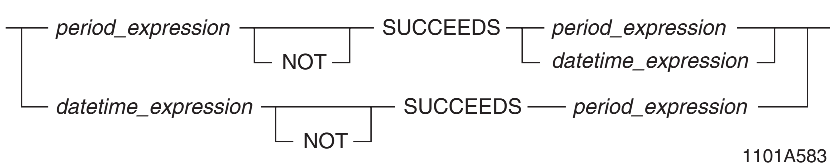
SUCCEEDS

Purpose

Predicate that operates on two Period expressions or one Period expression and one DateTime expression and evaluates to TRUE, FALSE, or UNKNOWN.

If both expressions have a Period data type, returns TRUE if the beginning bound of the first expression is greater than or equal to the ending bound of the second expression; otherwise, returns FALSE. If the first expression is a Period expression and the second expression is a DateTime expression, returns TRUE if the beginning bound of the first expression is greater than the second expression; otherwise, returns FALSE. If the first expression is a DateTime expression and the second expression is a Period expression, returns TRUE if the DateTime expression is greater than or equal to the ending bound of the second expression; otherwise, returns FALSE. If either expression is NULL, the operator returns UNKNOWN.

Syntax



where:

Syntax element...	Specifies...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
period_expression	any expression that evaluates to a Period data type. **Note:** The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.

Error Conditions

If either expression is other than a Period data type or a DateTime value expression, an error is reported.

If the expressions are not comparable types, an error is reported.

Example

In the following example, the SUCCEEDS operator is used in the WHERE clause.

```
SELECT * FROM employee WHERE period1 SUCCEEDS period2;
```

Assume the query is executed on the following table employee where period1 and period2 are PERIOD(DATE) columns:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is as follows:

ename	period1	period2
Jones	('2004-01-02','2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02','2004-03-05')	('2004-03-07', '2004-10-07')

Logical Operators and Search Conditions

Purpose

Specify the criteria for logically producing the result of a search condition.

Definition: Logical Operator

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* ——

FF07D220

Use NOT to negate an expression, for example:

—— *expression_1* —— AND NOT —— *expression_2* ——

FF07D221

Definition: Search Condition

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
- [NOT] BETWEEN
- LIKE
- [NOT] IN
- ALL or ANY/SOME
- [NOT] EXISTS
- OVERLAPS
- IS [NOT] NULL

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:

  1. NOT
  2. AND
  3. 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.

Example 1

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') ;
```

Example 2

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
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

CHAPTER 12 Attribute Functions

This chapter describes SQL attribute functions.

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 *SQL Data Types and Literals*.

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:2008 standard.

To maintain ANSI compatibility, use the ANSI equivalent functions instead of Teradata attribute functions, when available.

Change this Teradata function ...	To this ANSI function in new applications ...
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:

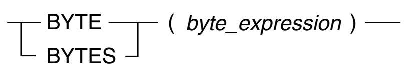
- BYTES
- FORMAT
- TYPE

BYTES

Purpose

Returns the number of bytes contained in the specified byte string.

Syntax



1101F174

where:

Syntax element ...	Specifies ...
byte_expression	the byte string for which the number of bytes is to be returned.

ANSI Compliance

BYTES is a Teradata extension to the ANSI SQL:2008 standard.

Argument Types

The data types of *byte_expression* are restricted to the following:

- 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 *SQL Data Definition Language*.

  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. For details, see *Utilities*.

  For more information on implicit type conversion of UDTs, see Chapter 17: “Data Type Conversions.”

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;
```

For more information on TRIM, see "TRIM" on page 420.

Example

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:

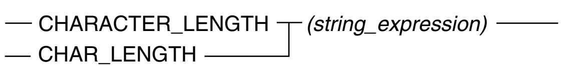
```
BadgePic        Bytes(BadgePic)
--------------  ---------------
20003BA0                      4
9A3243F805                    5
EEFF08C3441900                7
```

CHARACTER_LENGTH

Purpose

Returns the length of a string either in logical characters or in bytes.

Syntax



FF07D088

where:

Syntax element ...	Specifies ...
string_expression	the string expression for which the length is to be returned.

ANSI Compliance

CHARACTER_LENGTH is ANSI SQL:2008 compliant.

Usage Notes

CHARACTER_LENGTH is the ANSI form of the Teradata CHARACTERS function. Use CHARACTER_LENGTH instead of CHARACTERS for ANSI SQL:2008 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.)

Argument Types

The type of *string_expression* must be CHARACTER, VARCHAR, or CLOB. For non-character data types, the function returns an error.

By default, Teradata Database 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. For more information on CREATE CAST, see *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

Result

For all server character sets except KANJI1, CHARACTER_LENGTH returns the length of *string_expression* in characters.

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.

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.

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;
```

Example

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

Example Set 2: KanjiShift-JIS

FOR this server character set ...	AND example ...	CHARACTER_LENGTH returns ...
KANJI1	<><>	10
	DeF	3
UNICODE	ABC	3
GRAPHIC	ABC	3

Example Set 3: KanjiEUC

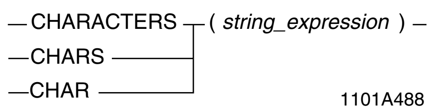
FOR this server character set ...	AND example ...	CHARACTER_LENGTH returns ...
KANJI1	ss$_3$**C**ss$_3$**D**	2
GRAPHIC		2
UNICODE	<><>	0
	d**A** ss$_2$**B** ss$_3$**E**	4
LATIN	ABC	3

CHARACTERS

Purpose

Returns an integer value representing the number of logical characters or bytes contained in the specified operand string.

Syntax



where:

Syntax element ...	Specifies ...
string_expression	a character (single byte, multibyte, mixed single byte and multibyte) string for which the number of characters is to be returned. The data types for *string_expression* are restricted to CHARACTER, VARCHAR, and CLOB.

ANSI Compliance

CHARACTERS is a Teradata extension to the ANSI SQL-99standard.

Value Returned by CHARACTERS and Server Character Set

Because CHARACTERS returns the number of logical characters or bytes in *string_expression*, the value differs depending on the server character set of *string_expression*. The following table illustrates the differences among the various character sets for a CHARACTER(12) column.

FOR this server character set ...	The length of *string_expression* ...
• UNICODE • LATIN • GRAPHIC	is always 12. Unicode, Latin, and Graphic are fixed width character types.
• KANJISJIS • KANJI1	varies depending on the mix of characters (multibyte and single byte) in the string. KanjiSJIS and KANJI1 are variable width character sets.

CHARACTER_LENGTH versus CHARACTERS

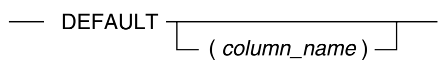
Use of the CHARACTERS function is deprecated. Instead, use the ANSI-equivalent "CHARACTER_LENGTH."

DEFAULT

Purpose

Returns the current default value for the specified or derived column.

Syntax



1101A394

where:

Syntax element …	Specifies …
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.

ANSI Compliance

DEFAULT is partially ANSI SQL:2008 compliant.

The form of DEFAULT that specifies a column name is a Teradata extension. Using DEFAULT in a predicate is also a Teradata extension.

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 *SQL Data Types and Literals*.

Result Value

The DEFAULT function returns the default value of the specified column or derived column (if the 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 3: Specifying a View Column Name" on page 506.
- 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 constant expression

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, Teradata Database 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 Teradata Database 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 2: Using DEFAULT in a Predicate" on page 505.

For details on using the DEFAULT function in INSERT, UPDATE, and MERGE statements, see *SQL Data Manipulation Language*.

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);
```

Restrictions

The DEFAULT function cannot be used as a partitioning expression for defining PPIs.

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 Teradata Database 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);
  ```

Example 1: 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 2: 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, Teradata Database 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 3: 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 Topics

For information on ...	See ...
using predicates	Chapter 11: “Logical Predicates.”
comparison operations in predicates	Chapter 4: “Comparison Operators.”
the DEFAULT value control phrase	*SQL Data Types and Literals*.
INSERT, UPDATE, and MERGE statements	*SQL Data Manipulation Language*.

FORMAT

Purpose

Returns the declared format for the named expression.

Syntax

```
— FORMAT — ( column_name ) —
```

1101A489

where:

Syntax element ...	Specifies ...
expression	the expression for which the FORMAT is to be reported.

ANSI Compliance

FORMAT is a Teradata extension to the ANSI SQL:2008 standard.

Result Type

FORMAT returns a CHAR(*n*) character string of up to 30 characters.

Example

The following statement requests the format of the Salary column in the Employee table.

```
SELECT FORMAT(Employee.Salary);
```

The result is the following.

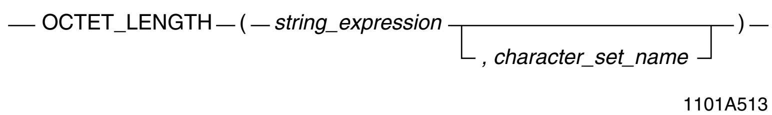
```
Format(Salary)
------------------------------
ZZZ,ZZ9.99
```

OCTET_LENGTH

Purpose

Returns the length of *string_expression* in octets when it is converted to the named character set (taking the export width value into consideration).

Syntax



where:

Syntax element ...	Specifies ...
string_expression	the character string for which the number of octets is required.
character_set_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. See the list of Teradata-provided character sets in the table on "Usage Notes" on page 509.

ANSI Compliance

OCTET_LENGTH is ANSI SQL:2008 compliant.

Argument Types

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. For more information on CREATE CAST, see *SQL Data Definition Language*.

  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. For details, see *Utilities*.

  For more information on implicit type conversion of UDTs, see Chapter 17: "Data Type Conversions."

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.

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		Where Found
• ASCII • EBCDIC	• UTF8 • UTF16	Built-in
• ARABIC1256_6A0 [a] • CYRILLIC1251_2A0 [a] • EBCDIC037_0E • EBCDIC273_0E • EBCDIC277_0E • HANGUL949_7R0 [a] • HANGULEBCDIC933_1II • HANGULKSC5601_2R4 • HEBREW1255_5A0 [a] • KANJI932_1S0 [a] • KANJIEBCDIC5026_0I • KANJIEBCDIC5035_0I • KANJIEUC_0U • KANJISJIS_0S • KATAKANAEBCDIC	• LATIN1250_1A0 [a] • LATIN1252_0A • LATIN1252_3A0 [a] • LATIN1254_7A0 [a] • LATIN1258_8A0 [a] • LATIN1_0A • LATIN9_0A • SCHEBCDIC935_2IJ • SCHGB2312_1T0 • SCHINESE936_6R0 [a] • TCHBIG5_1R0 • TCHEBCDIC937_3IB • TCHINESE950_8R0 [a] • THAI874_4A0 [a]	DBC.CharTranslationsV

a. Windows code page compatible session character set

Examples

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** ss$_2$**B** ss$_3$**E**	8
KanjiShift-JIS	KANJISJIS	**DeF**	5
KanjiShift-JIS	UNICODE	ABC	6

TITLE

Purpose

Returns the title of an expression as it would appear in the heading for displayed or printed results.

Syntax

── TITLE ─ (*expression*) ─

1101B039

where:

Syntax element ...	Specifies ...
expression	the expression for which the title is to be returned.

ANSI Compliance

TITLE is a Teradata extension to the ANSI SQL:2008 standard.

Result Type

TITLE returns a CHAR(*n*) character string of up to 60 characters.

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.

For more information, see *SQL Data Types and Literals*.

Example

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
```

TYPE

Purpose

Returns the data type defined for an expression.

Syntax

```
—TYPE —( expression ) —
```

1101A491

where:

Syntax element ...	Specifies ...
expression	the expression for which the data type is to be returned.

ANSI Compliance

TYPE is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Value

TYPE returns a CHAR(*n*) character string that contains the name of the data type of the expression.

For a list of the supported data types, see *SQL Data Types and Literals.* For information on geospatial types, see *SQL Geospatial Types*.

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.

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.

For examples, see "Example 1" and "Example 2" on page 513.

Example 1

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 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

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

If TYPE is used to request the data type of two columns, defined as GRAPHIC and LONG VARGRAPHIC, 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 VARGRAPHIC column, the length returned is the maximum length of 32000.

Example 5

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

Consider the following table definition:

```
CREATE TABLE images
  (imageid INTEGER
  ,imagedesc VARCHAR(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.

CHAPTER 13 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

Features

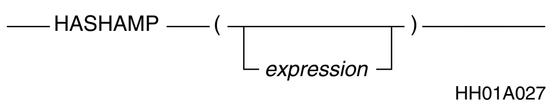
Use the hash-related functions to identify the statistical properties of the current primary index or secondary index, or to evaluate these properties for other columns to determine their suitability as a future primary index or secondary index. The statistics can help you to minimize hash synonyms and enhance the uniformity of data distribution.

HASHAMP

Purpose

Returns the identification number of the primary AMP corresponding to the specified hash bucket number. If no hash bucket number is specified, HASHAMP returns one less than the maximum number of AMPs in the system.

Syntax



where:

Syntax element ...	Specifies ...
expression	an optional expression that evaluates to a valid hash bucket number. For information on obtaining a hash bucket number that you can use for *expression*, see "HASHBUCKET" on page 522.

ANSI Compliance

HASHAMP is a Teradata extension to the ANSI SQL:2008 standard.

Argument Type and Value

The *expression* argument must evaluate to INTEGER data type where the valid range of values depends on the system setting for the hash bucket size.

IF the hash bucket size is ...	THEN the range of values for *expression* is ...
16 bits	0 to 65535.
20 bits	0 to 1048575.

For information on how to specify the system setting for the hash bucket size, see "DBS Control utility" in *Utilities*.

If *expression* cannot be implicitly converted to an INTEGER, an error is reported.

If *expression* results in a UDT, Teradata Database performs implicit type conversion on the UDT, provided that 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 on CREATE CAST, see *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 17: "Data Type Conversions."

Result

IF *expression* ...	THEN ...
evaluates to a valid hash bucket number	HASHAMP determines the primary AMP corresponding to the hash bucket and returns the AMP identification number. The result is an INTEGER value that is greater than or equal to zero and less than the maximum number of AMPs in the configuration.
does not appear in the argument list	HASHAMP returns an INTEGER value that is one less than the maximum number of AMPs in the system.
evaluates to NULL	HASHAMP returns NULL.

For information on the hash map that defines the relationship between hash buckets and primary AMPs, see "Reconfiguration utility" in the *Utilities* book.

Examples

The following 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.

```
CREATE TABLE T
   (column_1 INTEGER
   ,column_2 INTEGER
   ,B INTEGER)
UNIQUE PRIMARY INDEX (column_1, column_2);
```

Example 1

If you call HASHAMP without an argument, it returns one less than the maximum number of AMPs on the system.

```
SELECT HASHAMP();
```

Example 2

If you call HASHAMP with an argument of NULL, it returns NULL.

```
SELECT HASHAMP(NULL);
```

Example 3

The following query returns the distribution of the hash buckets among the primary AMPs.

```
SELECT B, HASHAMP (B)
FROM T
ORDER BY 1;
```

Example 4

The following query returns the number of rows on each primary AMP where column_1 and column_2 are to be the primary index of table T.

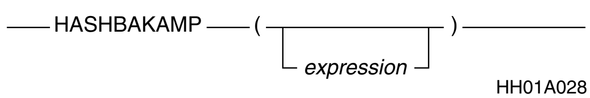
```
SELECT HASHAMP (HASHBUCKET (HASHROW (column_1,column_2))), COUNT (*)
FROM T
GROUP BY 1
ORDER BY 1;
```

HASHBAKAMP

Purpose

Returns the identification number of the fallback AMP corresponding to the specified hash bucket. If no hash bucket is specified, HASHBAKAMP returns one less than the maximum number of AMPs in the system.

Syntax



where:

Syntax element ...	Specifies ...
expression	an optional expression that evaluates to a valid hash bucket number. For information on obtaining a hash bucket number that you can use for *expression*, see "HASHBUCKET" on page 522.

ANSI Compliance

HASHBAKAMP is a Teradata extension to the ANSI SQL:2008 standard.

Argument Type and Value

The *expression* argument must evaluate to INTEGER data type where the valid range of values depends on the system setting for the hash bucket size.

IF the hash bucket size is ...	THEN the range of values for *expression* is ...
16 bits	0 to 65535.
20 bits	0 to 1048575.

For information on how to specify the system setting for the hash bucket size, see "DBS Control utility" in *Utilities*.

If *expression* cannot be implicitly converted to an INTEGER, an error is reported.

If *expression* results in a UDT, Teradata Database performs implicit type conversion on the UDT, provided that 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 on CREATE CAST, see *SQL Data Definition Language*.

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. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 17: "Data Type Conversions."

Result

IF *expression* ...	THEN ...
does not appear in the argument list	HASHBAKAMP returns an INTEGER value that is one less than the maximum number of AMPs in the system.
evaluates to NULL	HASHBAKAMP returns NULL.
evaluates to a valid hash bucket number	HASHBAKAMP determines the fallback AMP corresponding to the hash bucket and returns the identification number of the AMP. The result is an INTEGER value that is greater than or equal to zero and less than the maximum number of AMPs in the configuration.

For information on the hash map that defines the relationship between hash buckets and fallback AMPs, see "Reconfiguration utility" in the *Utilities* book.

Examples

The following examples assume 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 1

If you call HASHBAKAMP without an argument, it returns one less than the maximum number of AMPs on the system.

```
SELECT HASHBAKAMP();
```

Example 2

If you call a HASHBAKAMP function with an argument of NULL, the function returns NULL.

```
SELECT HASHBAKAMP(NULL);
```

Example 3

This query returns the distribution of the hash buckets among the fallback AMPs.

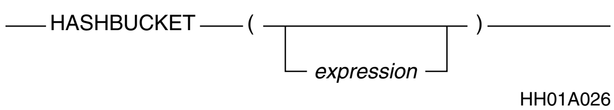
```
SELECT B, HASHBAKAMP (B)
FROM T
ORDER BY 1;
```

HASHBUCKET

Purpose

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.

Syntax



where:

Syntax element ...	Specifies ...
expression	an optional expression that evaluates to a valid BYTE(4) row hash value. If *expression* results in a UDT, Teradata Database 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 *SQL Data Definition Language*. 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. For details, see *Utilities*. For more information on implicit type conversion, see Chapter 17: "Data Type Conversions." For information on obtaining a row hash value that you can use for *expression*, see "HASHROW" on page 525.

ANSI Compliance

HASHBUCKET is a Teradata extension to the ANSI SQL:2008 standard.

Result

HASHBUCKET returns an INTEGER data type.

<table>
<tr><th>IF expression ...</th><th>THEN ...</th></tr>
<tr><td>does not appear in the argument list</td><td>HASHBUCKET returns an INTEGER value that is the highest hash bucket number.</td></tr>
<tr><td>evaluates to NULL</td><td>HASHBUCKET returns NULL.</td></tr>
<tr><td>evaluates to a valid BYTE(4) row hash value</td><td>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.
<table>
<tr><th>IF the hash bucket size is ...</th><th>THEN hash bucket numbers can have a value from ...</th></tr>
<tr><td>16 bits</td><td>0 to 65535.</td></tr>
<tr><td>20 bits</td><td>0 to 1048575.</td></tr>
</table></td></tr>
</table>

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

The following examples assume a table T with columns C1 and C2 and possibly other columns.

Example 1

If you call HASHBUCKET without an argument, it returns the maximum hash bucket.

```
SELECT HASHBUCKET();
```

Example 2

If you call a HASHBUCKET function with an argument of NULL, the function returns NULL.

```
SELECT HASHBUCKET(NULL);
```

Example 3

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 4

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 5

The results of the following example lists each hash bucket that has one or more rows and its corresponding primary AMP.

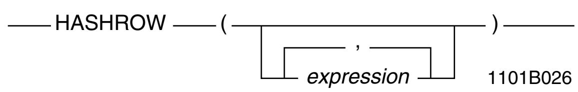
```
SELECT HASHAMP (HASHBUCKET (HASHROW (C1, C2))),
   HASHBUCKET (HASHROW (C1,C2))
   FROM T
   GROUP BY 1,2
   ORDER BY 1,2 ;
```

HASHROW

Purpose

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.

Syntax



where:

Syntax element ...	Specifies ...
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.

ANSI Compliance

HASHROW is a Teradata extension to the ANSI SQL:2008 standard.

Result

The resulting row hash value is typed BYTE(4).

IF the argument list 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	

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. For information on HASHBUCKET, see "HASHBUCKET" on page 522.

Example 1

If you call HASHROW without an argument, it returns 'FFFFFFFF'XB, which is the maximum hash code in the system.

```
SELECT HASHROW();
```

Example 2

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 3

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.

CHAPTER 14 Built-In Functions

Built-in functions, which are niladic (have no arguments), return various information about the system. Built-in functions are sometimes referred to as special registers.

The built-in functions can be used anywhere that a constant 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

Purpose

Returns the account string for the current user.

Syntax

```
—— ACCOUNT ——
```

FF07R001

ANSI Compliance

ACCOUNT is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

The data type and format for ACCOUNT are as follows:

Data Type	Format
VARCHAR(30) CHARACTER SET UNICODE	X(30)

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

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

Purpose

Returns the current system date.

Syntax

```
——CURRENT_DATE ——
```

FF07D135

ANSI Compliance

CURRENT_DATE is ANSI SQL:2008 compliant.

Result Type and Attributes

The data type and format for CURRENT_DATE are as follows:

Data Type	Format
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 *SQL Data Types and Literals*.

To convert CURRENT_DATE, 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 2: Changing the Default Output Format" on page 534.

CURRENT_DATE versus DATE

CURRENT_DATE provides similar functionality to the Teradata function DATE using ANSI-compliant syntax. For information on the Teradata DATE function, see "DATE" on page 539.

Example 1: Requesting the Current System Date

The following statement requests the current system date:

```
SELECT CURRENT_DATE;
```

The system responds with something like the following:

```
    Date
--------
01/12/28
```

Example 2: 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 time and specifies a format that is different from the default:

```
SELECT CURRENT_DATE (FORMAT 'MMMBDD,BYYYY');
```

The result looks like this:

```
        Date
------------
May 31, 2007
```

For more information on Teradata explicit conversion syntax, see "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 585. For more information on default data type formats and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

CURRENT_ROLE

Purpose

Returns the current role of the current authorized user.

Syntax

```
—— CURRENT_ROLE ——
                          1101A565
```

ANSI Compliance

CURRENT_ROLE is consistent with ANSI SQL:2008 usage.

Result Type and Attributes

The data type and format for CURRENT_ROLE are as follows:

Data Type	Format
VARCHAR(30) CHARACTER SET UNICODE	X(30)

Result Value

If you are not accessing the Teradata Database 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 details, see "ROLE" on page 542.

If you are accessing the Teradata Database 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

Usage Notes

CURRENT_ROLE is not supported in the FastLoad and MultiLoad utilities.

Example

You can identify the current role for the current authorized user with the following statement:

```
SELECT CURRENT_ROLE;
```

The system responds with something like the following:

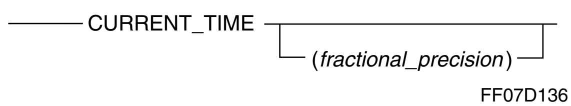
```
Current_Role
------------------------------
Buyers_role
```

CURRENT_TIME

Purpose

Returns the current system time and current session Time Zone displacement.

Syntax



where:

Syntax element ...	Specifies ...
fractional_precision	an optional precision range for the returned time value. The valid range is 0 through 6, inclusive. The default is 0.

ANSI Compliance

CURRENT_TIME is ANSI SQL:2008 compliant.

Result Type and Attributes

The data type and format for CURRENT_TIME are as follows:

Data Type	Format
TIME WITH TIME ZONE	Default format for the TIME WITH TIME ZONE data type. For more information on the default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

To convert CURRENT_TIME, 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 2: Changing the Default Output Format" on page 534.

CURRENT_TIME Fields

The fields in CURRENT_TIME are:

- HOUR
- MINUTE

- SECOND
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

CURRENT_TIME versus TIME

CURRENT_TIME provides similar functionality to the Teradata function TIME using ANSI-compliant syntax. For information on the Teradata TIME function, see “TIME” on page 546.

Precision

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.

Example 1: Requesting the Current System Time

The following statement requests the current system time and current session Time Zone displacement:

```
SELECT CURRENT_TIME;
```

The system responds with something like the following:

```
Current Time(0)
---------------
 15:53:34+00:00
```

Example 2: 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
```

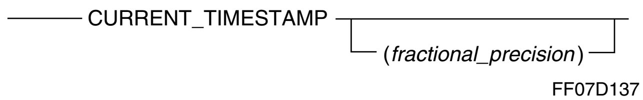
For more information on Teradata explicit conversion syntax, see “Teradata Conversion Syntax in Explicit Data Type Conversions” on page 585. For more information on default data type formats and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

CURRENT_TIMESTAMP

Purpose

Returns the current system timestamp and current session Time Zone displacement.

Syntax



where:

Syntax element ...	Specifies ...
fractional_precision	an optional precision range for the returned timestamp value. The valid range is 0 through 6, inclusive. The default is 6.

ANSI Compliance

CURRENT_TIMESTAMP is ANSI SQL:2008 compliant.

Result Type and Attributes

The data type and format for CURRENT_TIMESTAMP are as follows:

Data Type	Format
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 *SQL Data Types and Literals*.

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 2: Changing the Default Output Format” on page 536.

Precision

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.

CURRENT_TIMESTAMP Fields

The fields in CURRENT_TIMESTAMP are:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

Example 1: Requesting the Current System Timestamp

The following statement requests the system timestamp and session Time Zone displacement:

```
SELECT CURRENT_TIMESTAMP;
```

The system responds with something like the following:

```
            Current TimeStamp(6)
--------------------------------
2001-11-27 15:53:34.910000+00:00
```

Example 2: 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
```

For more information on Teradata explicit conversion syntax, see "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 585. For more information on default data type formats and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

CURRENT_USER

Purpose

Provides the user name of the current authorized user.

Syntax

```
—— CURRENT_USER ——
                    1101A564
```

ANSI Compliance

CURRENT_USER is consistent with ANSI SQL:2008 usage.

Result Type and Attributes

The data type and format for CURRENT_USER are as follows:

Data Type	Format
VARCHAR(30) CHARACTER SET UNICODE	X(30)

Result Value

If you are accessing the Teradata Database 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 details, see "USER" on page 548.

Example 1

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 2

The following example selects the job title for the current authorized user:

```
SELECT JobTitle FROM Employee WHERE Name = CURRENT_USER;
```

DATABASE

Purpose

Returns the name of the default database for the current user.

Syntax

```
——— DATABASE ———
```

FF07R002

ANSI Compliance

DATABASE is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

The data type and format for DATABASE are as follows:

Data Type	Format
VARCHAR(30) CHARACTER SET UNICODE	X(30)

Usage Notes

If a DATABASE request has changed the current default database, then the DATABASE function returns the new name of the default.

Example

The following statement requests the name of the default database:

```
SELECT DATABASE;
```

The system responds with something like the following:

```
Database
------------------------------
Customer_Service
```

DATE

Purpose

Returns the current date.

Syntax

```
—— DATE ——
```

FF07D134

Result Type and Attributes

<table>
<tr><th>Data Type</th><th>FORMAT</th></tr>
<tr><td>DATE</td><td>The default format of DATE depends on the value of the Dateform mode.
<table>
<tr><th>IF the value of the Dateform mode is …</th><th>THEN the format of the DATE function is …</th></tr>
<tr><td>INTEGERDATE</td><td>the default format for DATE data types as specified in the SDF.</td></tr>
<tr><td>ANSIDATE</td><td>'YYYY-MM-DD'</td></tr>
</table>
For more information on default data type formats, see “Data Type Formats and Format Phrases” in SQL Data Types and Literals.</td></tr>
</table>

ANSI Compliance

DATE is a Teradata extension to the ANSI SQL:2008 standard.

For the ANSI-compliant syntax and behavior for the equivalent function, see “CURRENT_DATE” on page 529.

Usage Notes

DATE is deprecated. Use the ANSI SQL:2008 compliant CURRENT_DATE function instead.

DATE cannot appear as the first argument in a user-defined method invocation.

Example 1

The following example selects the current date:

```
SELECT DATE;
```

The system returns:

```
    Date
--------
96/03/30
```

Example 2

Use the FORMAT phrase to change the presentation:

```
SELECT DATE (FORMAT 'mm-dd-yy') ;
    Date
--------
03-30-96
```

Example 3

Another form gives:

```
SELECT DATE (FORMAT 'mmmbdd,byyyy') ;
        Date
------------
Mar 30, 1996
```

PROFILE

Purpose

Returns the current profile for the session or NULL if none.

Syntax

```
—— PROFILE ——
```

KZ01A006

ANSI Compliance

PROFILE is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

The data type and format for PROFILE are as follows:

Data Type	Format
VARCHAR(30) CHARACTER SET UNICODE	X(30)

Example

You can identify the current profile for the session with the following statement:

```
SELECT PROFILE ;
```

ROLE

Purpose

Returns the session current role.

Syntax

```
—— ROLE ——
```

KZ01A007

ANSI Compliance

ROLE is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

The data type and format for ROLE are as follows:

Data Type	Format
VARCHAR(30) CHARACTER SET UNICODE	X(30)

Result Value

<table>
<tr><th>IF the session logon is ...</th><th>THEN ...</th></tr>
<tr><td>not directory-based</td><td>
<table>
<tr><th>IF the current role for the session is ...</th><th>THEN the result value is ...</th></tr>
<tr><td>an existing role</td><td>the name of the role.</td></tr>
<tr><td>ALL</td><td>'ALL'.</td></tr>
<tr><td>NONE or NULL</td><td>NULL.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>IF the session logon is ...</th><th>THEN ...</th></tr>
<tr><td>directory-based</td><td>
<table>
<tr><th>IF the session ...</th><th>THEN the result value is ...</th></tr>
<tr><td>is assigned a set of directory-managed roles and does not change the current role</td><td rowspan="2">'EXTERNAL'.</td></tr>
<tr><td>uses a SET ROLE EXTERNAL statement</td></tr>
<tr><td>• 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</td><td>the name of the default role of the permanent user.</td></tr>
<tr><td>uses a SET ROLE role_name statement, where role_name is either a directory-managed or database-managed role</td><td>the name of the specified role.</td></tr>
<tr><td>uses a SET ROLE ALL statement</td><td>'ALL'.</td></tr>
<tr><td>• is not assigned a set of directory-managed roles,
• does not change the current role, and
• one of the following conditions is true:
  • the directory-based logon does not map to a permanent user
  • the permanent user that the directory-based logon maps to does not have a default database-managed role</td><td rowspan="3">NULL.</td></tr>
<tr><td>uses a SET ROLE NONE statement</td></tr>
<tr><td>uses a SET ROLE NULL statement</td></tr>
</table>
</td></tr>
</table>

If you are accessing the Teradata Database through a proxy connection, and you want to get the current role of the proxy user, use the CURRENT_ROLE built-in function. For details, see "CURRENT_ROLE" on page 531.

Usage Notes

ROLE is not supported in the FastLoad and MultiLoad utilities.

Example

You can identify the session current role with the following statement:

```
SELECT ROLE;
```

The system responds with something like the following:

```
Role
```

```
------------------------------
EXTERNAL
```

SESSION

Purpose

Returns the number of the session for the current user.

Syntax

```
—— SESSION ——
```

FF07R003

ANSI Compliance

SESSION is a Teradata extension to the ANSI SQL:2008 standard.

Result Type and Attributes

The data type and format for SESSION are as follows:

Data Type	Format
INTEGER	Default format for the INTEGER data type. For more information on the default formats, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Example

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
```

TIME

Purpose

Provides the current system time based on a 24-hour day.

Syntax

```
—— TIME ——
```

FF07D271

ANSI Compliance

TIME is a Teradata extension to the ANSI SQL:2008 standard.

For the ANSI SQL:2008 compliant syntax and behavior for the equivalent function, see "CURRENT_TIME" on page 533.

Result Type and Attributes

The data type and format for TIME are as follows:

Data Type	Format
FLOAT	HHMMSS.CC (hours, minutes, seconds, hundredths of a second)

Usage Notes

TIME is deprecated. Use the ANSI SQL:2008 compliant CURRENT_TIME function instead.

TIME cannot appear as the first argument in a user-defined method invocation.

Example 1

The following statement requests the current system time:

```
SELECT TIME;
```

The system responds with something like the following:

```
    Time
--------
16:20:20
```

Example 2

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 3

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);
```

USER

Purpose

Provides the session user name.

Syntax

```
——USER——
```

FF07D272

ANSI Compliance

USER is ANSI SQL:2008 compliant.

Result Type and Attributes

The data type and format for USER are as follows:

Data Type	Format
VARCHAR(30) CHARACTER SET UNICODE	X(30)

Result Value

<table>
<tr><th>IF the session logon is …</th><th>THEN …</th></tr>
<tr><td>not directory-based</td><td>the result value is the session user name.</td></tr>
<tr><td>directory-based</td><td><table>
<tr><th>IF the session …</th><th>THEN the result value …</th></tr>
<tr><td>maps to a permanent user</td><td>is the name of the permanent user.</td></tr>
<tr><td>does not map to a permanent user</td><td>is the authcid of the external user.</td></tr>
</table></td></tr>
</table>

If you are accessing the Teradata Database through a proxy connection, and you want to get the name of the proxy user, use the CURRENT_USER built-in function. For details, see "CURRENT_USER" on page 537.

Example 1

You can identify the session user name with the following statement:

```
SELECT USER;
```

The system responds with something like the following.

```
User
------------------------------
JJ43901
```

Example 2

The following example selects the job title for the session user.

```
SELECT JobTitle FROM Employee WHERE Name = USER;
```

CHAPTER 15 UDF Expressions

A UDF expression is an SQL expression that consists of one or more user-defined functions (UDFs). Teradata Database supports the following types of UDFs:

- Scalar
- Aggregate
- Table

A scalar UDF can appear almost anywhere a standard SQL scalar function can appear, and an aggregate UDF can appear almost anywhere a standard SQL aggregate function can appear. A table UDF can only appear in the FROM clause of an SQL SELECT statement.

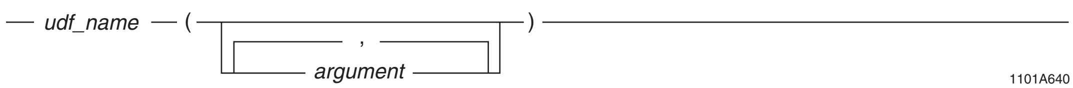
For details on UDFs, see *SQL External Routine Programming*.

Scalar UDF Expression

Purpose

An expression that includes calls to one or more scalar user-defined functions.

Syntax



where:

Syntax element …	Specifies …
udf_name	the name of the scalar UDF.
argument	a valid SQL expression. See Usage Notes for rules that apply to UDF arguments.

ANSI Compliance

UDF expressions are partially ANSI SQL:2008 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Restrictions

- Any restrictions that apply to standard SQL scalar functions also apply to scalar UDFs.
- Scalar UDF expressions cannot be used in a partitioning expression of the CREATE TABLE statement.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

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.

Usage Notes

When Teradata Database evaluates a scalar UDF expression, it invokes the scalar 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 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. For details, see *SQL External Routine Programming*.

  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 information, see "CAST in Explicit Data Type Conversions" on page 582.
- A NULL argument is compatible with a parameter of any data type.

The result type of a scalar UDF expression is based on the return type of the scalar UDF, which is specified in the RETURNS clause of the CREATE FUNCTION statement.

The default title of a scalar UDF expression appears as:

```
UDF_name(argument_list)
```

Example

Consider the following table definition and data:

```
CREATE TABLE pRecords (pname CHAR(30),
                       pkey INTEGER);

SELECT * FROM pRecords;
```

The output from the SELECT statement is:

```
pname                                  pkey
------------------------------  -----------
Tom                                       6
Bob                                       5
Jane                                      4
```

The following is the SQL definition of a scalar UDF that calculates the factorial of an integer argument:

```
CREATE FUNCTION factorial (i INTEGER)
RETURNS INTEGER
SPECIFIC factorial
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
NOT DETERMINISTIC
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'ss!factorial!factorial.c!F!fact'
```

The following query uses the scalar UDF expression to calculate the factorial of the pkey column + 1.

```
SELECT pname, factorial(pkey)+1
FROM pRecords;
```

The output from the SELECT statement is:

```
pname                           (factorial(pkey)+1)
------------------------------  -------------------
Tom                                             721
Bob                                             121
Jane                                             25
```

Related Topics

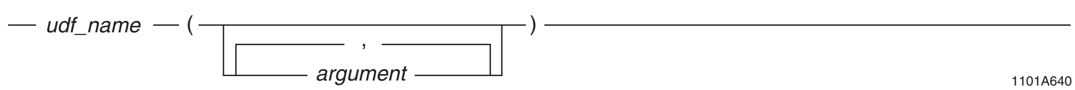
FOR more information on …	SEE …
Implementing UDFs	*SQL External Routine Programming*.
• CREATE FUNCTION • REPLACE FUNCTION	• *SQL Data Definition Language*. • *Database Administration*.
EXECUTE FUNCTION and UDTUSAGE privileges	*SQL Data Control Language*.

Aggregate UDF Expression

Purpose

An expression that includes calls to one or more aggregate user-defined functions.

Syntax



where:

Syntax element ...	Specifies ...
udf_name	the name of the aggregate UDF.
argument	a valid SQL expression. See Usage Notes for rules that apply to UDF arguments.

ANSI Compliance

UDF expressions are partially ANSI SQL:2008 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Restrictions

- Any restrictions that apply to standard SQL aggregate functions also apply to aggregate UDFs.
- Aggregate UDF expressions cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify an aggregate UDF.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

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.

Usage Notes

When Teradata Database evaluates an aggregate UDF expression, it invokes the aggregate function once for each item in an aggregation group, passing the detail values of a group through the input arguments. To accumulate summary information, the context is retained each time the aggregate function is called.

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 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. For details, see *SQL External Routine Programming*.

  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 information, see "CAST in Explicit Data Type Conversions" on page 582.
- A NULL argument is compatible with a parameter of any data type.

The result type of an aggregate expression is based on the return type of the aggregate UDF, which is specified in the RETURNS clause of the CREATE FUNCTION statement.

The default title of an aggregate UDF expression appears as:

```
UDF_name(argument_list)
```

Example

Consider the following table definition and data:

```
CREATE TABLE Product_Life
(Product_ID INTEGER,
 Product_class VARCHAR(30),
 Hours INTEGER);

SELECT * FROM Product_Life;
```

The output from the SELECT statement is:

```
Product_ID  Product_class                        Hours
----------- ------------------------------ -----------
        100  Bulbs                                 100
        100  Bulbs                                 200
        100  Bulbs                                 300
```

The following is the SQL definition of an aggregate UDF that calculates the standard deviation of the input arguments:

```
CREATE FUNCTION STD_DEV (i INTEGER)
RETURNS FLOAT
CLASS AGGREGATE (64)
SPECIFIC std_dev
LANGUAGE C
NO SQL
```

```
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'ss!stddev!stddev.c!f!STD_DEV'
```

The following query uses the aggregate UDF expression to calculate the standard deviation for the life of a light bulb.

```
SELECT Product_ID, SUM(Hours), STD_DEV(Hours)
FROM Product_Life
WHERE Product_class = 'Bulbs'
GROUP BY Product_ID;
```

The output from the SELECT statement is:

```
Product_ID   Sum(hours)          std_dev(hours)
-----------  -----------  ----------------------
        100          600    8.16496580927726E 001
```

Related Topics

FOR more information on ...	SEE ...
Implementing UDFs	*SQL External Routine Programming.*
• CREATE FUNCTION • REPLACE FUNCTION	• *SQL Data Definition Language.* • *Database Administration.*
EXECUTE FUNCTION and UDTUSAGE privileges	*SQL Data Control Language.*

Table UDF Expression

Purpose

An expression that includes a call to a table user-defined function.

Syntax

See the TABLE option of the FROM clause in *SQL Data Manipulation Language*.

ANSI Compliance

UDF expressions are partially ANSI SQL:2008 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Restrictions

A table UDF expression can only appear in the FROM clause of an SQL SELECT statement. The SELECT statement containing the table function can appear as a subquery.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

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.

Usage Notes

When Teradata Database evaluates a table UDF expression, it invokes the table function which returns a table a row at a time in a loop to the SELECT statement. The function can produce the rows of a table from the input arguments passed to it or by reading an external file or message queue.

A table function can have 128 input parameters. 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 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. For details, see *SQL External Routine Programming*.

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 information, see "CAST in Explicit Data Type Conversions" on page 582.

- A NULL argument is compatible with a parameter of any data type.

Table UDFs do not have return values. The columns in the result rows that they produce are returned as output parameters.

The output parameters of a table function are defined by the RETURNS TABLE clause of the CREATE FUNCTION statement. The number of output parameters is limited by the maximum number of columns that can be defined for a regular table.

The number and data types of the output parameters can be specified statically in the CREATE FUNCTION statement or dynamically at runtime in the SELECT statement that invokes the table function.

Example

In this example, the extract_field table UDF is used to extract the *customer ID*, *store number*, and *item ID* from the pending_data column of the raw_cust table.

The raw_cust table is defined as:

```
CREATE SET TABLE raw_cust ,NO FALLBACK ,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL,
   CHECKSUM = DEFAULT
   (
    region INTEGER,
    pending_data VARCHAR(32000) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX (region);
```

The pending_data text field is a string of numbers with the format:

store number, *entries*:*customer ID*, *item ID*, ...; repeat;

where:

- *store number* is the store that sold these items to customers.
- *entries* is the number of items that were sold.
- *customer ID*, *item ID* represent the item each customer bought. *customer ID*, *item ID* is repeated *entries* times ending with a semi-colon ';'.
- The above sequence can be repeated.

The following shows sample data from the raw_cust table:

```
  region  pending_data
--------  ----------------------------------------------------------
       2  7,2:879,3788,879,4500;08,2:500,9056,390,9004;
       1  25,3:9005,3789,9004,4907,398,9004;36,2:738,9387,738,9550;
       1  25,2:9005,7896,9004,7839;36,1:737,9387;
```

The following shows the SQL definition of the extract_field table UDF:

```
CREATE FUNCTION extract_field (Text VARCHAR(32000),
```

```
                              From_Store INTEGER)
RETURNS TABLE (Customer_ID INTEGER,
               Store_ID INTEGER,
               Item_ID INTEGER)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME extract_field;
```

The following query extracts and displays the customers and the items they bought from store 25 in region 1.

```
SELECT DISTINCT cust.Customer_ID, cust.Item_ID
FROM raw_cust,
TABLE (extract_field(raw_cust.pending_data, 25))
AS cust
WHERE raw_cust.region = 1;
```

The output from the SELECT statement is similar to:

```
Customer_ID       Item_ID
------------  -----------
        9005         3789
        9004         4907
         398         9004
        9005         7896
        9004         7839
```

Related Topics

FOR more information on ...	SEE ...
Implementing UDFs	*SQL External Routine Programming*.
• CREATE FUNCTION • REPLACE FUNCTION	• *SQL Data Definition Language*. • *Database Administration*.
EXECUTE FUNCTION and UDTUSAGE privileges	*SQL Data Control Language*.
the TABLE option in the FROM clause of an SQL SELECT statement	*SQL Data Manipulation Language*.

CHAPTER 16 UDT Expressions and Methods

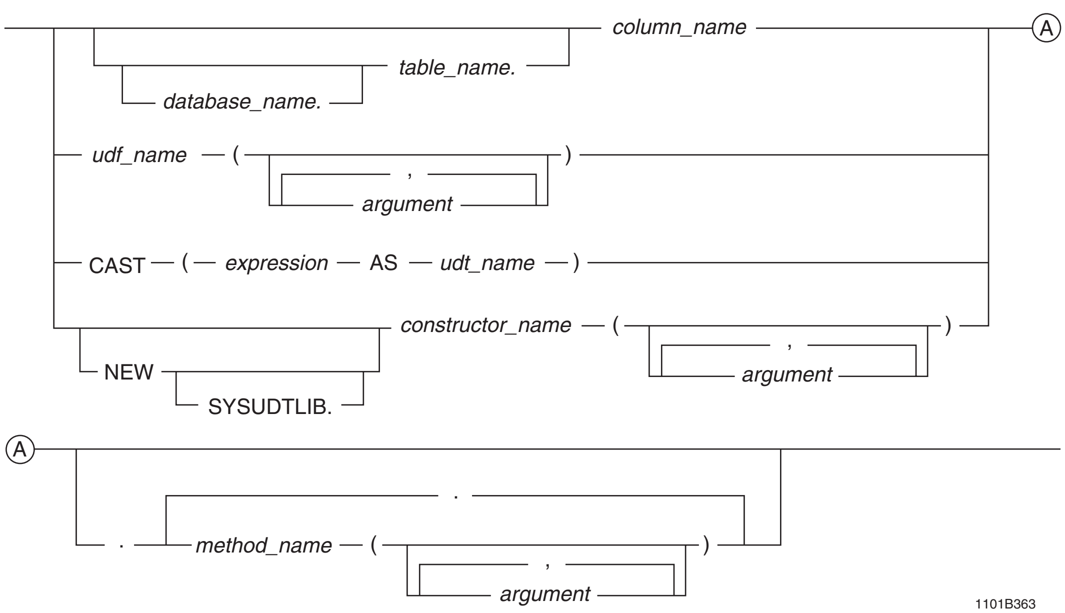
This chapter describes expressions related to user-defined types (UDTs).

UDT Expression

Purpose

Returns a distinct or structured UDT data type.

Syntax



where:

Syntax element ...	Specifies ...
database_name	an optional qualifier for the *column_name*.
table_name	an optional qualifier for the *column_name*.
column_name	the name of a distinct or structured UDT column.
udf_name	the name of a UDF that returns a distinct or structured UDT.
argument	an argument to the UDF.
CAST	a CAST expression that converts a source data type to a distinct or structured UDT. Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Syntax element ...	Specifies ...
expression	an expression that results in a data type that is compatible as the source type of a cast definition for the target UDT.
udt_name	the name of a distinct or structured UDT data type.
NEW	an expression that constructs a new instance of a structured type and initializes it using the specified constructor method. For details on NEW, see “NEW” on page 566.
SYSUDTLIB.	the database in which the constructor exists. Teradata Database only searches the SYSUDTLIB database for UDT constructors, regardless of whether the database name appears in the expression.
constructor_name	the name of a constructor method associated with a UDT. Constructor methods have the same name as the UDT with which they are associated.
argument	an argument to pass to the constructor. Parentheses must appear even though the argument list may be empty.
method_name	the name of an instance method that returns a UDT. For details on method invocation, see “Method Invocation” on page 572.
argument	an argument to pass to the method. Parentheses must appear even though the argument list may be empty.

ANSI Compliance

UDT expressions are partially ANSI SQL:2008 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Authorization

To use a UDT expression, you must have the UDTTYPE, UDTMETHOD, or UDTUSAGE on the SYSUDTLIB database or the UDTUSAGE privilege on all of the specified UDTs.

Usage Notes

You can use UDT expressions as input arguments to UDFs written in C or C++. You cannot use UDT expressions as input arguments to UDFs written in Java.

You can also use UDT expressions as IN and INOUT parameters of stored procedures and external stored procedures written in C or C++. However, you cannot use UDT expressions as IN and INOUT parameters of external stored procedures written in Java.

You can use UDT expressions with most SQL functions and operators, with the exception of ordered analytical functions, provided that a cast definition exists that casts the UDT to a predefined type that is accepted by the function or operator. For details, see other chapters in this book.

Examples

Consider the following statements that create a distinct UDT named *euro* and a structured UDT named *address*:

```
CREATE TYPE euro
AS DECIMAL(8,2)
FINAL;

CREATE TYPE address
AS (street VARCHAR(20)
   ,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE european_sales
  (region INTEGER
  ,location address
  ,sales DECIMAL(8,2));
```

Example 1: Column Name

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE italian_sales
  (location address
  ,sales DECIMAL(8,2));
```

The *location* column reference in the following statement returns an *address* UDT expression.

```
INSERT INTO italian_sales
   SELECT location, sales
   FROM european_sales
   WHERE region = 1151;
```

Example 2: CAST

The following statement creates a table that defines a *euro* column named *sales*:

```
CREATE TABLE swiss_sales
  (location address
  ,sales euro);
```

The following statement uses CAST to return a *euro* UDT expression. Using CAST requires a cast definition that converts the DECIMAL(8,2) predefined type to a *euro* type.

```
INSERT INTO swiss_sales
   SELECT location, CAST (sales AS euro)
   FROM european_sales
   WHERE region = 1038;
```

Example 3: NEW

The following INSERT statement uses NEW to return an *address* UDT expression and insert it into the *european_sales* table.

```
INSERT european_sales (1001, NEW address(), 0);
```

Example 4: Methods and Functions

The following statement uses the built-in constructor function and mutator methods to return a new instance of the *address* UDT and insert it into the *european_sales* table:

```
INSERT INTO european_sales
VALUES (101, address().street('210 Stanton').zip('76543'), 500);
```

Teradata Database executes the UDT expression in the following order:

Step	Invocation	Result
1	*address()* constructor function	Default UDT instance
2	mutator method for *street*	UDT instance with *street* attribute set to '210 Stanton'
3	mutator method for *zip*	UDT instance with *zip* attribute set to '76543'

The final result of the UDT expression is an instance of the *address* UDT with the *street* attribute set to '210 Stanton' and the *zip* attribute set to '76543'.

Related Topics

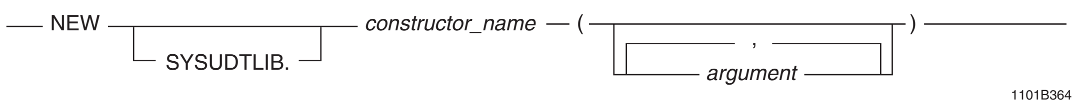
FOR more information on …	SEE …
creating a UDT	CREATE TYPE in *SQL Data Definition Language*.
creating cast definitions for a UDT	CREATE CAST in *SQL Data Definition Language*.
using UDT expressions in DML statements such as SELECT and INSERT	CREATE TYPE in *SQL Data Manipulation Language*.

NEW

Purpose

Constructs a new instance of a structured type and initializes it using the specified constructor method or function.

Syntax



where

Syntax element ...	Specifies ...
SYSUDTLIB.	the database in which the constructor exists. Teradata Database only searches the SYSUDTLIB database for UDT constructors, regardless of whether the database name appears in the NEW expression.
constructor_name	the name of the constructor, which is the same as the name of the structured type.
argument	an argument to pass to the constructor. Parentheses must appear even for constructors that take no arguments.

ANSI Compliance

NEW is partially ANSI SQL:2008 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Usage Notes

You can also construct a new instance of a structured type by calling the constructor method or function. For an example, see "Example" on page 567.

To construct a new instance of a dynamic UDT and define the run time composition of the UDT, you must use the NEW VARIANT_TYPE expression. For details, see "NEW VARIANT_TYPE" on page 569.

Default Constructor

When a structured UDT is created, Teradata Database automatically generates a constructor function with an empty argument list that you can use to construct a new instance of the structured UDT and initialize the attributes to NULL.

Determining Which Constructor is Invoked

Teradata Database uses the rules in the following table to select a UDT constructor:

<table>
<tr><th>IF the NEW expression specifies a constructor with an argument list that is …</th><th>THEN …</th></tr>
<tr><td>empty</td><td><table>
<tr><th>IF a constructor method that takes no parameters and has the same name as the UDT …</th><th>THEN Teradata Database selects …</th></tr>
<tr><td>exists in the SYSUDTLIB database</td><td>that constructor method.</td></tr>
<tr><td>does not exist in the SYSUDTLIB database</td><td>the constructor function that is automatically generated when the structured UDT is created.</td></tr>
</table></td></tr>
<tr><td>not empty</td><td>Teradata Database selects the constructor method in SYSUDTLIB with a parameter list that matches the arguments passed to the constructor in the NEW expression.</td></tr>
</table>

Example

Consider the following statement that creates a structured UDT named *address*:

```
CREATE TYPE address
AS (street VARCHAR(20)
   ,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE european_sales
  (region INTEGER
  ,location address
  ,sales DECIMAL(8,2));
```

The following statement uses NEW to insert an *address* value into the *european_sales* table:

```
INSERT european_sales (1001, NEW address(), 0);
```

Teradata Database selects the default constructor function that was automatically generated for the *address* UDT because the argument list is empty and the *address* UDT was created with no constructor method. The default *address* constructor function initializes the *street* and *zip* attributes to NULL.

The following statement is equivalent to the preceding INSERT statement but calls the constructor function instead of using NEW:

```
INSERT european_sales (1001, address(), 0);
```

Related Topics

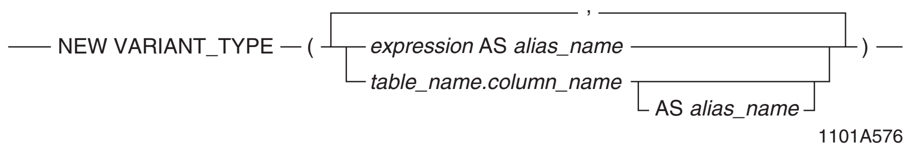
FOR more information on …	SEE …
creating constructor methods	CREATE METHOD in *SQL Data Definition Language*.
the constructor function that Teradata Database automatically generates when the structured type is created	CREATE TYPE (Structured Form) in *SQL Data Definition Language*.
constructing a new instance of a dynamic UDT and defining the run time composition of the UDT	"NEW VARIANT_TYPE" on page 569

NEW VARIANT_TYPE

Purpose

Constructs a new instance of a dynamic or VARIANT_TYPE UDT and defines the run time composition of the UDT.

Syntax



where

Syntax element ...	Specifies ...
expression	any valid SQL expression; however, the following restrictions apply: • *expression* cannot contain a dynamic UDT expression. Nesting of dynamic UDT expressions is not allowed. • the first expression (that is, the first attribute of the dynamic UDT) cannot be a LOB, UDT, or LOB-UDT expression.
alias_name	a name representing the expression or column reference which corresponds to an attribute of the dynamic UDT. When provided, *alias_name* is used as the name of the attribute. You must provide an alias name for any expression that is not a column reference. You cannot assign the same alias name to more than one attribute of the dynamic UDT. Also, you cannot specify an alias name that is the same as a column name if that column name is already used as an attribute name in the dynamic UDT.
table_name	the name of the table in which the column being referenced is stored.
column_name	the name of the column being referenced. If you do not provide an alias name, the column name is used as the name of the corresponding attribute in the dynamic UDT. The same column name cannot be used as an attribute name for more than one attribute of the dynamic UDT. If a column has the same name as an alias name, the column name cannot be used as an attribute name.

ANSI Compliance

NEW VARIANT_TYPE is a Teradata extension to the ANSI SQL standard.

Usage Notes

You can use the NEW VARIANT_TYPE expression to define the run time composition or internal attributes of a dynamic UDT. Each expression you pass into the NEW VARIANT_TYPE constructor corresponds to one attribute of the dynamic UDT. You can assign an alias name to represent each NEW VARIANT_TYPE expression parameter. The name of the attribute will be the alias name provided or the column name associated with the column reference if no alias is provided. This is summarized in the following table:

IF...	THEN the attribute name is...
alias_name is provided	*alias_name*
table_name.column_name is provided, but *alias_name* is not provided	*column_name*
an *expression* is provided that is not a column reference and *alias_name* is not provided	an error is returned.

Note that you must provide an alias name for all expressions that are not column references. In addition, the attribute names must be unique. Therefore, you must provide unique alias names and/or column references.

The data type of the attribute will be the result data type of the expression. The resultant value of the expression will become the value of the corresponding attribute.

Restrictions

- You can use the NEW VARIANT_TYPE expression only to construct dynamic UDTs for use as input parameters to UDFs. To construct a new instance of other structured UDTs, use the NEW expression. For details, see "NEW" on page 566.
- UDFs support a maximum of 128 parameters. Therefore, you cannot use NEW VARIANT_TYPE to construct a dynamic UDT with more than 128 attributes.
- The sum of the maximum sizes for all the attributes of the dynamic UDT must not exceed the maximum permissible column size as configured for the Teradata Database. Exceeding the maximum column size results in the following SQL error: "ERR_TEQRWOVRFLW _T("Row size or Sort Key size overflow.")".

Example 1

The following NEW VARIANT_TYPE expression creates a dynamic UDT with a single attribute named *weight*:

```
NEW VARIANT_TYPE (Table1.a AS weight)
```

In the next example, the NEW VARIANT_TYPE expression creates a dynamic UDT with a single attribute named *height*. In this example, no alias name is specified; therefore, the column name is used as the attribute name.

```
NEW VARIANT_TYPE (Table1.height)
```

In the next example, the first attribute is named *height* based on the column name. However, the second attribute is also named *height* based on the specified alias name. This is not allowed since attribute names must be unique; therefore, the Teradata Database returns the error, "ERRTEQDUPLATTRNAME - "Duplicate attribute names in the attribute list. %VSTR", being returned to the user."

```
NEW VARIANT_TYPE (Table1.height, Table1.a AS height)
```

Example 2

This example shows a user-defined aggregate function with an input parameter named *parameter_1* declared as VARIANT_TYPE data type. The SELECT statement calls the new function using the NEW VARIANT_TYPE expression to create a dynamic UDT with two attributes named *a* and *b*.

```
CREATE TYPE INTEGERUDT AS INTEGER FINAL;

CREATE FUNCTION udf_agch002002dynudt (parameter_1  VARIANT_TYPE)
RETURNS INTEGERUDT CLASS AGGREGATE (4) LANGUAGE C  NO SQL
EXTERNAL NAME   'CS!udf_agch002002dynudt!udf_agch002002dynudt.c'
PARAMETER STYLE SQL;

SELECT udf_agch002002dynudt(NEW VARIANT_TYPE (Tbl1.a AS a,
                                              (Tbl1.b + Tbl1.c) AS b))
FROM Tbl1;
```

Related Topics

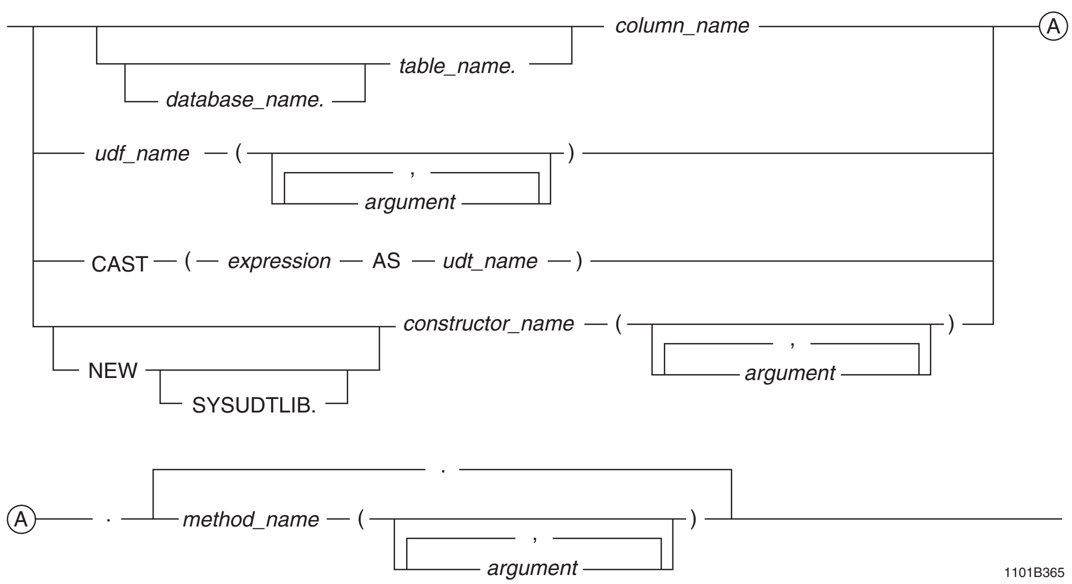
FOR more information on ...	SEE ...
dynamic UDTs	"VARIANT_TYPE UDT" in *SQL Data Types and Literals*.
constructing a new instance a structured UDT that is *not* a dynamic UDT	"NEW" on page 566.
writing UDFs which use input parameters of VARIANT_TYPE data type	*SQL External Routine Programming*

Method Invocation

Purpose

Invokes a method associated with a UDT.

Syntax



where:

Syntax element ...	Specifies ...
database_name	an optional qualifier for the *column_name*.
table_name	an optional qualifier for the *column_name*.
column_name	the name of a distinct or structured UDT column.
udf_name	the name of a UDF that returns a distinct or structured UDT.
argument	an argument to the UDF.
CAST	a CAST expression that converts a source data type to a distinct or structured UDT. Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Syntax element ...	Specifies ...
expression	an expression that results in a data type that is compatible as the source type of a cast definition for the target UDT.
udt_name	the name of a distinct or structured UDT.
NEW	an expression that constructs a new instance of a structured type and initializes it using the specified constructor method. For details on NEW, see "NEW" on page 566.
SYSUDTLIB.	the database in which the constructor exists. Teradata Database only searches the SYSUDTLIB database for UDT constructors, regardless of whether the database name appears in the expression.
constructor_name	the name of a constructor method associated with a UDT. Constructor methods have the same name as the UDT with which they are associated.
argument	an argument to pass to the constructor. Parentheses must appear even though the argument list may be empty.
method_name	the name of an observer, mutator, or user-defined method (UDM). You must precede each method name with a period.
argument	an argument to pass to the method. Parentheses must appear even though the argument list may be empty.

ANSI Compliance

Invocation of UDT methods is partially ANSI SQL:2008 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Additionally, when a statement specifies an ambiguous expression that can be interpreted as a UDF invocation or a method invocation, Teradata Database gives UDF invocation higher precedence over method invocation. ANSI SQL:2008 gives method invocation higher precedence over UDF invocation.

Observer and Mutator Methods

Teradata Database automatically generates *observer* and *mutator* methods for each attribute of a structured UDT. Observer and mutator methods have the same name as the attribute for which they are generated.

Method	Description	Invocation Example
Observer	Takes no arguments and returns the current value of the attribute.	"Example" on page 574

Method	Description	Invocation Example
Mutator	Takes one argument and returns a new UDT instance with the specified attribute set to the value of the argument.	"Example 4: Methods and Functions" on page 565

Usage Notes

When you invoke a UDM on a UDT, Teradata Database searches the SYSUDTLIB database for a UDM that has the UDT as its first parameter followed by the same number of parameters as the method invocation.

If several UDMs have the same name, Teradata Database must determine which UDM to invoke. For details on the steps that Teradata Database uses, see *SQL External Routine Programming*.

Restrictions

To use any of the following functions as the first argument of a method invocation, you must enclose the function in parentheses:

- DATE
- TIME
- VARGRAPHIC

For example, consider a structured UDT called *datetime_record* that has a DATE type attribute called *start_date*. The following statement invokes the *start_date* mutator method, passing in the result of the DATE function:

```
SELECT datetime_record_column.start_date((DATE)) FROM table1;
```

Example

Consider the following statement that creates a structured UDT named *address*:

```
CREATE TYPE address
AS (street VARCHAR(20)
   ,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE european_sales
  (region INTEGER
  ,location address
  ,sales DECIMAL(8,2));
```

The following statement invokes the *zip* observer method to retrieve the value of each *zip* attribute in the *location* column:

```
SELECT location.zip() FROM european_sales;
```

Related Topics

FOR more information on …	SEE …
creating methods	CREATE METHOD in *SQL Data Definition Language*.
creating UDTs	CREATE TYPE in *SQL Data Definition Language*.
UDM programming	*SQL External Routine Programming*.

CHAPTER 17 Data Type Conversions

This chapter describes the SQL CAST function and the rules for converting data from one type to another, both explicitly and implicitly.

A data type conversion modifies the data definition (data type, data attributes, or both) of an expression and can be either implicit or explicit. Explicit conversions can be made using the CAST function or Teradata conversion syntax.

For details on data types and data attributes, see *SQL Data Types and Literals*.

Forms of Data Type Conversions

Teradata Database supports the following forms of data conversion:

- Implicit

  See "Implicit Type Conversions" on page 577.
- Explicit using the CAST function

  See "CAST in Explicit Data Type Conversions" on page 582.
- Explicit using Teradata conversion syntax

  See "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 585.

Implicit Type Conversions

Teradata Database permits the assignment and comparison of some types without requiring the types to be explicitly converted. Teradata Database also performs implicit type conversions on some argument types passed to macros, stored procedures, and SQL functions such as SQRT.

ANSI Compliance

Implicit conversions are Teradata extensions to the ANSI standard.

Example 1: Implicit Type Conversion During Assignment

Consider the following tables:

```
CREATE TABLE T1
   (Fname VARCHAR(25)
   ,Fid   INTEGER
   ,Yrs   CHARACTER(2));
```

```
CREATE TABLE T2
   (Wname VARCHAR(25)
   ,Wid   INTEGER
   ,Age   SMALLINT);
```

In the following statement, Teradata Database implicitly converts the character string in T1.Yrs to a numeric value:

```
UPDATE T2 SET Age = T1.Yrs + 5;
```

This is not evident in the syntax of the source statement, but becomes evident when the dictionary information for tables T1 and T2 is accessed.

Example 2: Implicit Type Conversion During Comparison

Consider the table T1 in "Example 1: Implicit Type Conversion During Assignment."

In the following statement, Teradata Database implicitly converts both operands of the comparison operation to FLOAT values before performing the comparison:

```
SELECT Fname, Fid
FROM T1
WHERE T1.Yrs < 55;
```

For details on implicit type conversion of operands for comparison operations, see "Implicit Type Conversion of Comparison Operands" on page 108.

Example 3: Implicit Type Conversion in Parameter Passing Operations

Consider the SQRT system function that computes the square root of an argument.

In the following statement, Teradata Database implicitly converts the character argument to a FLOAT type:

```
SELECT SQRT('13147688');
```

Supported Data Types

Teradata Database performs implicit conversion on the following types:

FROM ...	TO ...	For further details, see ...
Byte	Byte Byte types include BYTE, VARBYTE, and BLOB.	"Byte Conversion" on page 588.
	UDT[a]	
Numeric	Numeric	"Numeric-to-Numeric Conversion" on page 664.
	DATE	"Numeric-to-DATE Conversion" on page 659.
	Character	"Numeric-to-Character Conversion" on page 654.
	UDT[a]	"Numeric-to-UDT Conversion" on page 668.

FROM ...	TO ...	For further details, see ...
DATE	Numeric	"DATE-to-Numeric Conversion" on page 634.
	DATE	"DATE-to-DATE Conversion" on page 632.
	Character	"DATE-to-Character Conversion" on page 628.
	UDT[a]	"DATE-to-UDT Conversion" on page 642.
Character	Numeric	"Character-to-Numeric Conversion" on page 608.
	DATE	"Character-to-DATE Conversion" on page 597.
	Character Character types include CHAR, VARCHAR, and CLOB.	"Character-to-Character Conversion" on page 592.
	Period	"Character-to-Period Conversion" on page 605.
	TIME	"Character-to-TIME Conversion" on page 614.
	TIMESTAMP	"Character-to-TIMESTAMP Conversion" on page 620.
	UDT[a]	"Character-to-UDT Conversion" on page 625.
TIME	UDT[a]	"TIME-to-UDT Conversion" on page 699.
TIMESTAMP	UDT[a]	"TIMESTAMP-to-UDT Conversion" on page 716.
Interval	UDT[a]	"INTERVAL-to-UDT Conversion" on page 652.
UDT	Predefined data types that are the target of implicit casts defined for the UDT[b]	• "UDT-to-Character Conversion" on page 721. • "UDT-to-DATE Conversion" on page 725. • "UDT-to-INTERVAL Conversion" on page 728. • "UDT-to-Numeric Conversion" on page 731. • "UDT-to-TIME Conversion" on page 734. • "UDT-to-TIMESTAMP Conversion" on page 737.
	Other UDTs that are the target of implicit casts defined for the UDT[b]	"UDT-to-UDT Conversion" on page 740.

a. The UDT must have an implicit cast that casts the predefined type to a UDT. 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 *SQL Data Definition Language*.

b. 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 *SQL Data Definition Language*.

For details on data types, see *SQL Data Types and Literals*.

Implicit Conversion of DateTime types

Teradata Database performs implicit conversion on DateTime data types in the following cases:

- When passing data using dynamic parameter markers, or the question mark (?) placeholder.
- With INSERT, INSERT...SELECT, and UPDATE statements.
- With MERGE INTO statements.
- When handling default values for the CREATE/ALTER TABLE statements. For details, see "DEFAULT Phrase" in *SQL Data Types and Literals*.
- During stored procedure execution, including the execution of the following statements: DECLARE, SELECT...INTO, and SET. See *SQL Stored Procedures and Embedded SQL*.

Implicit conversion is dependent on client-side support. For information about the client products which support implicit conversion of DateTime types, see the Teradata Tools and Utilities user documentation.

The following conversions are supported:

FROM...	TO...	For further details, see...
DATE	TIMESTAMP	"Implicit DATE-to-TIMESTAMP Conversion" on page 640.
TIME	TIMESTAMP	"Implicit TIME-to-TIMESTAMP Conversion" on page 697.
TIMESTAMP	DATE	"Implicit TIMESTAMP-to-DATE Conversion" on page 706.
TIMESTAMP	TIME	"Implicit TIMESTAMP-to-TIME Conversion" on page 711.
INTERVAL	INTERVAL	"Implicit INTERVAL-to-INTERVAL Conversion" on page 648.

Teradata Database performs implicit conversion on DateTime data types during assignment in the following cases:

FROM...	TO...	For further details, see...
DATE	TIMESTAMP	"Implicit DATE-to-TIMESTAMP Conversion" on page 640.
TIME	TIMESTAMP	"Implicit TIME-to-TIMESTAMP Conversion" on page 697.
TIMESTAMP	DATE	"Implicit TIMESTAMP-to-DATE Conversion" on page 706.
TIMESTAMP	TIME	"Implicit TIMESTAMP-to-TIME Conversion" on page 711.
Interval[a]	Exact Numeric	"Implicit INTERVAL-to-Numeric Conversion" on page 651.
Exact Numeric	Interval[a]	"Implicit Numeric-to-INTERVAL Conversion" on page 663.

a. The INTERVAL type must have only one field, e.g. INTERVAL YEAR.

For more information, see "ANSI DateTime and Interval Data Type Assignment Rules" on page 152.

Teradata Database performs implicit conversion on DateTime data types in single table predicates and join predicates in the following cases:

FROM...	TO...	For further details, see...
TIMESTAMP	DATE	"Implicit TIMESTAMP-to-DATE Conversion" on page 706.
Interval[a]	Exact Numeric	"Implicit INTERVAL-to-Numeric Conversion" on page 651.
Exact Numeric	Interval[a]	"Implicit Numeric-to-INTERVAL Conversion" on page 663.

a. The INTERVAL type must have only one field, e.g. INTERVAL YEAR.

For more information, see "Implicit Type Conversion of Comparison Operands" on page 108.

The following are not supported:

- Implicit conversion from TIME to TIMESTAMP and from TIMESTAMP to TIME are not supported in comparisons.
- Implicit conversion of DateTime types in set operations.

For details on data types, see *SQL Data Types and Literals*.

Implicit Conversion Rules

Teradata SQL performs implicit type conversions on expressions before any operation is performed.

The implementation of implicit type conversion follows the same rules as the implementation of explicit type conversion using Teradata conversion syntax.

For details on implicit type conversion of operands for comparison operations, see "Implicit Type Conversion of Comparison Operands" on page 108.

Restrictions

Teradata Database does not perform implicit conversion on input arguments to UDFs, UDMs, or external stored procedures (external routines). Arguments do not necessarily have to be exact matches to the parameter types, but they must be compatible. For example, you can pass a SMALLINT argument to an external routine that expects an INTEGER argument because SMALLINT and INTEGER are compatible. To pass a DATE type argument to an external routine that expects an INTEGER argument, you must explicitly cast the DATE type to an INTEGER type. For details, see *SQL External Routine Programming*.

Some SQL functions and operators require arguments that are exact matches to the parameter types. For details, refer to the documentation for the specific function or operator.

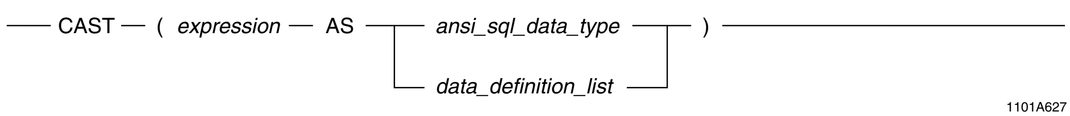
CAST in Explicit Data Type Conversions

Purpose

Converts an expression of a given data type to a different data type or the same data type with different attributes.

Teradata SQL supports two different syntaxes for CAST functionality, only one of which is ANSI SQL:2008 compliant.

Syntax



where:

Syntax element ...	Specifies ...
expression	an expression with known data type to be cast as a different data type.
ansi_sql_data_type	the new data type for *expression*.
data_definition_list	the new data type or data attributes or both for *expression*.

ANSI Compliance

The form of CAST syntax that specifies *ansi_sql_data_type* is ANSI SQL:2008 compliant.

The form of CAST syntax that specifies *data_definition_list* is a Teradata extension to the ANSI SQL:2008 standard. Note that when *data_definition_list* consists solely of an ANSI data type declaration, then this form of the syntax is also ANSI-compliant.

Usage Notes

The ANSI SQL:2008 compliant form can be used to convert data types in either ANSI-compliant SQL statements or Teradata SQL statements.

The Teradata extended syntax is more general. It allows a type declaration *or* data attributes *or* both. For more information on data types and attributes, see *SQL Data Types and Literals*.

Avoid using the extended form of CAST for any application intended to be ANSI-compliant and portable.

CAST functions identically in both ANSI and Teradata modes.

CAST does not convert the following data type pairs:

- Numeric to character, if the server character set is GRAPHIC.
- Character expressions having different server character sets.

  To make such a conversion, use the TRANSLATE function (see "TRANSLATE" on page 407).
- Byte (BYTE, VARBYTE, and BLOB) to any data type other than UDT or byte, and data types other than byte or UDT to byte.
- CLOB to any data type other than UDT or character, and data types other than character or UDT to CLOB.

For information on casting to and from geospatial types, see *SQL Geospatial Types*.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Character Truncation Rules

The following rules apply to character strings:

IF the string is cast in this mode …	THEN it is truncated of …
ANSI	trailing pad character spaces to achieve the desired length. Truncation of other characters, or part of a multibyte character, returns an error.
Teradata	trailing characters to achieve the desired length. Truncation on Kanji1 character data types containing multibyte characters might result in truncating one byte of the multibyte character.

Server Character Set Rules

When *data_definition_list* specifies a data type of CHARACTER (CHAR) or CHARACTER VARYING (VARCHAR) and does not specify a CHARACTER SET clause to indicate which server character set to use, then the resulting server character set is as follows:

IF the data type of *expression* is …	THEN the server character set of the resulting characters is …
non-character	the user default server character set.
character	the server character set of *expression*.

Numeric Overflow, Field Mode, and CAST

Numeric overflows are handled differently depending on whether you are running ANSI or Teradata mode, and whether you are running in Field Mode or not.

Field Mode is not ANSI SQL:2008 compatible. In Field Mode, conversion to a numeric or decimal data type that results in a numeric overflow is returned as asterisks ('***') rather than an error message.

Record and Indicator Modes do not behave in this manner and return an error message.

Related Topics

For further rules that apply to the conversion between specific data types, for example, numeric-to numeric or character-to-numeric, see the appropriate succeeding topic in this chapter.

Examples

The following examples illustrate how to perform data type conversions using CAST.

Example 1

Using ANSI CAST syntax:

```
SELECT ID_Col, Name_Col
FROM T1
WHERE Int_Col = CAST(SUBSTRING(Char_Col FROM 3 FOR 3) AS INTEGER);
```

Example 2

Using ANSI CAST syntax:

```
SELECT CAST(SUBSTRING(Char_Col FROM 1 FOR 2) AS INTEGER),
   CAST(SUBSTRING (Char_Col FROM 3 FOR 3) AS INTEGER)
FROM T1;
```

Example 3

Using Teradata extensions to the ANSI CAST syntax:

```
CREATE TABLE t2 (f1 TIME(0) FORMAT 'HHhMIm');

INSERT t2 (CAST('15h33m' AS TIME(0) FORMAT 'HHhMIm'));

SELECT f1 FROM t2;
```

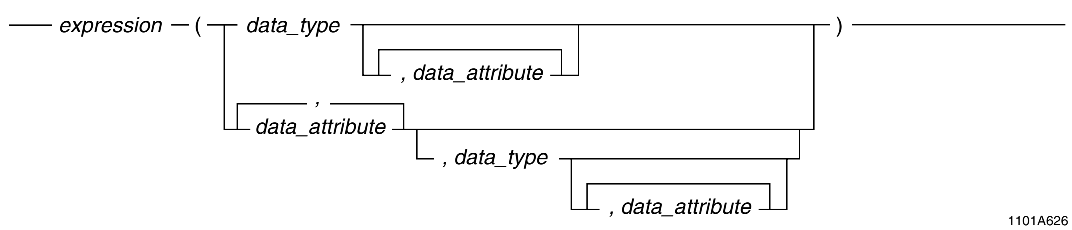
The result from the SELECT statement is:

```
    f1
------
15h33m
```

Teradata Conversion Syntax in Explicit Data Type Conversions

Teradata conversion syntax is defined as follows:

Syntax



where:

Syntax element ...	Specifies ...
expression	the data expression to be converted to the new definition specified by *data_type* and *data_attributes*.
data_type	a data type declaration such as INTEGER or DATE.
data_attribute	a data attribute such as FORMAT or TITLE.

ANSI Compliance

This syntax is a Teradata extension to the ANSI SQL:2008 standard.

Using CAST Instead of Teradata Conversion Syntax

Using Teradata conversion syntax is strongly discouraged. It is an extension to the ANSI SQL:2008 standard and is retained only for backward compatibility with existing applications. Instead, use CAST to explicitly convert data types.

Usage Notes

When the conversion specifies *data_type*, then the data is converted at run time. At that time, a data conversion or range check error may occur.

For any kind of data type conversion using Teradata conversion syntax, where the item that includes a data type declaration is an operand of a complex expression, you must either enclose the appropriate entities in parentheses or use the CAST syntax.

You should always use the CAST function to perform conversions in new applications to ensure ANSI compatibility.

Related Topics

For further rules that apply to the conversion between specific data types, for example, numeric-to numeric or character-to-numeric, see the appropriate succeeding topic in this chapter.

Example 1

To evaluate an expression of the following form correctly:

```
column_name (INTEGER) + variable
```

You could enter the expression as follows:

```
(column_name (INTEGER)) + variable
```

or, preferably, as:

```
CAST (column_name AS INTEGER) + variable
```

For more information on using CAST, see “CAST in Explicit Data Type Conversions” on page 582.

Example 2

Here is an example that uses the Teradata conversion syntax, and specifies the FORMAT data attribute to convert the format of a DATE data type.

```
CREATE TABLE date1 (d1 DATE FORMAT 'E4,BM4BDD,BY4');
CREATE TABLE char1 (c1 CHAR(10));

INSERT date1 ('Saturday, March 16, 2002');

INSERT INTO char1 (c1)
SELECT ((d1 (FORMAT 'YYYY/MM/DD')))
FROM date1;

SELECT * FROM char1;
```

The result from the SELECT statement is:

```
        c1
----------
2002/03/16
```

If the second INSERT statement did not convert the DATE format to 'YYYY/MM/DD', the result from the SELECT statement is:

```
        c1
----------
Saturday,
```

Data Conversions in Field Mode

Field Mode: User Response Data

In Field Mode, a report format used in BTEQ, all data is returned in character form. The alignment and spacing of columns is controlled by data formats and title information. Each row returned is essentially a character string ready for display.

In Field Mode, it is unnecessary to explicitly convert numeric data to character format.

Conversions to Numeric Types

When in Field Mode, a numeric overflow returned for character to numeric data type conversion is not treated as an error. If the result exceeds the number of digits normally reserved for the numeric data type, the result appears as a set of asterisks in the report.

For example, the character to SMALLINT conversion in the following statement results in numeric overflow because the number of digits normally reserved for a SMALLINT is five:

```
SELECT '100000' (SMALLINT);
```

The result is:

```
'100000'
--------
  ******
```

Additionally, when in Field Mode, asterisks appear in the report for conversions to numeric types involving results that do not fit the specified output format.

For example, the DATE to INTEGER conversion in the following statement results in a value that does not fit the format specified by the FORMAT phrase:

```
SELECT CAST (CURRENT_DATE as integer format '9999');
```

The result is:

```
Date
----
****
```

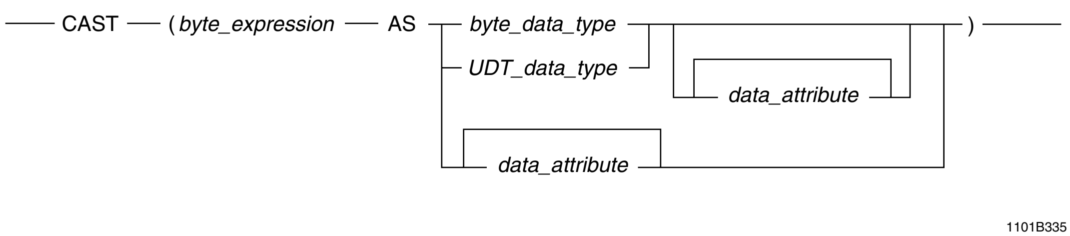
The same query executed in Record or Indicator Variable Mode reports an error.

Byte Conversion

Purpose

Converts a byte expression to a different data definition.

CAST Syntax



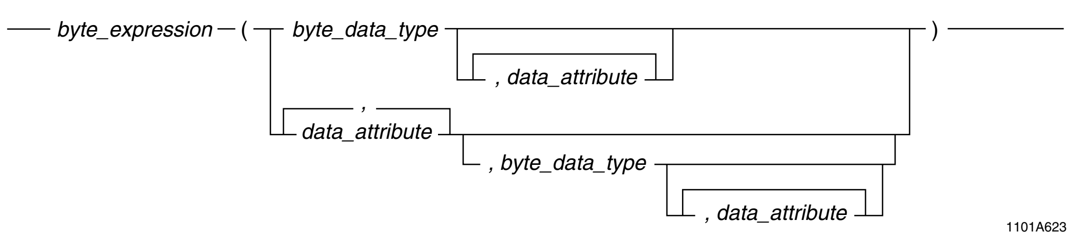
where:

Syntax element ...	Specifies ...
byte_expression	an expression in byte format to be cast to a different data definition.
byte_data_type	the new byte type to which *byte_expression* is to be converted.
UDT_data_type	a UDT that has a cast definition that casts the byte type to the UDT. To define a cast for a UDT, use the CREATE CAST statement. For details on CREATE CAST, see *SQL Data Definition Language*.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant, provided the syntax does not specify data attributes.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
byte_expression	an expression in byte format to be cast to a different byte data definition.
byte_data_type	an optional byte type to which *byte_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Conversions Where Source and Target Types Differ in Length

If the length specified by *byte_data_type* is less than the length of *byte_expression*, bytes beyond the specified length are truncated. No error is reported.

If *byte_data_type* is fixed-length and the length is greater than that of *byte_expression*, bytes of value binary zero are appended as required.

Supported Source and Target Data Types

Teradata Database supports byte data type conversions according to the following table.

Source Data Type	Target Data Type	Allowable Conversions
BYTE VARBYTE BLOB	• BYTE • VARBYTE • BLOB	• Implicit • Explicit using CAST and Teradata conversion syntax
BYTE VARBYTE BLOB	UDT[a]	• Implicit • Explicit using CAST
UDT[a]	• BYTE • VARBYTE • BLOB	• Implicit • Explicit using CAST and Teradata conversion syntax

a. Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Rules for Implicit Byte-to-UDT Conversions

Teradata Database performs implicit Byte-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit Byte-to-UDT data type conversion requires a cast definition (see "Usage Notes") that specifies the following:

- the AS ASSIGNMENT clause
- a BYTE, VARBYTE, or BLOB source data type

  The source data type of the cast definition does not have to be an exact match to the source of the implicit type conversion.

If multiple implicit cast definitions exist for converting different byte types to the UDT, Teradata Database uses the implicit cast definition for the byte type with the highest precedence. The following list shows the precedence of byte types in order from lowest to highest precedence:

- BYTE
- VARBYTE
- BLOB

Using HASHBUCKET to Convert a BYTE Type to an INTEGER Type

You can use the HASHBUCKET function to convert a BYTE(1) or BYTE(2) type to an INTEGER type. For details, see "Using HASHBUCKET to Convert a BYTE Type to an INTEGER Type" on page 523.

Example 1: Explicit Conversion of BLOB to VARBYTE

Consider the following table definition:

```
CREATE TABLE large_images
  (id INTEGER
  ,image BLOB);
```

The following statement casts the BLOB column to a VARBYTE type, and uses the result as an argument to the POSITION function:

```
SELECT POSITION('FFF1'xb IN (CAST(image AS VARBYTE(64000))))
FROM large_images
WHERE id = 5;
```

Example 2: Implicit Conversion of VARBYTE to BLOB

Consider the following table definitions:

```
CREATE TABLE small_images
  (id INTEGER
  ,image1 VARBYTE(30000)
  ,image2 VARBYTE(30000));

CREATE TABLE large_images
  (id INTEGER
  ,image BLOB);
```

Teradata Database performs a VARBYTE to BLOB implicit conversion for the following INSERT statement:

```
INSERT large_images
SELECT id, image1 || image2
FROM small_images;
```

Related Topics

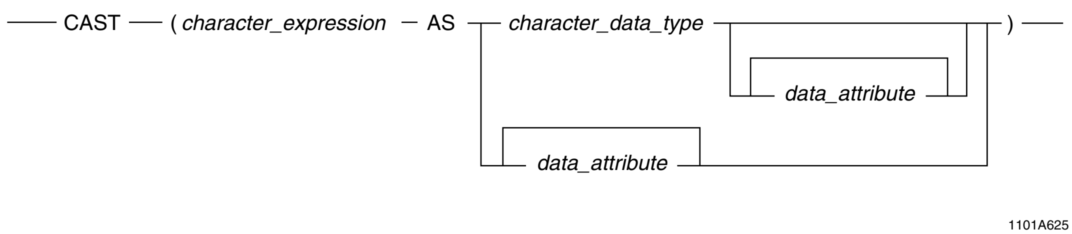
For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-Character Conversion

Purpose

Shortens or expands output character strings.

CAST Syntax



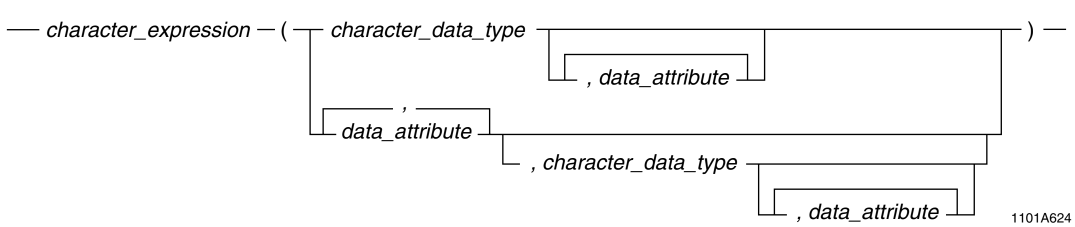
where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a different character data definition.
character_data_type	the new data type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE • CHARACTER SET

ANSI Compliance

CAST is ANSI SQL:2008 compliant, provided the syntax does not specify any data attributes.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a different character data definition.
character_data_type	an optional character type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE • CHARACTER SET If the syntax specifies *character_data_type*, CHARACTER SET can only appear after *character_data_type*.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Implicit Character-to-Character Conversion

CLOB types can only be converted to or from CHAR or VARCHAR types. For example, implicit conversion is performed on CLOB data that is inserted into a CHAR or VARCHAR column.

Comparisons of strings (both fixed- and variable-length) require operands of equal length. The following table shows that the shorter string is converted by being padded on the right.

THIS expression ...	IS converted to ...	AND the result is ...
'x'='x '	'xΔ'='x '	TRUE
'x'='xx'	'xΔ'='xx'	FALSE

where Δ is a pad character.

If a character is not in the repertoire of the target character set, an error is reported.

For rules on the effect of server character sets on character conversion, see "Implicit Character-to-Character Translation" on page 595.

CAST Syntax Usage Notes

The server character set of *character_expression* must have the same server character set as the target data type.

If CAST is used to convert data to a character string and non-pad characters would be truncated, an error is reported.

Teradata Conversion Syntax Usage Notes

The server character set of *character_expression* can be changed to a different server character set specified as *data_attribute*, where *data_attribute* is the CHARACTER SET phrase.

This is not the recommended way to perform this translation. Instead, use the TRANSLATE function. For information, see "TRANSLATE" on page 407.

General Usage Notes

If the source string (CHAR, VARCHAR, or CLOB) is longer than the target data type (CHAR, VARCHAR, or CLOB), excess characters are truncated.

IF the session doing an INSERT or UPDATE is in this mode …	AND non-pad characters would be truncated to store character values in a table, THEN …
ANSI	an error is reported.
Teradata	no error is reported.

Pad characters are trimmed or appended, according to the following rules:

IF the source string data type is …	AND it is …	AND the target data type is …	THEN …
CHAR	longer than the target	CLOB or VARCHAR	any trailing pad characters are trimmed.
CHAR, VARCHAR, or CLOB	shorter than the target	CHAR	trailing pad characters are appended to the target.
CHAR	all pad characters	CLOB or VARCHAR	the field is truncated to zero length.

Examples

Following are examples of character to character conversions:

Character String	String Length	Character Description	Conversion Result	Converted Length
'HELLO'	5	CHAR(3)	'HEL', if session is in Teradata mode	3
			Error, if session is in ANSI mode	
'HELLO'	5	CHAR(7)	'HELLO '	7
'HELLO'	5	VARCHAR(7)	'HELLO'	5
'HELLO '	7	VARCHAR(6)	'HELLO '	6

Character String	String Length	Character Description	Conversion Result	Converted Length
'HELLO '	7	VARCHAR(3)	'HEL', if session is in Teradata mode	3
			Error, if session is in ANSI mode	

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Implicit Character-to-Character Translation

Implicit string translation occurs when two character strings are incompatible within a given operation. For example,

```
SELECT *
FROM string_table
WHERE clatin < csjis;
```

where clatin represents a character column defined as CHARACTER SET LATIN and csjis represents a character column defined as CHARACTER SET KANJISJIS.

If an implicit translation of character string '*string*' to a UNICODE character string is required, it is equivalent to executing the TRANSLATE*(string* USING *source_repertoire_name*_TO_Unicode*)* function, where *source-repertoire-name* is the server character set of *string*.

More specifically, if as in the above example, *string* is of KANJISJIS type, then the translation is equivalent to executing the TRANSLATE*(string* USING KanjiSJIS_TO_Unicode) function.

ANSI Compliance

Implicit translations are Teradata extensions to the ANSI standard.

Character Constants

The following rules apply to implicit character-to-character translation involving character constants.

IF one operand is a …	AND the other operand is a …	THEN …
constant	constant	both operands are translated to UNICODE.
	non-constant	the constant is translated to the type of the non-constant. If that fails, both are translated to UNICODE.
	constant expression	the constant is translated to the type of the constant expression. If that fails, both are translated to UNICODE.

IF one operand is a ...	AND the other operand is a ...	THEN ...
constant expression	constant expression	both operands are translated to UNICODE.
	non-constant	the constant expression is translated to the type of the non-constant. If that fails, both are translated to UNICODE.
non-constant	non-constant	both operands are translated to UNICODE.

KANJISJIS Server Character Set

Implicit character-to-character translation always converts a character string argument that has the KANJISJIS server character set to UNICODE.

SQL Rules for Implicit Translation for Expression and Function Arguments

The following are the rules for implicit translation between types of expressions and function arguments.

For string functions that produce a character result, the results are summarized by this table.

FOR this function ...	The result is ...
TRIM	converted back to the type of the main string argument (last argument).
|| (concatenation)	not translated and remains with the character data type of the arguments after any implicit translation.

Note that the other string functions either do not involve conversion or the type of the result is based on the function and not the server character set of the argument.

For example, in the following TRIM function, <unicode-constant> is first translated to Latin, and then the trim operation is performed.

```
...
TRIM(<unicode-constant> FROM <latin-value>)
```

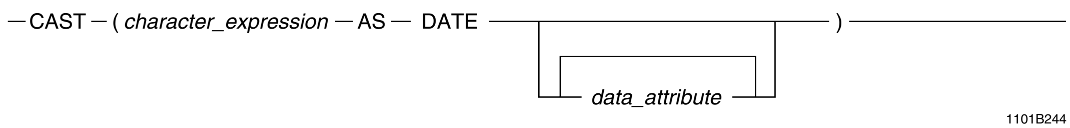
The result is Latin.

Character-to-DATE Conversion

Purpose

Converts a character string to a date value.

CAST Syntax



where:

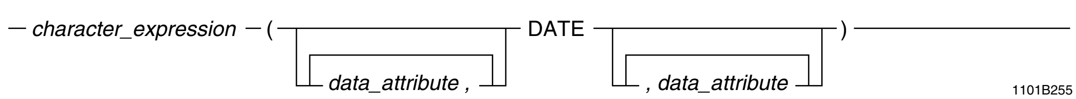
Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a DATE value.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the date data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a DATE value.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Implicit Character-to-DATE Conversion

If the string does not represent a valid date, an error is reported.

In record or indicator mode, when the DateForm mode of the session is set to ANSIDate, the string must use the ANSI DATE format.

Usage Notes

The character expression is trimmed of leading and trailing pad characters and handled as if it were a quoted string in the declaration of a DATE literal.

Character-to-DATE conversion is supported for CHAR and VARCHAR types only. The source character type cannot be CLOB.

If the string can be converted to a valid DATE, then it is. Otherwise, an error is returned.

Character String Format

If the dateform of the current session is INTEGERDATE, the date representation in the character string must match the DATE output format according to the rules in the following table:

<table>
<tr><th>IF the statement ...</th><th>THEN ...</th></tr>
<tr><td>specifies a FORMAT phrase for the DATE</td><td>the character string must match that DATE format.</td></tr>
<tr><td>does not specify a FORMAT phrase</td><td>
<table>
<tr><th>IF the DATE column definition ...</th><th>THEN the character string must match ...</th></tr>
<tr><td>specifies a FORMAT phrase</td><td>that DATE format.</td></tr>
<tr><td>does not specify a FORMAT phrase</td><td>‘YY/MM/DD’, or the current setting of the default date format in the specification for data formatting (SDF) file</td></tr>
</table>
</td></tr>
</table>

For an example, see "Example 1: IntegerDate Dateform Mode" on page 600.

If the dateform of the current session is ANSIDATE, the date representation in the character string must match the DATE output format according to the rules in the following table:

<table>
<tr><th>IF the statement ...</th><th>THEN ...</th></tr>
<tr><td>specifies a FORMAT phrase for the DATE</td><td>the character string must match that DATE format.</td></tr>
<tr><td>does not specify a FORMAT phrase</td><td>
<table>
<tr><th>IF in ...</th><th>THEN ...</th></tr>
<tr><td>field mode</td><td>
<table>
<tr><th>IF the DATE column definition ...</th><th>THEN the character string must match ...</th></tr>
<tr><td>specifies a FORMAT phrase</td><td>that DATE format.</td></tr>
<tr><td>does not specify a FORMAT phrase</td><td>the ANSI format ('YYYY-MM-DD')</td></tr>
</table>
</td></tr>
<tr><td>record or indicator mode</td><td>the character string must match the ANSI format ('YYYY-MM-DD')</td></tr>
</table>
</td></tr>
</table>

For an example, see "Example 2: ANSIDate Dateform Mode" on page 601.

Forcing a FORMAT on CAST for Converting Character to DATE

You can use a FORMAT phrase to convert a character string that does not match the format of the target DATE data type. A character string in a conversion that does not specify a FORMAT phrase uses the output format for the DATE data type.

For example, suppose the session dateform is INTEGERDATE and the default DATE format of the system is set to 'yyyymmdd' through the tdlocaledef utility. The following statement fails, because the character string contains separators, which does not match the default DATE format:

```
SELECT CAST ('2005-01-01' AS DATE);
```

To override the default DATE format, and convert a character string that contains separators, specify a FORMAT phrase for the DATE target type:

```
SELECT CAST ('2005-01-01' AS DATE FORMAT 'YYYY-MM-DD');
```

In character-to-DATE conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- EEEE
- E4
- EEE
- E3

For more information on default formats and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Character Strings That Omit Day, Month, or Year

If the character string in a character-to-DATE conversion omits the day, month, or year, the system uses default values for the target DATE value.

IF the character string omits the …	THEN the system uses the …
day	value of 1 (the first day of the month).
month	value of 1 (the month of January).
year	current year.

Consider the following table:

```
CREATE TABLE date_log
   (id INTEGER
   ,start_date DATE
   ,end_date DATE
   ,log_date DATE);
```

The following INSERT statement converts three character strings to DATE values. The first character string omits the day, the second character string omits the month, and the third character string omits the year. Assume the current year is 1992.

```
INSERT date_log
   (1001
   ,CAST ('January 1992' AS DATE FORMAT 'MMMMBYYYY')
   ,CAST ('1992-01' AS DATE FORMAT 'YYYY-DD')
   ,CAST ('01/01' AS DATE FORMAT 'MM/DD'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM date_log;

         id  start_date  end_date  log_date
-----------  ----------  --------  --------
       1001    92/01/01  92/01/01  92/01/01
```

Example 1: IntegerDate Dateform Mode

For example, suppose the session dateform is INTEGERDATE, and the default DATE format of the system is set to 'yyyymmdd' through the tdlocaledef utility.

Consider the following table, where the start_date column uses the default DATE format and the end_date column uses the format 'YYYY/MM/DD':

```
CREATE TABLE date_log
   (id INTEGER
   ,start_date DATE
   ,end_date DATE FORMAT 'YYYY/MM/DD');
```

The following INSERT statement works because the character strings match the formats of the corresponding DATE columns and Teradata Database can successfully perform implicit character-to-DATE conversion:

```
INSERT INTO date_log (1099, '20030122', '2003/01/23');
```

To perform character-to-DATE conversion on character strings that do not match the formats of the corresponding DATE columns, you must use a FORMAT phrase:

```
INSERT INTO date_log
   (1047
   ,CAST ('Jan 12, 2003' AS DATE FORMAT 'MMMBDD,BYYYY')
   ,CAST ('Jan 13, 2003' AS DATE FORMAT 'MMMBDD,BYYYY'));
```

Example 2: ANSIDate Dateform Mode

Suppose the session dateform is ANSIDATE. The default DATE format of the system is 'YYYY-MM-DD'.

Consider the following table, where the start_date column uses the default DATE format and the end_date column uses the format 'YYYY/MM/DD':

```
CREATE TABLE date_log
   (id INTEGER
   ,start_date DATE
   ,end_date DATE FORMAT 'YYYY/MM/DD');
```

The following INSERT statement works because the character strings match the formats of the corresponding DATE columns and Teradata Database can successfully perform implicit character-to-DATE conversion:

```
INSERT INTO date_log (1099, '2003-01-22', '2003/01/23');
```

To perform character-to-DATE conversion on character strings that do not match the formats of the corresponding DATE columns, you must use a FORMAT phrase:

```
INSERT INTO date_log
   (1047
   ,CAST ('Jan 12, 2003' AS DATE FORMAT 'MMMBDD,BYYYY')
   ,CAST ('Jan 13, 2003' AS DATE FORMAT 'MMMBDD,BYYYY'));
```

Example 3: Implicit Character-to-DATE Conversion

Assume that the DateForm mode of the session is set to ANSIDate.

The following CREATE TABLE statement specifies a FORMAT phrase for the DATE data type column:

```
CREATE SET TABLE datetab (f1 DATE FORMAT 'MMM-DD-YYYY');
```

In field mode, the following INSERT statement successfully performs the character to DATE implicit conversion because the format of the string conforms to the format of the DATE column in the datetab table:

```
INSERT INTO datetab ('JAN-10-1999');
```

In record or indicator mode, when the DateForm mode of the session is set to ANSIDate, the following INSERT statement successfully performs the character to DATE implicit conversion because the format of the string is in the ANSI DATE format:

```
INSERT INTO datetab ('2002-05-10');
```

Related Topics

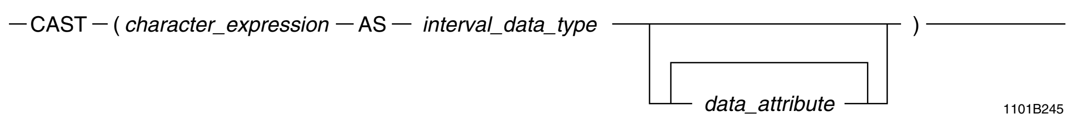
For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-INTERVAL Conversion

Purpose

Converts a character string to an interval value.

CAST Syntax



where:

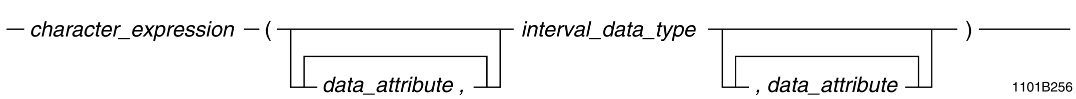
Syntax element ...	Specifies ...
character_expression	a character expression to be cast to an INTERVAL value.
interval_data_type	an INTERVAL data type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI SQL, Teradata supports the specification of data attributes.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to an INTERVAL value.
data_attribute	one of the following optional data attributes: • NAMED • TITLE

Syntax element ...	Specifies ...
interval_data_type	an INTERVAL data type to which *character_expression* is to be converted.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

The character value is trimmed of leading and trailing pad characters and handled as if it were a quotestring in the declaration of an INTERVAL string literal.

Character-to-INTERVAL conversion is supported for CHAR and VARCHAR types only. The source character type cannot be CLOB.

If the contents of the character string can be converted to a valid INTERVAL, then they are; otherwise, an error is returned.

You cannot convert a character data type of GRAPHIC to an INTERVAL string literal.

Example 1

The following query returns ' -265-11'.

```
SELECT CAST('-265-11' AS INTERVAL YEAR(4) TO MONTH);
```

Example 2

If the source character string contains values not normalized in the INTERVAL form, but which nevertheless can be converted to a proper INTERVAL, the conversion is made.

For example, the following query returns '-267-06'

```
SELECT CAST('265-30' AS INTERVAL YEAR(4) TO MONTH);
```

Related Topics

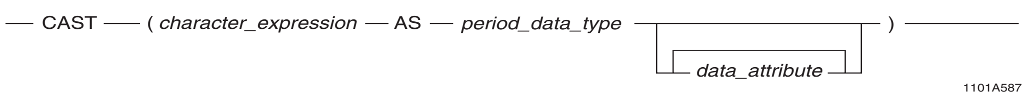
For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-Period Conversion

Purpose

Converts a character string to a Period value.

CAST Syntax



where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a Period value.
period_data_type	Period data type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

Usage Notes

A character value expression can be cast as PERIOD(DATE), PERIOD(TIME), or PERIOD(TIMESTAMP) using the CAST function or implicit casting. A character input value can also be implicitly cast as a Period type.

After any leading and trailing pad characters in the source character value are trimmed, the resulting character string must conform to the format of the target type. Conversion of the beginning and ending portions of the character value expression to corresponding DateTime values follow the existing rules of CHARACTER/VARCHAR to DateTime data type conversions.

The existing rules include conversion of the source value with a TIME or TIMESTAMP format to UTC based on the specified time zone in the source or, if not specified, the current session time zone. The exception to conversion to UTC for Period data types is when the ending portion of the source character is a TIMESTAMP value without a time zone and the value is equal to the maximum value that is used to represent UNTIL_CHANGED; in this case, the value is not changed to UTC.

If the target type has a TIME or TIMESTAMP element type and the beginning or ending bound portions of the character value expression contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.

If target type has a TIME or TIMESTAMP element type and the target precision is lower than either precision specified in the source character string, an error is reported. If the target precision is higher than a precision specified for a bound in the source character string, trailing zeros are added to the fractional seconds of the corresponding bound of the Period value resulting from the cast.

The target elements are set to the corresponding resulting values.

If the result beginning bound is not less than the result ending bound in their UTC forms, an error is reported.

If an ANSI DateTime format is used to interpret the character data during conversion, then enclosing the beginning and ending values inside quotation marks is optional. For details, see "Character Strings that Use ANSI DateTime Format" on page 607.

Implicit Character-to-Period Conversion

A CHARACTER or VARCHAR value is implicitly cast as a Period data type for an assignment, update, insert, merge, or parameter passing operation when the target site has a Period data type and for a comparison operation if the other operand has a Period data type. If any other non-Period value is directly assigned to a Period target site, an error is reported. In the same manner, if any other non-Period value is directly compared to a Period value, an error is reported.

Note: In some cases, a value may be explicitly cast as a Period data type in order to avoid this error.

During implicit conversion from CHARACTER or VARCHAR to Period data type, the ANSI DateTime format string is used to interpret the beginning and ending element values in the character string, if the response mode is other than the Field mode or if the character string data is parameterized. If the response mode is Field mode and if the character string data is not parameterized, then the target period format is used to interpret the beginning and ending element values in the character string. The following table describes this in detail.

Mode	Parameterized Data Present	Format for Implicit Cast Interpretation
Field	No	Target format
Field	Yes	ANSI format
Non-field	Yes	ANSI format
Non-field	No	ANSI format

When the ANSI DateTime format string is used to interpret the beginning and ending element values in the character string, enclosing the beginning and ending values inside the quotes is optional. This relaxation applies even during an explicit cast. For details, see "Character Strings that Use ANSI DateTime Format" on page 607.

Character Strings that Use ANSI DateTime Format

Here is a list of valid character string representations when the implicit or explicit character-to-period conversion uses the ANSI DateTime format to interpret the beginning and ending bound elements.

- `'(''`*`beginning_element_value`*`'',Δ''`*`ending_element_value`*`'')'`
- `'(`*`beginning_element_value`*`,Δ`*`ending_element_value`*`)'`
- `'(''`*`beginning_element_value`*`'',Δ`*`ending_element_value`*`)'`
- `'(`*`beginning_element_value`*`,Δ''`*`ending_element_value`*`'')'`

where formats of *beginning_element_value* and *ending_element_value* depend on the target data type.

Target Data Type	Format
PERIOD(DATE)	YYYY-MM-DD
PERIOD(TIME[(n)])	HH:MI:SS.S(F)
PERIOD(TIMESTAMP[(n)])	YYYY-MM-DDBHH:MI:SS.S(F)

For the meanings of the format characters, see the description of the FORMAT phrase in *SQL Data Types and Literals.*

Example

In the following example, two concatenated character literals are cast as PERIOD(TIMESTAMP(2)). The output is adjusted according to the current session time zone during display. Assume the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE and the format derived from SDF is 'YYYY-MM-DDBHH:MI:SS.S(2)Z'.

```
SELECT CAST('(''2005-02-02 12:12:12.34+08:00'', ' ||
       '''2006-02-03 12:12:12.34+08:00'')'
       AS PERIOD(TIMESTAMP(2)));
```

The following PERIOD(TIMESTAMP(2)) value is returned:

```
('2005-02-01 20:12:12.34', '2006-02-02 20:12:12.34')
```

Related Topics

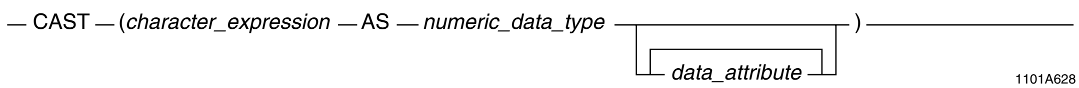
For details on data types and data attributes, see *SQL Data Types and Literals.*

Character-to-Numeric Conversion

Purpose

Converts a character data string to a numeric value.

CAST Syntax



where:

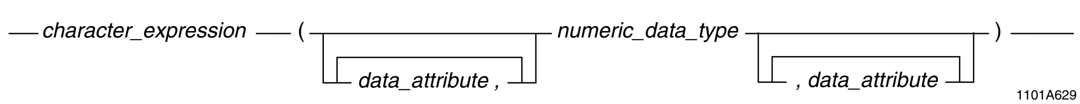
Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a numeric type.
numeric_data_definition	the numeric type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the numeric data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a numeric type.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
numeric_data_type	the numeric type to which *character_expression* is to be converted.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Implicit Character-to-Numeric Conversion

Implicit character to numeric conversion produces a valid result only if the character string represents a numeric value.

If a CHAR or VARCHAR character string is present in an expression that requires a numeric operand, it is read as a formatted numeric and is converted to a FLOAT value, using the default format for FLOAT.

To override the implicit format, use a FORMAT phrase.

Or, to change the default format for FLOAT, you can change the setting of the *REAL* element in the specification for data formatting (SDF) file. For information on default data type formats, the SDF file, and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

To use a CLOB type in an expression that requires a numeric operand, you must first explicitly convert the CLOB to CHAR or VARCHAR.

An empty character string (zero length) or a character string consisting only of pad characters is interpreted as having a numeric value of zero.

If the default format for FLOAT is -9.99E-99, then:

THIS expression ...	IS converted to ...	AND the result is ...
1.1*'$20.00'	1.10E 00*2.00E1	2.20E 01
'2'+'2'	2.00E 00+2.00E 00	4.00E 00
'A' + 2	----------	error

If a column or parameter of numeric data type is specified with a string value, the string is again assumed to be a formatted numeric. For example, the following INSERT statement specifies the Salary as a numeric string:

```
INSERT INTO Employee (EmpNo, Name, Salary)
VALUES (10022, 'Clements D', '$38,000.00');
```

The conversion to numeric type removes editing symbols. When selected, the salary data contains only the special characters allowed by the FORMAT phrase for Salary in the CREATE TABLE statement. If the FORMAT phrase is 'G-(9)D9(2)', then the output looks like this:

```
   Salary
---------
38,000.00
```

If the FORMAT phrase is 'G-L(9)D9(2)', then the output looks like this:

```
    Salary
----------
$38,000.00
```

Supported Character Types

The character expression to be converted must be CHAR or VARCHAR. CLOBs cannot be explicitly converted to numeric types.

Usage Notes

Before processing begins, the numeric description is scanned for a FORMAT phrase, which is used to determine the radix separator, group separator, currency sign or string, signzone (S), or implied decimal point (V) formatting.

Conversion is performed positionally, character by character, from left to right, until the end of the number.

Only all-numeric character strings can be converted from character to numeric formats. For example, you can convert the character strings 'US Dollars 123456' or '123456' to the integer value 123456, but you cannot convert the string 'EX1AM2PL3E' to a numeric value.

The following list shows the steps for converting character type data to numeric. Note that you cannot convert a *character_expression* of GRAPHIC character type to numeric.

Conversion is performed stage by stage, without returning to a previous stage; however, stages can be skipped.

1. Leading pad characters are ignored. Trailing pad characters are ignored, except for signed zoned decimal input.

   Embedded spaces are only allowed according to the following rules:

   - If the current SDF file defines the group separator as a space, then the character string can include spaces to separate groups of digits to the left of the radix separator, according to the grouping rule defined by *GroupingRule* or *CurrencyGroupingRule*.
   - If the current SDF file defines the radix separator as a space, then the character string can include one space as the radix character.
   - If the FORMAT phrase contains a currency formatting character, such as N, and the matching currency string in the SDF file, such as *CurrencyName*, contains a space, the character string can include spaces as part of that currency string.

2 The sign (+ or -) is saved as part of the number. A mantissa sign may appear before the first digit in the string, or after the last digit in the string. An exponent sign may appear with a preceding mantissa sign.

3 The currency sign is ignored if it matches the FORMAT. A currency string is ignored if it matches the FORMAT. Only one currency is allowed in the string.

4 Digits are saved as the integral, fractional, or exponent part of the number, depending on whether the radix or the letter E has been parsed.

5 Separators are ignored, unless they match the radix specified in the FORMAT.

   If a separator matches the radix specified in the FORMAT, the location is saved as the beginning of the fractional part of the number. V marks the fractional component for implied decimals.

   The allowance of currency and separators is a non-ANSI Teradata extension of character to numeric conversion.

6 Embedded dashes (between digits) are allowed, unless the number is signed or includes a radix, currency, or exponent.

7 The letter E is saved as the beginning of the exponent part of the number. One space is allowed following an E.

8 The exponent sign (+ or -) is saved.

9 The exponent digits are saved. A sign character cannot appear after any exponent digit.

Numeric Overflow

In Field Mode, numeric overflow in character to numeric conversion is not treated as an error. If the result exceeds the number of digits normally reserved for the data type, asterisks are displayed.

In Record and Indicator Variable Modes, numeric overflow is reported as an error. This behavior applies to both the CAST and Teradata conversion syntax.

FORMAT Phrase Controls Parsing of the Data

A FORMAT phrase, by itself, cannot convert a character type value to a numeric type value. The phrase controls partially how the resultant value is parsed.

Some examples of character to numeric conversion appear in the following table. For FORMAT phrases that contain G, D, C, and N formatting characters, assume that the related entries in the specification for data formatting file (SDF) are:

```
RadixSeparator {"."}
GroupSeparator {","}
GroupingRule   {"3"}
Currency       {"$"}
ISOCurrency    {"USD"}
CurrencyName   {"US Dollars"}
```

Character String	Converted To	Resultant Numeric Value	Field Mode Display Result
'$20,000.00'	DECIMAL(10,2)	20000.00	20000.00
'$$$.50'	DECIMAL(10,2)	error[a]	error
'$.50'	DECIMAL(8,2)	.50	.50
'.345'	DECIMAL(8,3)	.345	.345
'-1.234E-02'	FLOAT	-.01234	-.01234
'-1E.-2'	FLOAT	error[b]	error
'00000000-.93'	DECIMAL(12,4)	error[c]	error
'- 55'	INTEGER	-55	-55
'E67'	FLOAT	0.0	0.00000000000000E 000
'9876'	DECIMAL(4,2) FORMAT '99V99'	98.76	9876
'-123'	INTEGER	-123	-123
'9876'	DECIMAL(4,2) FORMAT '9(2)V9(2)'	98.76	9876
'1-2-3'	INTEGER	123	123
'123-'	INTEGER	-123	-123
'123- '	INTEGER	-123	-123
'-1.234E 02'	FLOAT	-123.4	-1.23400000000000E 002
'111,222,333'	INTEGER FORMAT 'G9(I)'	111222333	0,111,222,333
'2.49US Dollars'	DECIMAL(10,2) FORMAT 'GZ(I)D9(F)BN'	2.49	2.49 US Dollars
'25000USD'	INTEGER FORMAT '9(I)C'	25000	0000025000USD

a. Only one currency is allowed in the character string.

b. The radix must precede the exponent part of the number.

c. Embedded dashes cannot appear in a string containing a radix.

A conversion that does not specify a FORMAT phrase uses the corresponding data type default format as defined in the SDF.

For more information on default data type formats, the SDF file, and the meaning of formatting characters in a FORMAT phrase, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Example: Implicit Conversion of Character to Numeric

The INSERT statement in the following example implicitly converts the character data type to the target numeric data type:

```
CREATE TABLE t1
(f1 DECIMAL(10,2) FORMAT 'G-U(9)D9(2)');

INSERT t1 ('USD12,345,678.90');
```

If a column definition in a CREATE TABLE statement does not specify a FORMAT phrase for the data type, the column uses the corresponding data type default format as defined in the specification for data formatting (SDF) file. For more information on the default format of data types and the definition of formatting characters in a FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals.*

Related Topics

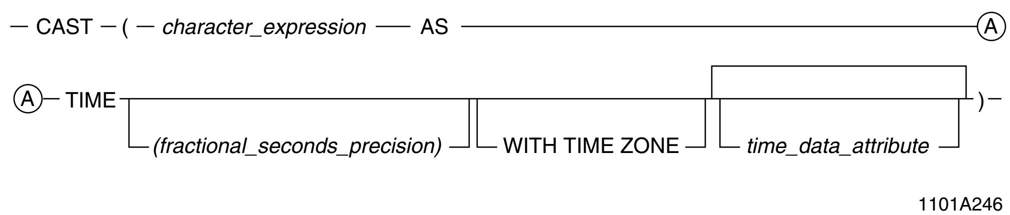
For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-TIME Conversion

Purpose

Converts a character data string to a TIME or TIME WITH TIME ZONE value.

CAST Syntax



where:

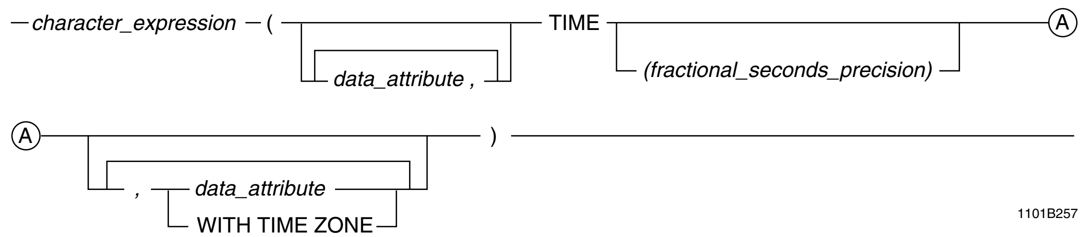
Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a TIME type.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative output formatting for the time data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a TIME type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Implicit Character-to-TIME Conversion

In field mode, the string must conform to the format of the target TIME type.

In record or indicator mode, the string must use the ANSI TIME format.

Usage Notes

The character value is trimmed of leading and trailing pad characters and handled as if it were a quoted string in the declaration of a TIME string literal.

If the contents of the string can be converted to a valid TIME, the conversion is made; otherwise, an error is returned to the application.

Character-to-TIME conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or GRAPHIC to TIME.

You can use a FORMAT phrase to specify an explicit format for the TIME target data type. A conversion that does not specify a FORMAT phrase uses the default format for the TIME data type.

IF the character string is converted to …	THEN the default format …
TIME	does not use the time zone formatting character and does not display a time zone.
TIME WITH TIME ZONE	uses the time zone formatting character to display the time zone.

For more information on default formats and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Conversions That Include Time Zone

The following rules apply to character-to-TIME conversions that include time zone information:

- If the target data type does not specify a time zone, for example, TIME(0), the source character string may contain a time zone of the format +hh:mi or -hh:mi, but only if it appears immediately before or immediately after the time.

  For example, the following conversion is successful:

  ```
  SELECT CAST ( '-02:0011:23:44'
  AS TIME(0) );
  ```

  The following conversion is not successful because of the blank separator character between the time zone and the time:

  ```
  SELECT CAST ( '+02:00 11:23:44.56'
  AS TIME(2) );
  ```

- If the source character string contains a time zone, and the target data type does not specify a time zone, for example, TIME(0), the conversion uses the time zone in the character string to convert the character string to Universal Coordinated Time (UTC). This is done regardless of whether the FORMAT phrase contains the time zone formatting character.

  ```
  SELECT CAST ('10:15:12+12:30'
  AS TIME(0));
  ```

- If the source character string does not contain a time zone, and the target data type specifies a time zone and a target FORMAT phrase that includes time zone formatting characters, the output includes the session time zone.

  ```
  SELECT CAST ('10:15:12'
  AS TIME(0) WITH TIME ZONE FORMAT 'HH:MI:SSBZ');
  ```

Conversions That Include Fractional Seconds

The following rules apply to conversions that include fractional seconds:

- The fractional seconds precision in the source character string must be less than or equal to the fractional seconds precision specified by the target type.

  ```
  SELECT CAST('12:30:25.44' AS TIME(3));
  ```

  If no fractional seconds appear in the source character string, then the fractional seconds precision is always less than or equal to the target data type fractional seconds precision, because the valid range for the precision is zero to six, where the default is six.

  ```
  SELECT CAST('12:30:25' AS TIME(3));
  ```

- If the target data type is defined by a FORMAT phrase, the fractional seconds precision formatting characters must be greater than or equal to the precision specified by the data type.

  ```
  SELECT CAST('12h:15.12s:30m'
  AS TIME(4) FORMAT 'HHh:SSDS(4)s:MIm');
  ```

  A FORMAT phrase must specify a fractional seconds precision of six if the target data type does not specify a fractional seconds precision, because the default precision is six.

  ```
  SELECT CAST ('12:30:25' AS TIME FORMAT 'HH:MI:SSDS(6)');
  ```

Character Strings That Omit Hour, Minute, or Second

If the character string in a character-to-TIME conversion omits the hour, minute, or second, the system uses default values for the target TIME value.

IF the character string omits the …	THEN the system uses the …
hour	value of 0.
minute	
second	

Consider the following table:

```
CREATE TABLE time_log
   (id INTEGER
   ,start_time TIME
   ,end_time TIME
   ,log_time TIME);
```

The following INSERT statement converts three character strings to TIME values. The first character string omits the hour, the second character string omits the minute, and the third character string omits the second.

```
INSERT time_log
   (1001
   ,CAST ('01:02.030405' AS TIME FORMAT 'MI:SS.S(6)')
   ,CAST ('01:02.030405' AS TIME FORMAT 'HH:SS.S(6)')
   ,CAST ('01:02' AS TIME FORMAT 'HH:MI'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM time_log;
```

```
         id       start_time         end_time         log_time
-----------  ---------------  ---------------  ---------------
       1001  00:01:02.030405  01:00:02.030405  01:02:00.000000
```

FORMAT Phrase Restrictions

In character-to-TIME conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- Z
- T

Example 1: Fractional Seconds

This query returns the value ‘12:23:39.999900’ (with the fractional seconds extended to 6 places as requested by CASTing to a TIME(6) type).

```
SELECT CAST(' 12:23:39.9999 '
AS TIME(6));
```

Example 2: Truncation of Non-pad Character Data

This query returns an error because the requested conversion requires truncation of non-pad character data.

```
SELECT CAST(' 12:23:39.9999 '
AS TIME(3));
```

Example 3: Invalid MINUTE Value

This query returns an error because the MINUTE value of 63 is not valid.

```
SELECT CAST(' 12:63:39.9999 '
AS TIME(6));
```

Example 4: FORMAT Phrase

This query returns the value '15h33m'.

```
SELECT CAST('15h33m'
AS TIME(0) FORMAT 'HHhMIm');
```

Example 5: Implicit Conversion of Character to TIME

The following CREATE TABLE statement specifies a FORMAT phrase for the TIME data type column:

```
CREATE SET TABLE timetab (f1 TIME(0) FORMAT 'TBHHhMImSSs');
```

In field mode, the following INSERT statement successfully performs the character to TIME implicit conversion because the format of the string conforms to the format of the TIME column in the timetab table:

```
INSERT INTO timetab ('AM 10h20m30s');
```

In record or indicator mode, the following INSERT statement successfully performs the character to TIME implicit conversion because the format of the string is in the ANSI TIME format:

```
INSERT timetab ('11:23:34');
```

Related Topics

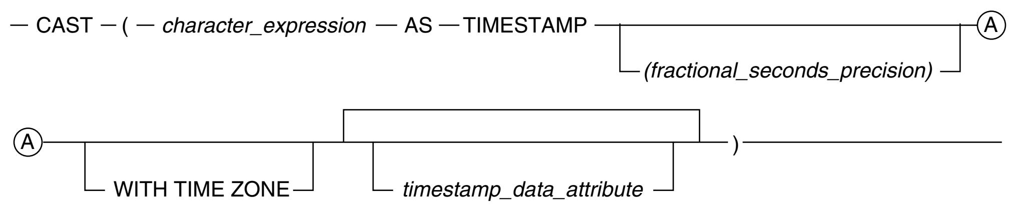
For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-TIMESTAMP Conversion

Purpose

Converts a character data string to a TIMESTAMP or TIMESTAMP WITH TIME ZONE value.

CAST Syntax



1101A247

where:

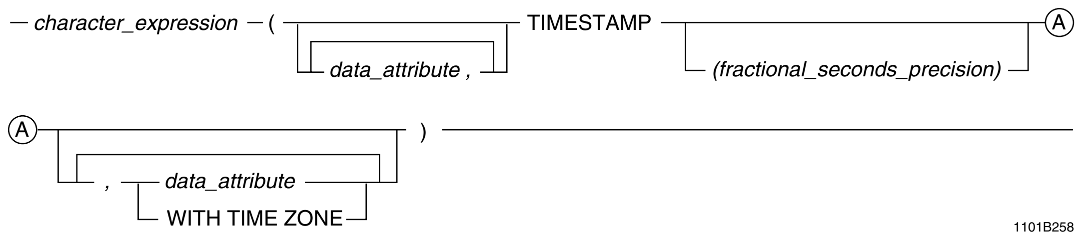
Syntax element …	Specifies …
character_expression	a character expression to be cast to a TIMESTAMP type.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
timestamp_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the time data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a TIMESTAMP type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Implicit Character-to-TIMESTAMP Conversion

In field mode, the string must conform to the format of the target TIMESTAMP type.

In record or indicator mode, the string must use the ANSI TIMESTAMP format.

Usage Notes

The source expression is trimmed of leading and trailing pad characters and then handled as if it were a quoted string in the declaration of a TIMESTAMP string literal.

Character-to-TIMESTAMP conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or GRAPHIC to TIMESTAMP.

If the contents of the string can be converted to a valid TIMESTAMP value, then the conversion is performed; otherwise an error is returned.

You can use a FORMAT phrase to specify an explicit format for the TIMESTAMP target data type. A conversion that does not specify a FORMAT phrase uses the default format for the TIMESTAMP data type.

IF the character string is converted to …	THEN the default format …
TIMESTAMP	does not use the time zone formatting character and does not display a time zone.
TIMESTAMP WITH TIME ZONE	uses the time zone formatting character to display the time zone.

For more information on default formats and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Example

The following query returns ‘2007-12-31 23:59:59.999999-08:00’.

```
SELECT CAST('2007-12-31 23:59:59.999999'
AS TIMESTAMP(6) WITH TIME ZONE);
```

Notice that the source character string did not need to have explicit Time Zone fields for this conversion to work properly.

Conversions That Include Time Zone

The following rules apply to character-to-TIMESTAMP conversions that include time zone information:

- If the target data type does not specify a time zone, for example, TIMESTAMP(0), the source character string may contain a time zone of the format +hh:mi or -hh:mi, but only if it appears immediately before or immediately after the time.

  For example, the following conversion is successful:

  ```
  SELECT CAST ( '2008-09-19 11:23:44-02:00'
  AS TIMESTAMP(0) FORMAT 'Y4-MM-DDBHH:MI:SSBZ' );
  ```

  The following conversion is not successful because of the blank separator character between the time zone and the time:

  ```
  SELECT CAST ( '2008-01-19 +02:00 11:23:44'
  AS TIMESTAMP(0) FORMAT 'Y4-MM-DDBZBHH:MI:SS' );
  ```

- If the source character string contains a time zone, and the target data type does not specify a time zone, the conversion uses the time zone in the character string to convert the character string to Universal Coordinated Time (UTC). This is done whether or not the FORMAT phrase contains the time zone formatting character.

  ```
  SELECT CAST ('2002-02-20 10:15:12+12:30' AS TIMESTAMP(0));
  ```

- If the target FORMAT phrase includes time zone formatting characters, and the source character string does not contain a time zone, the output includes the session time zone. This is done whether or not the target data type specifies a time zone.

  ```
  SELECT CAST ('2002-02-20 10:15:12'
  AS TIMESTAMP(0) WITH TIME ZONE FORMAT 'Y4-MM-DDBHH:MI:SSBZ');
  ```

Conversions That Include Fractional Seconds

The following rules apply to conversions that include fractional seconds:

- The fractional seconds precision in the source character string must be less than or equal to the fractional seconds precision specified by the target type.

  ```
  SELECT CAST('2002-01-01 12:30:25.44' AS TIMESTAMP(3));
  ```

  If no fractional seconds appear in the source character string, then the fractional seconds precision is always less than or equal to the target data type fractional seconds precision, because the valid range for the precision is zero to six, where the default is six.

  ```
  SELECT CAST('2002-01-01 12:30:25' AS TIMESTAMP(3));
  ```

- If the target data type is defined by a FORMAT phrase, the fractional seconds precision formatting characters must be greater than or equal to the precision specified by the data type.

  ```
  SELECT CAST('12-02-07 12:30:25' AS TIMESTAMP(3)
  FORMAT 'DD-MM-YYBHH:MI:SSDS(3)');
  ```

  A FORMAT phrase must specify a fractional seconds precision of six if the target data type does not specify a fractional seconds precision, because the default precision is six.

  ```
  SELECT CAST('12-02-07 12h:15.12s:30m'
  AS TIMESTAMP FORMAT 'DD-MM-YYBHHh:SSDS(6)s:MIm');
  ```

Character Strings That Omit Day, Month, Year, Hour, Minute, or Second

If the character string in a character-to-TIMESTAMP conversion omits the day, month, year, hour, minute, or second, the system uses default values for the target TIMESTAMP value.

IF the character string omits the …	THEN the system uses the …
day	value of 1 (the first day of the month).
month	value of 1 (the month of January).
year	current year.
hour	value of 0.
minute	
second	

Consider the following table:

```
CREATE TABLE timestamp_log
(id INTEGER, start_ts TIMESTAMP, end_ts TIMESTAMP);
```

The following INSERT statement converts two character strings to TIMESTAMP values. Both strings omit the hour, minute, and second. Additionally, the first character string omits the day and the second character string omits the month.

```
INSERT timestamp_log
   (1001
   ,CAST ('January 2006' AS TIMESTAMP FORMAT 'MMMMBYYYY')
   ,CAST ('2006-01' AS TIMESTAMP FORMAT 'YYYY-DD'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM timestamp_log;

         id                    start_ts                      end_ts
-----------  --------------------------  --------------------------
       1001  2006-01-01 00:00:00.000000  2006-01-01 00:00:00.000000
```

Here is an INSERT statement where both character strings omit the year. Additionally, the first character string omits the hour and the second character string omits the minute. Assume the current year is 2003.

```
INSERT timestamp_log
   (1002
   ,CAST ('January 23 04:05' AS TIMESTAMP FORMAT 'MMMMBDDBMI:SS')
   ,CAST ('01-23 04:05' AS TIMESTAMP FORMAT 'MM-DDBHH:SS'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM timestamp_log WHERE id = 1002;

         id                    start_ts                      end_ts
-----------  --------------------------  --------------------------
       1001  2003-01-23 00:04:05.000000  2003-01-23 04:00:05.000000
```

Restrictions on FORMAT Phrase

In character-to-TIMESTAMP conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- EEEE
- E4
- EEE
- E3
- T
- Z

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-UDT Conversion

Purpose

Converts a character data string to a UDT.

CAST Syntax

```
── CAST ── (character_expression ── AS ── UDT_data_definition ) ──
```

1101A336

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a UDT.
UDT_data_definition	the UDT type to which *character_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit character-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Character-to-UDT Conversion

Teradata Database performs implicit Character-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

The source character type of the cast definition does not have to be an exact match to the source character type of the implicit conversion. Teradata Database can use an implicit cast definition that specifies a CHAR, VARCHAR, or CLOB source type.

If multiple implicit cast definitions exist for converting different character types to the UDT, Teradata Database uses the implicit cast definition for the character type with the highest precedence. The following list shows the precedence of character types in order from lowest to highest precedence:

- CHAR
- VARCHAR
- CLOB

For non-CLOB character types, if no Character-to-UDT implicit cast definitions exist, Teradata Database looks for other cast definitions that can substitute:

IF the following combination of implicit cast definitions exists ...				THEN Teradata Database ...
Numeric-to-UDT	DATE-to-UDT	TIME-to-UDT	TIMESTAMP-to-UDT	
X				uses the numeric-to-UDT implicit cast definition. If multiple numeric-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.
	X			uses the DATE-to-UDT implicit cast definition.
		X		uses the TIME-to-UDT implicit cast definition.
			X	uses the TIMESTAMP-to-UDT implicit cast definition.
X	X			reports an error.
X		X		
X			X	
	X	X		
	X		X	
		X	X	
X	X	X		
X	X		X	
X		X	X	
	X	X	X	
X	X	X	X	

Substitutions are valid because Teradata Database can implicitly cast the non-CLOB character type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character Data Type Assignment Rules

Server Character Sets

LATIN, UNICODE, KANJISJIS, KANJI1, and GRAPHIC server character sets are generally mutually assignable.

Consider an assignment of an expression to a character string column. The assignment may be the result of the SQL UPDATE or INSERT statement, or it may be the result of a Load utility assignment.

The expression is converted to the server character set of the character column.

Exceptions to GRAPHIC Data

The following exceptions apply to GRAPHIC data:

- When you import GRAPHIC data and assign it to a character column, that column must be defined as GRAPHIC.
- When you import character data that is not GRAPHIC, you cannot assign it to a column defined as GRAPHIC.

  For more information, see the documentation on the USING row descriptor in *SQL Data Manipulation Language*.
- You cannot assign non-GRAPHIC data to a GRAPHIC column from BTEQ or load utilities.

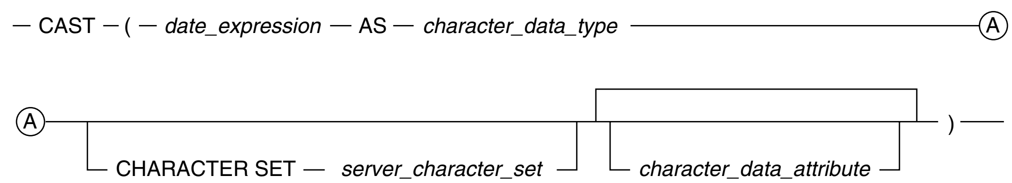
  For more information, see the documentation on the USING row descriptor in *SQL Data Manipulation Language*.
- You cannot assign or export GRAPHIC data from a single byte character set like ASCII or EBCDIC.

DATE-to-Character Conversion

Purpose

Converts a DATE value to a character string.

CAST Syntax



1101A248

where:

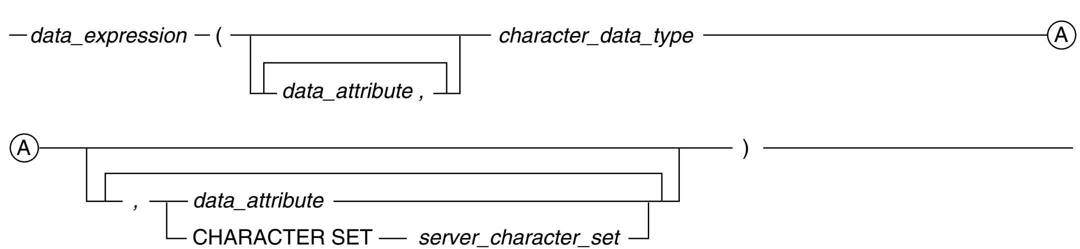
Syntax element ...	Specifies ...
date_expression	a date expression to be cast to a character string.
character_data_type	the character data type to which *date_expression* is to be converted.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

Teradata Conversion Syntax



1101B259

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be cast to a character string.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character data type to which *date_expression* is to be converted.
server_character_set	the server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default character set is used for the conversion.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

When converting DATE to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the DATE value as represented by a character string literal.

IF the target data type is ...	AND *n* is ...	THEN ...
CHAR(*n*)	greater than the length of the DATE value as represented by a character string literal	trailing pad characters are added to pad the representation.
	too small	a string truncation error is returned.
VARCHAR(*n*)	greater than the length of the DATE value as represented by a character string literal	no blank padding is added to the character representation.
	too small	a string truncation error is returned.

Restrictions

DATE types cannot be implicitly or explicitly converted to character types if the server character set is GRAPHIC.

DATE to CLOB conversion is not supported.

Forcing a FORMAT on CAST for Converting DATE to Character

The default format for DATE to character conversion uses the format in effect for the DATE value.

To override the default format, you can convert a DATE value to a string using a FORMAT phrase. The resulting format, however, is the same as the DATE value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from DATE to CHAR and force an explicit FORMAT on the result regardless of the format associated with the DATE value. This is because of the rules for matching FORMAT phrases to data types.

Example 1

The dateform mode of the session is INTEGERDATE and column F1 in the table INTDAT is a DATE value with the explicit format 'YYYY,MMM,DD'.

```
SELECT F1 FROM INTDAT ;
```

The result (without a type change) is the following report:

```
F1
----------
1900,Dec,31
```

Assume that you want to convert this to a value of CHAR(12), and an explicit output format of 'MMMBDD,BYYYY'. Use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```
SELECT
CAST( (CAST (F1 AS FORMAT 'MMMBDD,BYYYY')) AS CHAR(12))
FROM INTDAT;
```

The result after the nested CASTs is the following report.

```
F1
------------
Dec 31, 1900
```

The inner CAST establishes the display format for the DATE value and the outer CAST indicates the data type of the desired result.

Example 2

Suppose you need to create a script to convert date values to the ANSI DATE format, regardless of the source of the DATE value or the DATEFORM mode of the session.

You can use nested CASTs and a FORMAT to do this as demonstrated by the example that follows.

```
SELECT
CAST( (CAST (F1 AS FORMAT 'YYYY-MM-DD')) AS CHAR(10))
FROM INTDAT;
```

The result after the nested CASTs is the following report.

```
F1
----------
1900-12-31
```

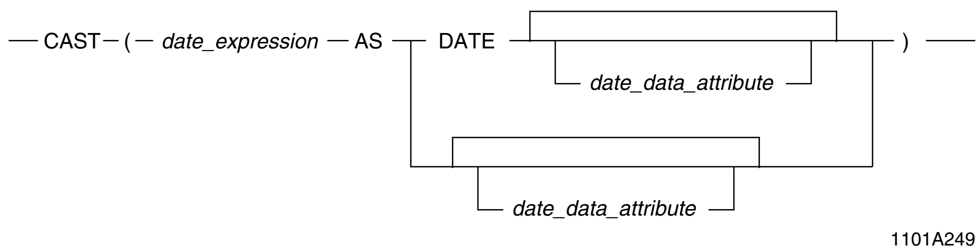
Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-DATE Conversion

Use DATE-to-DATE conversion to convert the format or title of a DATE type.

CAST Syntax



where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
date_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

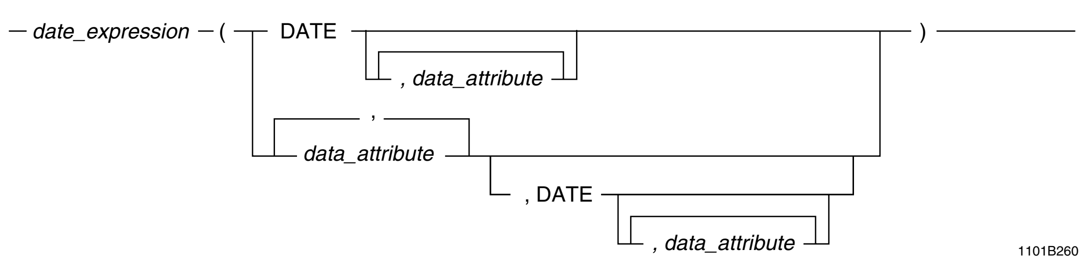
ANSI Compliance

CAST is ANSI SQL:2008 compliant.

The following are Teradata extensions to CAST:

- CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative output formatting of date data.
- A DATE-to-DATE conversion involving a DATE type with a dateform of INTEGERDATE is a Teradata extension to the ANSI SQL:2008 standard.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Example

Consider a table named employee that was created with a session dateform mode of INTEGERDATE where dob is a DATE column with a format of M3BDDBY4. To list employees who were born between January 30, 1938, and March 30, 1943, you could specify the date information as follows:

```
SELECT name, dob
FROM employee
WHERE dob BETWEEN 'Jan 30 1938' AND 'Mar 30 1943'
ORDER BY dob;
```

The result returns the date of birth information as specified for the Employee table:

```
Name                  DOB
----------    -----------
Inglis C      Mar 07 1938
Peterson J    Mar 27 1942
```

To change the date format to an alternate form, change the SELECT to:

```
SELECT name, dob (FORMAT 'yy-mm-dd')
FROM employee
WHERE dob BETWEEN 'Jan 30 1938' AND 'Mar 30 1943'
ORDER BY dob ;
```

The format specification changes the display to the following:

```
Name               DOB
----------    --------
Inglis C      38-03-07
Peterson J    42-03-27
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

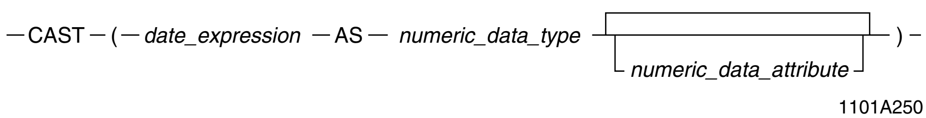
DATE-to-Numeric Conversion

Introduction

DATE data may be converted to the following numeric types:

- SMALLINT
- BYTEINT
- INTEGER
- BIGINT
- DECIMAL(n,m)
- FLOAT

CAST Syntax



where:

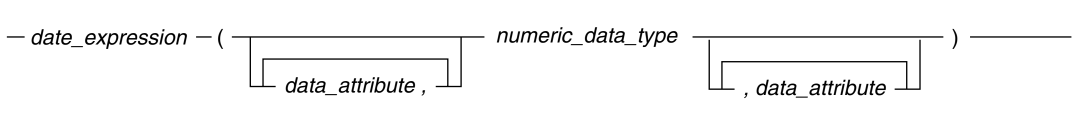
Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
numeric_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of numeric data attribute phrases.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
numeric_data_type	the target numeric type to which the date expression is to be converted.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

When a date is converted to a numeric, the value returned is the integer value for the internal stored date, which is encoded using the following formula:

```
(year - 1900) * 10000 + (month * 100) + day
```

Allowable date values range from AD January 1, 0001 to AD December 31, 9999.

For example, December 31, 1985 would be stored as the integer 851231; July 4, 1776 stored as -1239296; and March 30, 2041 stored as 1410330.

Conversion of DATE to DECIMAL(n,m) where the number of digits (n) is too small generates a numeric overflow error. Conversion of DATE to BYTEINT or SMALLINT generates a numeric overflow error if the value returned is outside the range of values that the data type can represent.

No error is generated on conversion of DATE to INTEGER or FLOAT.

FORMAT Phrase

A FORMAT phrase in DATE to numeric conversion may only contain the 9 or Z formatting character. For example:

```
SELECT CAST (DATE '2007-12-31' AS INTEGER FORMAT '9999999');
```

Implicit DATE-to-Numeric Conversion

Teradata Database performs implicit DATE-to-numeric type conversion when you assign a DATE type to a numeric type, compare a DATE type and numeric type, or pass a DATE type to a system function that takes a numeric type.

Example

The following example converts DATE data in the dob column of the employee table to a numeric format.

Note that the best practice is to define date data as a DATE type; do not define date data as a numeric type.

To change the display from date format to integer format, change the statement to:

```
SELECT name, dob (INTEGER)
FROM employee
WHERE dob BETWEEN 380307 AND 420825
ORDER BY dob ;
```

or

```
SELECT name, CAST (dob AS INTEGER)
FROM employee
WHERE dob BETWEEN 380307 AND 420825
ORDER BY dob ;
```

and the display becomes:

```
Name             DOB
----------    ------
Inglis C      380307
Peterson J    420327
```

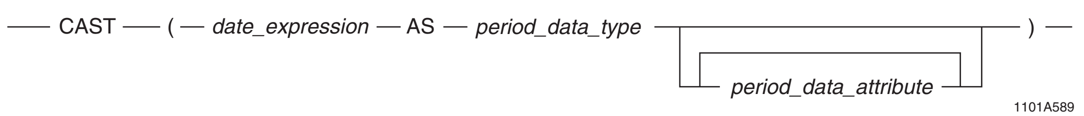
Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-Period Conversion

Casts as PERIOD(DATE) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]).

CAST Syntax



where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
period_data_type	the target Period type to which the date expression is to be converted.
period_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Usage Notes

A DATE value can be cast as PERIOD(DATE) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function. If an attempt is made to cast a DATE value as PERIOD(TIME[(n)] [WITH TIME ZONE]), an error is reported.

If the target type is PERIOD(DATE), the result beginning element is set to the source value. The result ending element is set to the result beginning bound plus one granule of the target type (that is, INTERVAL '1' DAY). If the result ending bound exceeds the maximum DATE value (that is, the source value is equal to the maximum DATE value), or the result ending bound equal to maximum DATE value (that is, the resulting ending bound value equal to value of UNTIL_CHANGED) an error is reported.

If the target type is PERIOD(TIMESTAMP[(n)]), the result beginning element is set to the UTC value obtained using the current session time zone and a timestamp value formed from the source DATE value and a time portion of zero. The result ending element is set to the result beginning bound plus one granule of the target type (note that this cannot cause an error).

If the target type is PERIOD(TIMESTAMP[(n)] WITH TIME ZONE), the time portion of the result beginning element is set to the UTC value obtained using the current session time zone and a timestamp value formed from the source DATE value and a time portion of zero. The time zone of the result beginning element is set to the current session time zone displacement. The result ending element is set to the result beginning bound plus one granule of the target type (note that this cannot cause an error).

Note: The result has the same value for the beginning bound and last value.

Example 1

In the following example, a DATE literal is cast as PERIOD(DATE). The result beginning bound is obtained from the source. The result ending element is set to the result beginning bound plus INTERVAL '1' DAY.

```
SELECT CAST(DATE '2005-02-03' AS PERIOD(DATE));
```

The following PERIOD(DATE) value is returned:

```
('2005-02-03', '2005-02-04')
```

Example 2

In the following example, a DATE literal is cast as PERIOD(TIMESTAMP(4)). The result beginning bound is formed from the DATE literal and a time portion of zero. The result ending element is set to the result beginning bound plus INTERVAL '0.0001' SECOND.

```
SELECT CAST(DATE '2005-02-03' AS PERIOD(TIMESTAMP(4)));
```

The following PERIOD(TIMESTAMP(4)) value is returned:

```
('2005-02-03 00:00:00.0000', '2005-02-03 00:00:00.0001')
```

Related Topics

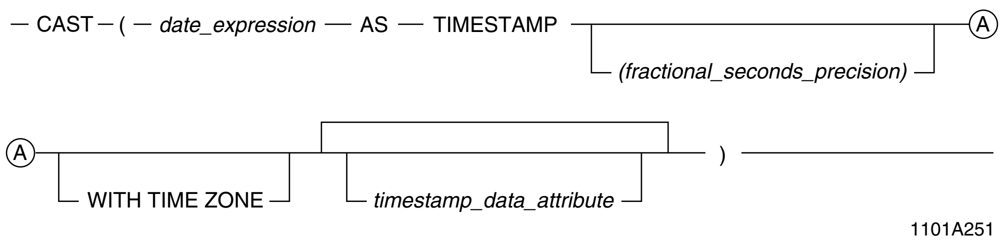
For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-TIMESTAMP Conversion

Purpose

Converts a DATE value to a TIMESTAMP or TIMESTAMP WITH TIME ZONE value.

CAST Syntax



where:

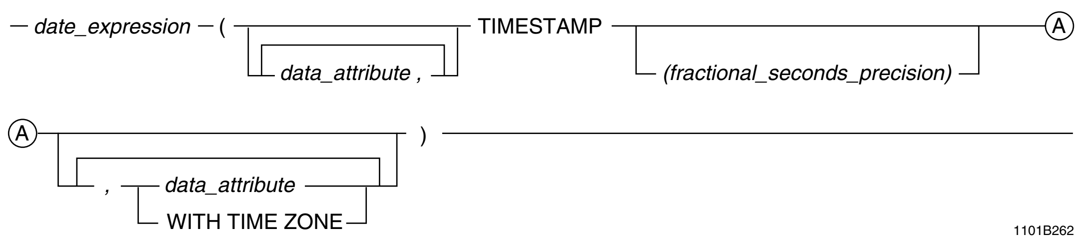
Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
timestamp_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting of timestamp data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Implicit DATE-to-TIMESTAMP Conversion

Teradata Database performs implicit conversion from DATE to TIMESTAMP types in some cases. See "Implicit Conversion of DateTime types" on page 580.

The following conversions are supported:

From source type...	To target type...
DATE[a]	TIMESTAMP
	TIMESTAMP WITH TIME ZONE

a. ANSIDate dateform mode or IntegerDate dateform mode

Conversion from DATE to TIMESTAMP types is performed as follows:

- The YEAR, MONTH, and DAY fields are set to the values from the source DATE record.

- The HOUR, MINUTE, and SECONDS fields are set to zeros.
- If WITH TIME ZONE is specified, then the time zone displacement of the current session is used.

The TIMESTAMP value is always converted to DATE in case of comparison. See “TIMESTAMP-to-DATE Conversion” on page 705.

Example

Assuming that the current date is July 10, 2008 and the local time zone is -8, the following query returns ‘2008-10-07 00:00:00.000-08:00’.

```
SELECT CAST(CURRENT_DATE AS TIMESTAMP(3) WITH TIME ZONE);
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-UDT Conversion

Purpose

Converts DATE data to UDT data.

CAST Syntax

```
── CAST ── ( date_expression ── AS ── UDT_data_definition ) ──
```

1101A337

where:

Syntax element ...	Specifies ...
date_expression	a DATE expression to be cast to a UDT.
UDT_data_definition	the UDT type to which *date_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit DATE-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit DATE-to-UDT Conversion

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

Teradata Database performs implicit DATE-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

If no DATE-to-UDT implicit cast definition exists, Teradata Database looks for other cast definitions that can substitute:

IF the following combination of implicit cast definitions exists …		THEN Teradata Database …
Numeric-to-UDT	Character[a]-to-UDT	
X		uses the Numeric-to-UDT implicit cast definition. If multiple Numeric-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.
	X	uses the Character-to-UDT implicit cast definition. If multiple Character-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.
X	X	reports an error.

a. a non-CLOB character type

Substitutions are valid because Teradata Database can implicitly cast a DATE type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

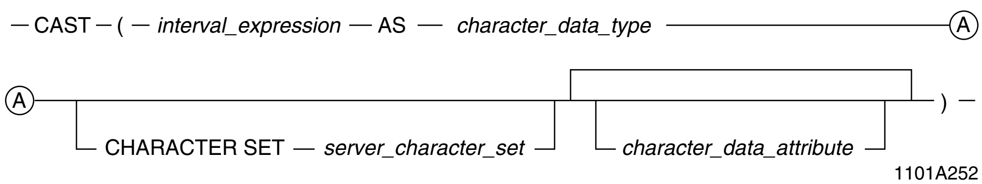
INTERVAL-to-Character Conversion

Purpose

Use CAST syntax or Teradata explicit conversion syntax to convert an INTERVAL type to its canonical character string representation.

INTERVAL-to-Character conversion is supported for CHAR and VARCHAR types only. The target type cannot be CLOB.

CAST Syntax



where:

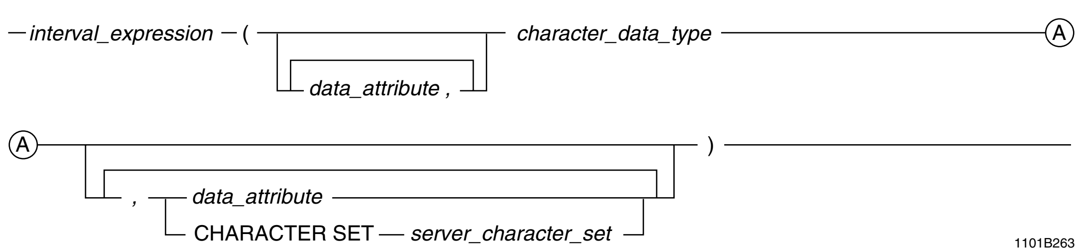
Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.
character_data_type	the target character type to which the interval expression is to be converted.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the target character type to which the interval expression is to be converted.
server_character_set	which server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default character set is used to convert the INTERVAL expression.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

INTERVAL-to-Fixed CHARACTER Conversion

When the target data type is CHAR(*n*), then *n* must be equal to or greater than the length of the canonical form of the value as represented by a character string literal.

If *n* is greater than that length, trailing pad characters are added to pad the canonical representation.

If *n* is too small, then a string truncation error is returned.

INTERVAL-to-VARCHAR Conversion

When the target data type is VARCHAR(*n*), then *n* must be equal to or greater than the length of the canonical form of the value as represented by a varying character string literal.

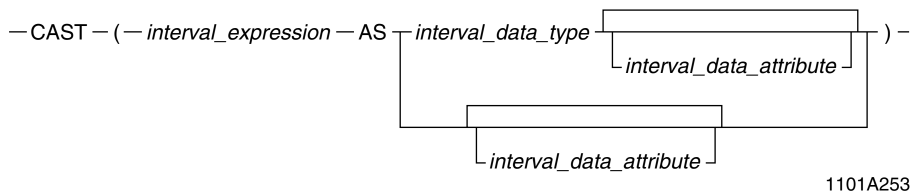
If *n* is too small, then a string truncation error is returned.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

INTERVAL-to-INTERVAL Conversion

CAST Syntax



where:

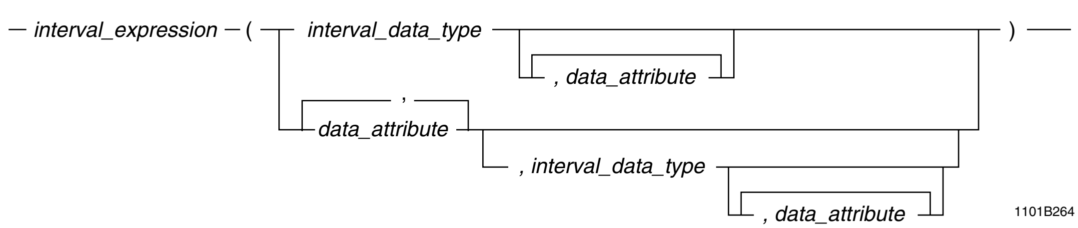
Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.
interval_data_type	the target INTERVAL type to which the interval expression is to be converted.
interval_data_attribute	one of the following optional data attributes: • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.

Syntax element ...	Specifies ...
interval_data_type	the optional target INTERVAL type to which the interval expression is to be converted.
data_attribute	one of the following optional data attributes: • NAMED • TITLE

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Compatible Types

Both data types must be from the same INTERVAL family: either Year-Month or Day-Time. Types cannot be mixed.

This INTERVAL data type ...	Belongs to this INTERVAL family ...
• INTERVAL YEAR • INTERVAL YEAR TO MONTH • INTERVAL MONTH	Year-Month
• INTERVAL DAY • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE • INTERVAL MINUTE TO SECOND • INTERVAL SECOND	Day-Time

Conversion of INTERVAL types is performed only when the fields and precisions are different.

Precision of Source and Target Types

A conversion can result in an overflow error if the precision of the target data type is smaller than the corresponding precision for the source data type.

If the least significant value of the source is lower than that of the target, then those source values having lower precision than the least significant field of the target are ignored. The result is truncation. Recovery from this action is installation-dependent.

If the most significant field in the source value has higher significance than the most significant field in the target value, then the higher order fields of the source are converted into a scalar value of the precision of the most significant field in the target, using the factors of 12 months per year, 24 hours per day and so on.

If the compared scalar value overflows the defined precision for the target field, an error is returned.

Implicit INTERVAL-to-INTERVAL Conversion

Teradata Database performs implicit conversion from INTERVAL to INTERVAL data types in some cases. See "Implicit Conversion of DateTime types" on page 580.

Conversion of INTERVAL types is performed only when both data types are from the same INTERVAL family: either Year-Month or Day-Time. See "Compatible Types" on page 647.

Example 1: Least Significant Field in Source Lower Than Target

The following query converts ' 3-11' to ' 3'. Source is INTERVAL YEAR(2). The truncation completes the conversion.

```
SELECT CAST(INTERVAL '3-11' YEAR TO MONTH AS INTERVAL YEAR(2));
```

Example 2: Least Significant Field in Source Lower Than Target

The following query converts ' 135 12:37:25.26' to '3252'. Source is DAY(3) TO SECOND(2)

```
SELECT CAST(INTERVAL '135 12:37:25.26' DAY(3) TO SECOND(2) AS INTERVAL
HOUR(4));
```

Example 3: Least Significant Field in Source Higher Than Target

The following query converts '3' to '3-00'. Source is INTERVAL YEAR. The insertion of zeros completes the conversion.

```
SELECT CAST(INTERVAL '3' YEAR AS INTERVAL YEAR TO MONTH);
```

Example 4: Least Significant Field in Source Higher Than Target

The following query converts ' 135 00:00:00.0' to ' 3240:00:00.00' after you perform the additional conversion of multiplying 135 * 24 to obtain 3240, which is the final HOUR value. The source had a data type of DAY.

```
SELECT CAST(INTERVAL ' 135 00:00:00.0' DAY AS INTERVAL HOUR TO SECOND);
```

Example 5: Most Significant Field in Source Higher Than Target

The following query first treats the source INTERVAL value as '135 12' and then computes HOURS as (135*24)+12=3252. The result of the query is INTERVAL '3252' HOUR unless the precision for the target value is less than 4, in which case an error is returned. The source had a data type of DAY TO SECOND.

```
SELECT CAST(INTERVAL '135 12:37:25.26' DAY TO SECOND AS INTERVAL HOUR);
```

Example 6: Implicit Type Conversion During Assignment

Consider the following table which has an INTERVAL YEAR TO MONTH column:

```
CREATE TABLE TimeInfo
  (YrToMon INTERVAL YEAR TO MONTH);
```

If you insert data into the column using the following parameterized request, and you pass an INTERVAL YEAR or INTERVAL MONTH value to the request, Teradata Database implicitly converts the value to an INTERVAL YEAR TO MONTH value before inserting the value.

```
INSERT INTO TimeInfo
VALUES (?);
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

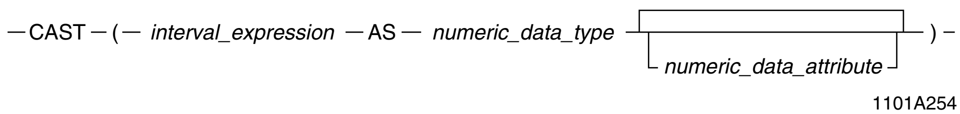
INTERVAL-to-Numeric Conversion

Purpose

Convert an INTERVAL with only one field to an exact numeric data type.

This numeric value is the value of the single numeric field in the INTERVAL record.

CAST Syntax



where:

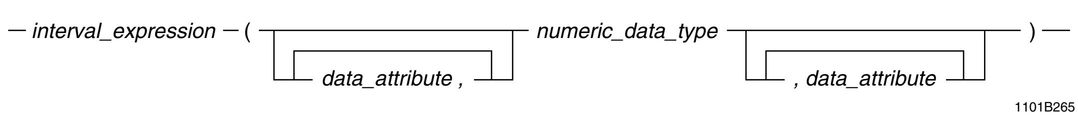
Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.
numeric_data_type	the target numeric type to which the interval expression is to be converted.
numeric_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
numeric_data_type	the target numeric type to which the interval expression is to be converted.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Implicit INTERVAL-to-Numeric Conversion

Teradata Database performs implicit conversion of an Interval data type to an exact numeric data type in some cases. See"Implicit Conversion of DateTime types" on page 580.

Example

Consider the following table definition:

```
CREATE TABLE sales_intervals
   ( sdate DATE
   , sinterval INTERVAL MONTH
   , stotals DECIMAL(5,0));
```

The following query uses CAST to convert INTERVAL MONTH values in the sinterval column to INTEGER.

```
SELECT stotals,
   (EXTRACT (MONTH FROM sdate)) + (CAST(sinterval AS INTEGER))
FROM sales_intervals;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

INTERVAL-to-UDT Conversion

Purpose

Converts interval data to UDT data.

CAST Syntax

```
── CAST ── ( interval_expression ── AS ── UDT_data_definition ) ──
```

1101A338

where:

Syntax element ...	Specifies ...
interval_expression	an interval expression to be cast to a UDT.
UDT_data_definition	the UDT type to which *interval_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit INTERVAL-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit INTERVAL-to-UDT Conversion

Performing an implicit data type conversion requires a cast definition (see "Usage Notes") that specifies the following:

- the AS ASSIGNMENT clause
- a source data type that is in the same INTERVAL family as the source of the implicit cast:

This INTERVAL data type …	Belongs to this INTERVAL family …
• INTERVAL YEAR • INTERVAL YEAR TO MONTH • INTERVAL MONTH	Year-Month
• INTERVAL DAY • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE • INTERVAL MINUTE TO SECOND • INTERVAL SECOND	Day-Time

The source data type of the cast definition does not have to be an exact match to the source of the implicit type conversion.

Teradata Database performs implicit INTERVAL-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Related Topics

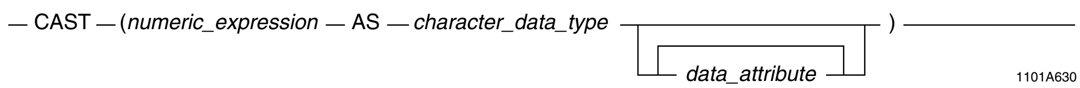
For details on data types and data attributes, see *SQL Data Types and Literals*.

Numeric-to-Character Conversion

Purpose

Converts a numeric data type to a character data type.

CAST Syntax



where:

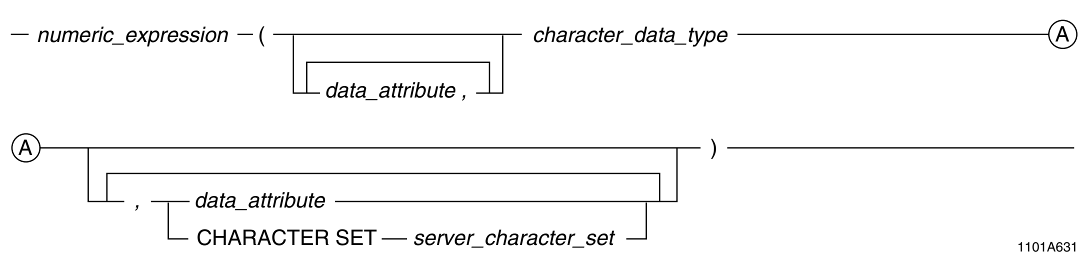
Syntax element …	Specifies …
numeric_expression	the numeric data expression to be cast to a character type.
character_data_type	the character type to which the numeric data expression is to be converted.
data_attribute	one of the following optional data attributes: • CHARACTER SET • FORMAT • NAMED • TITLE If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax



where:

Syntax element …	Specifies …
numeric_expression	the numeric data expression to be cast to a character type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character type to which the numeric data expression is to be converted. If *character_data_definition* does not specify a CHARACTER SET clause to indicate which server character set to use, the user default server character set is used.
server_character_set	which server character set to use. If the CHARACTER SET clause is not specified, the user default server character set is used.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Implicit Numeric-to-Character Conversion

If a numeric argument in an SQL string function is implicitly converted to a CHAR or VARCHAR character type, and the format of the numeric argument includes any of the following formatting characters, the server character set of the character type is UNICODE:

- G
- F
- O
- A
- D
- L
- U
- I
- C
- N

For all other formats, the server character set is LATIN.

Numeric items cannot be converted to CLOB types or GRAPHIC characters.

For information on data type formats, formatting characters, and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

How CAST Differs from Teradata Conversion Syntax

The process for the CAST function is as follows:

1. Convert the numeric value to a character string using the default or specified format for the numeric value.
2. Trim leading and trailing pad characters.
3. Extend to the right as required by the target string length.

4 If truncation of non-pad characters is required to conform to the target string length, report string truncation error.

The CAST operation differs from the Teradata SQL conversion as follows:

- Results are left justified. Column displays are not aligned.
- Truncation of significant data generates a string truncation error.

Using Teradata conversion syntax (that is, not using CAST) for explicit conversion of *numeric*-to-*character* data requires caution.

The process is as follows:

1 Convert the numeric value to a character string using the default or specified FORMAT for the numeric value.

   Leading and trailing pad characters are not trimmed.

2 Extend to the right with pad characters if required, or truncate from the right if required, to conform to the target length specification.

   If non-pad characters are truncated, no string truncation error is reported.

For an example of numeric to character conversion that results in truncation of significant data, see "Example 1" on page 657.

Supported Character Types

Numeric to character conversion is supported for CHAR and VARCHAR types only. Numeric types cannot be converted to CLOB types.

Usage Notes

To convert a numeric type value to a character string, the character description must contain a data type declaration. A FORMAT phrase, by itself, cannot be used to convert a numeric type value to a character type value. The phrase only controls how to display the resultant value.

If the character description does not include a FORMAT phrase, then the format of the original numeric value determines how to display the data.

The Teradata conversion syntax form of numeric-to-character conversion uses explicit or default FORMATs to convert to a character representation. It then truncates or extends with pad characters, depending what length the character string dictates. This can lead to a loss of significance.

Attempting to convert from a numeric type to a character type that uses a GRAPHIC server character set generates an error.

As a general rule, you should store numbers as numeric data, not as character data. For example, a table is created with the following code:

```
CREATE TABLE job AS
   (job_code CHAR(6) PRIMARY KEY
   ,description CHAR(70) );
```

Subsequently, the following query is made:

```
SELECT job_code, description
FROM job
WHERE job_code = 1234;
```

The problem here is that ‘1234’, ‘ 1234’, ‘01234’, ‘001234’, ‘+1234’, and so on, are all valid character representations of the numeric literal value, and the system cannot tell which value to use for hashing. Therefore, the system must do a full table scan to convert all job_code values to their numeric equivalents so that it can do the comparisons.

Example 1

T1.Field1 has a numeric INTEGER data type with the default format ‘-(10)9’. The user has values such as 123456, with no values of over 999999. The values, defined as being in INTEGER format, are to be converted to CHAR(8).

The following example illustrates the Teradata syntax for performing this numeric-to-character conversion.

```
SELECT Field1(CHAR(8)) FROM T1;
```

returns ‘   123’ for the value 123456, where the result includes 5 leading pad characters and truncates significant digits.

Example 2

Based on the following description of Salary, data is converted as illustrated in the following table (Δ = pad character):

```
Salary (DECIMAL(8,2), FORMAT '$$$,$$9.99')
```

Data	Conversion	Result
20000.00	Salary (CHAR(10))	'$20,000.00'
9000.00	Salary (CHAR(10))	'Δ$9,000.00'
20000.00	Salary (FORMAT'9(5)') (CHAR (5))	'20000'
9000.00	CAST (Salary AS CHAR(10))	'$9,000.00Δ'

The resultant character string is either extended with pad characters or truncated to conform to the given character description.

Example 3

Suppose EmpNo was defined as SMALLINT with the default format of ‘9(6)’. Suppose a value in EmpNo is 12501. The statement:

```
SELECT EmpNo(CHAR(5)) FROM Employee;
```

returns the ‘1250’, with a leading pad character and the low order digit missing. The CAST function used for the same conversion, converts to the character representation of the numeric value, trims leading pad characters, and finally truncates or pads on the right. For example, the following SELECT statement returns ‘12501’.

```
SELECT CAST (EmpNo AS CHAR(5)) FROM Employee;
```

Related Topics

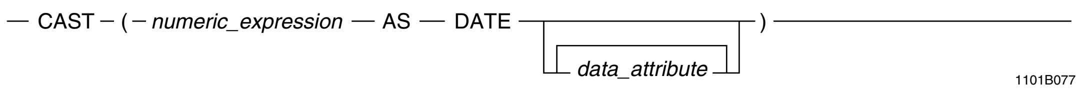
For details on data types and data attributes, see *SQL Data Types and Literals*.

Numeric-to-DATE Conversion

Purpose

Converts a numeric expression to a DATE data type.

CAST Syntax



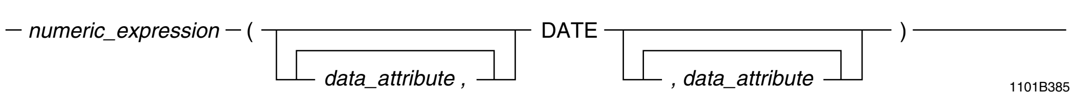
where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.
data_attribute	any of the following optional data attributes: • FORMAT • NAMED • TITLE A *date_data_definition* that specifies a FORMAT clause enables an alternative format. Specifying data attributes in CAST is a non-ANSI Teradata extension.

ANSI Compliance

CAST is ANSI SQL:2008 compliant; however, converting a numeric type to a date type is a Teradata extension to the ANSI SQL:2008 standard.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE Specifying a FORMAT clause enables an alternative format.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Translation of Numbers to Dates

Although not recommended, you can explicitly convert numbers to dates.

Teradata Database stores each DATE value as a four-byte integer using the following formula:

```
(year - 1900) * 10000 + (month * 100) + day
```

For example, December 31, 1985 would be stored as the integer 851231; July 4, 1776 stored as -1239296; and March 30, 2041 stored as 1410330.

The following table demonstrates how numeric dates are interpreted when inserted into a column. Note the translation of the third date, which was probably intended to be 1990-12-01.

This numeric value ...	Translates to this date value ...
901201	1990-12-01
1001201	2000-12-01
19901201	3890-12-01

Notice that this formula best fits two-digit dates in the 1900s. Because of the difficulty of using this format outside of the 1900s, dates are best specified as ANSI date literals instead.

Range of Allowable Values

Allowable date values range from AD January 1, 0001 (-18989899) to AD December 31, 9999 (80991231).

If the numeric value does not represent a valid date, an error is reported.

Numeric-to-DATE Implicit Type Conversion

Although not recommended, you can specify a numeric type in the assignment of a DATE type. Teradata Database performs implicit numeric-to-DATE type conversion prior to the assignment. The value of the numeric type must represent a valid date.

However, for comparison operations involving a numeric type operand and a DATE type operand, Teradata Database converts the DATE type to a numeric type. If you compare a numeric type and a DATE type and expect the comparison to be between two DATE types, you must explicitly convert the numeric type to a DATE type.

Example

This example casts the numeric integer expression to a date format.

```
SELECT CAST (1071201 AS DATE);
```

The result looks like this when the DateForm mode of the session is set to ANSIDate:

```
   1071201
----------
2007-12-01
```

Related Topics

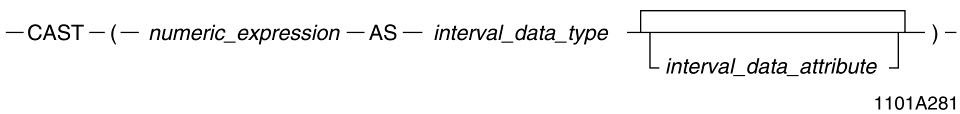
FOR information on …	SEE …
implicit type conversion of operands for comparison operations	"Implicit Type Conversion of Comparison Operands" on page 108.
data type compatibility rules for assignments involving DateTime types	"ANSI DateTime and Interval Data Type Assignment Rules" on page 152.
data type compatibility rules for arithmetic operations involving DateTime types	"Arithmetic Operators" on page 168.
data types and data attributes	*SQL Data Types and Literals*.

Numeric-to-INTERVAL Conversion

Purpose

Convert numeric data to an INTERVAL value with a single DateTime field.

CAST Syntax



where:

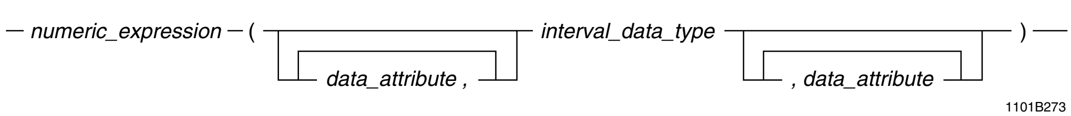
Syntax element …	Specifies …
numeric_expression	an expression or existing field having a numeric data type.
interval_data_type	the target INTERVAL data type to which the numeric expression is being converted.
interval_data_attribute	one of the following optional data attributes: • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of interval data attribute phrases.

Teradata Conversion Syntax



where:

Syntax element …	Specifies …
numeric_expression	an expression or existing field having a numeric data type.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • NAMED • TITLE
interval_data_type	the target INTERVAL data type to which the numeric expression is being converted.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

Numeric data is converted to an INTERVAL value with a single DateTime field.

If the numeric value is in the value range allowed for the INTERVAL, the value is used as the single field of the INTERVAL. Otherwise, an overflow error is returned.

Implicit Numeric-to-INTERVAL Conversion

Teradata Database performs implicit conversion of an exact numeric data type to an Interval data type in some cases. See "Implicit Conversion of DateTime types" on page 580.

Example

The following query returns '   -5' (with three leading pad characters).

```
SELECT CAST(-5 AS INTERVAL YEAR(4));
```

Related Topics

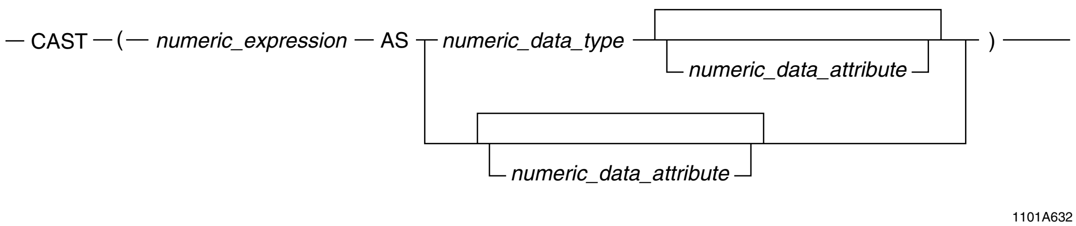
For details on data types and data attributes, see *SQL Data Types and Literals*.

Numeric-to-Numeric Conversion

Purpose

Converts a numeric expression defined with one data type to a different numeric data type.

CAST Syntax



where:

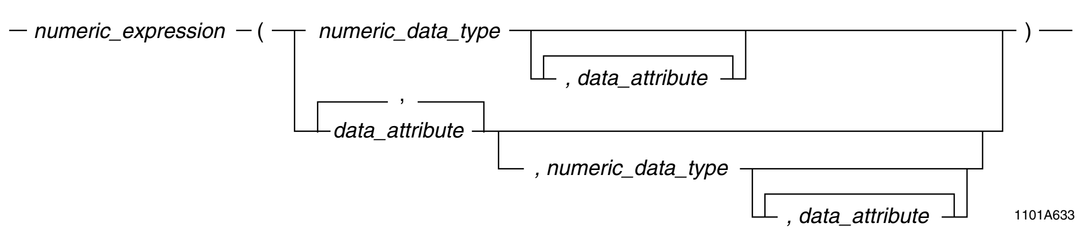
Syntax element …	Specifies …
numeric_expression	an expression or existing field having a numeric data type.
numeric_data_type	the optional numeric data type to which *numeric_expression* is to be converted.
numeric_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI SQL, CAST permits data attributes such as the FORMAT phrase that enables an alternative format for *numeric_expression*.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.
numeric_data_type	the optional numeric data type to which *numeric_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Implicit Numeric-to-Numeric Conversion

Numeric items are converted to the same numeric type before any arithmetic or comparison operation is performed. The result returned is of this same underlying type.

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.

For details on implicit type conversions for binary arithmetic expressions, see "Binary Arithmetic Result Data Types" on page 47.

For details on implicit type conversions for comparison operations, see "Implicit Type Conversion of Comparison Operands" on page 108.

Conversion to FLOAT/REAL/DOUBLE PRECISION

Because floating point numbers are not stored as exact values, conversion of DECIMAL and integer values to FLOAT values might result in a loss of precision or produce a number that cannot be represented exactly. For example, a value like 0.1, when cast to FLOAT, no longer exactly equals to 0.1.

Truncation and Rounding During Conversion

Conversion of DECIMAL/NUMERIC to BIGINT, INTEGER, BYTEINT, or SMALLINT truncates any decimal portion. Conversion to DECIMAL produces a rounded result. If a range violation occurs, the operation may fail.

Conversion to FLOAT/REAL/DOUBLE PRECISION rounds to the nearest value available. Neither decimal fractions nor numbers greater than 9,007,199,254,740,992 can be guaranteed to be represented exactly, so the nearest representable value is chosen. If there are two representable values that qualify as the nearest value, then the representation with a '0' in the least significant bit is chosen. For example, 0.1, when stored in a FLOAT column, is rounded to a value slightly higher: 0.1000000000000000055511151231257827021181583404541015625.

For details on rounding, see "Decimal/Numeric Data Types" in *SQL Data Types and Literals*.

Some examples of numeric conversions are:

Value	Converted To	Result
20000.99	INTEGER	20000
20000.99	DECIMAL(6,1)	20001.0
20000.99	DECIMAL(4, 1)	error
200000	SMALLINT	error

Using CAST in Applications With DECIMAL Type Size Restrictions

Some applications require DECIMAL types to have 15 digits or less.

Applications with this requirement may need to access DECIMAL columns that have more than 15 digits or use expressions that may produce DECIMAL results with more than 15 digits. To help with DECIMAL type size requirements, you can use CAST to convert DECIMAL types to a size of 15 or fewer digits.

For example, consider the following expression where A, B, and C are columns defined as DECIMAL(8,2):

```
SELECT (A*B)/C FROM table1;
```

The resulting value may be less than 15 digits, but A*B could be up to 18.

To ensure a result of less than 16 digits, use CAST:

```
SELECT CAST ((A*B)/C AS DECIMAL(15,2)) FROM table1;
```

Using CAST To Avoid Numeric Overflow

Because of the way the Teradata SQL compiler works, it is essential that you CAST the arguments of your expressions whenever large values are expected.

For example, suppose f1 is defined as DECIMAL(14,2) and you are going to multiply by an integer or get SUM(f1).

In this case, the following operations:

```
CAST(f1 AS DECIMAL(18,2))*100
```

or

```
SUM(CAST(f1 AS DECIMAL(18,2)))
```

are proper techniques for ensuring correct answer sets.

On the other hand, if you were to cast the *results* of the expressions, such as the following:

```
CAST(f1*100 AS DECIMAL(18,2))
```

or

```
CAST(SUM(f1) AS DECIMAL(18,2)
```

then you will likely experience overflow during the computations (and before the CAST is made)—not the desired result.

Example 1

This example casts the numeric integer expression named IntegerField to DECIMAL(7,2).

```
CAST (IntegerField AS DECIMAL (7,2))
```

Conversion of DECIMAL/NUMERIC to BIGINT, INTEGER, BYTEINT, or SMALLINT truncates any decimal portion. Conversion to DECIMAL produces a rounded result. If a range violation occurs, the operation may fail.

For details on rounding, see "Decimal/Numeric Data Types" in *SQL Data Types and Literals*.

Some examples of numeric conversions are:

Value	Converted To	Result
20000.99	INTEGER	20000
20000.99	DECIMAL(6,1)	20001.0
20000.99	DECIMAL(4, 1)	error
200000	SMALLINT	error

Example 2

Although the FORMAT phrase cannot be used to change the underlying data type defined for a column, the phrase may be used to change the display for a numeric value.

For example, if the field values for columns Wholesale and Retail, both defined as DECIMAL(7,2), are 12467.75 and 21500.50, respectively, the result of the expression:

```
CAST (Wholesale - Retail AS FORMAT '-99999')
```

is:

```
-09033
```

A FORMAT phrase does not affect data that is returned to the client system in Record Mode (client system internal format).

In the previous example, the value returned to the client system is still in packed decimal format (for example, -9032.75).

The use of FORMAT in CAST is a Teradata extension to the ANSI standard.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Numeric-to-UDT Conversion

Purpose

Converts numeric data to UDT data.

CAST Syntax

```
── CAST ── ( numeric_expression ── AS ── UDT_data_definition ) ──
```

1101A334

where:

Syntax element …	Specifies …
numeric_expression	a numeric expression to be cast to a UDT.
UDT_data_definition	the UDT type, followed by any optional FORMAT, NAMED or TITLE data attribute phrases, to which *numeric_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit numeric-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Numeric-to-UDT Conversion

Teradata Database performs implicit Numeric-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

The source numeric type of the cast definition does not have to be an exact match to the source numeric type of the implicit conversion. Teradata Database can use an implicit cast definition that specifies a BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, or REAL/FLOAT/DOUBLE target type.

If multiple implicit cast definitions exist for converting different numeric types to the UDT, Teradata Database uses the implicit cast definition for the numeric type with the highest precedence. The following list shows the precedence of numeric types in order from lowest to highest precedence:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- DECIMAL/NUMERIC
- REAL/FLOAT/DOUBLE

If no numeric-to-UDT implicit cast definitions exist, Teradata Database looks for other cast definitions that can substitute:

IF the following combination of implicit cast definitions exists ...		THEN Teradata Database ...
DATE-to-UDT	Character[a]-to-UDT	
X		uses the DATE-to-UDT implicit cast definition.
	X	uses the character-to-UDT implicit cast definition. If multiple character-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.
X	X	reports an error.

a. a non-CLOB character type

Substitutions are valid because Teradata Database can implicitly cast a numeric type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

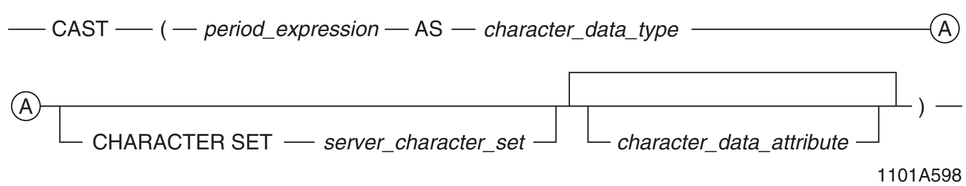
Period-to-Character Conversion

Purpose

Converts a Period data type to its canonical character string representation.

Period-to-Character conversion is supported for CHAR and VARCHAR types only. The target type cannot be CLOB.

CAST Syntax



where:

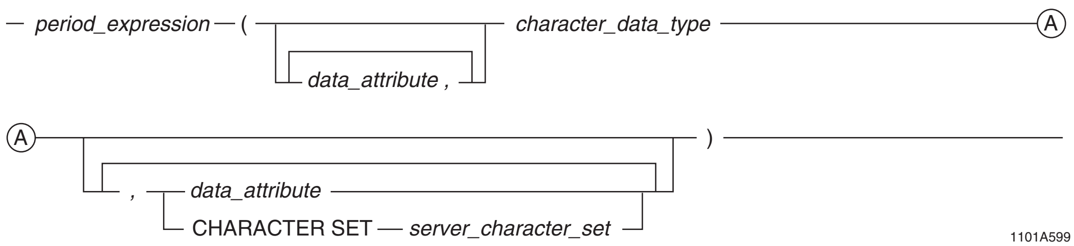
Syntax element ...	Specifies ...
period_expression	the Period data expression to be cast to a character type.
character_data_type	the character type to which the Period data expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be cast to a character type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character type to which the Period data expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

A period value expression can be cast as a character string representation using the CAST function or the Teradata cast syntax, or when forming the output for field mode. Assume *L* is the maximum length of the formatted character string for the format associated with the period value expression being cast. The resulting character string contains two strings representing the beginning and ending bounds of the period value expression, each up to length *L*, and each enclosed in single quotation marks (' '), separated by comma and a space (,), and then enclosed within a left parenthesis and a right parenthesis [( )]. Thus, the maximum length of the resulting character string is 2**L*+8. Assume the actual length is *K* (which may be less than 2**L*+8, for example, if the format includes the full names of months and the specific month for a bound is July) and the target type is CHARACTER(*n*) or VARCHAR(*n*):

- If *n* is equal to *K*, the period is cast into the resulting character string of length *K*.
- If *n* is greater than *K* and the target is VARCHAR(*n*), the period is cast into the resulting character string with length *K*.

- If *n* is greater than *K* and the target is CHARACTER(*n*), the period is cast into the resulting character string and trailing pad characters are added to extend to length *n*.
- If *n* less than *K* and the session is in ANSI mode, a truncation error is reported.
- If *n* less than *K* and the session is in Teradata mode, a truncated string of length *n* is returned.

For data of Period data types with TIME and TIMESTAMP element types, the UTC value of the Period value expression is adjusted to the time zone of the value or the current session time zone if the value does not have a time zone. The exception to conversion from UTC is for an ending bound of a PERIOD(TIMESTAMP(n)) value equal to the maximum value that is used to represent UNTIL_CHANGED; in this case, the value is not changed. Due to such adjustments, the ending bound may appear less than the beginning bound in the result, although in UTC the ending bound is greater than the beginning bound. This happens since the hour value for the TIME data type wraps over every 24 hrs (that is, the hour value is obtained using 'module 24').

Example

Assume pts is a PERIOD(TIMESTAMP(2)) column in table t with a value of PERIOD '(2005-02-02 12:12:12.34, 2006-02-03 12:12:12.34)'.

In the following example, a PERIOD(TIMESTAMP(2)) column is cast as CHARACTER(52) using the CAST function.

```
SELECT CAST(pts AS CHARACTER(52)) FROM t;
```

The following is returned:

```
('2005-02-02 12:12:12.34', '2006-02-03 12:12:12.34')
```

Related Topics

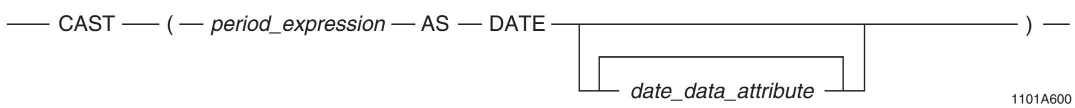
For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-DATE Conversion

Purpose

Converts Period data to a DATE value.

CAST Syntax



where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be cast to a DATE type.
date_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of DATE data attribute phrases.

Usage Notes

A PERIOD(DATE) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as DATE using the CAST function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the source type is PERIOD(DATE), the result is the source beginning bound.

If the source type is PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), the result is the date portion of the source beginning bound after adjusting to the current session time zone.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]), an error is reported.

Example

Assume pd is a PERIOD(DATE) column in table t with a value of PERIOD '(2005-02-02, 2005-02-03)'.

In the following example, a PERIOD(DATE) column is cast as DATE. The result is the beginning bound of the column.

```
SELECT CAST(pd AS DATE) FROM t;
```

The following is returned:

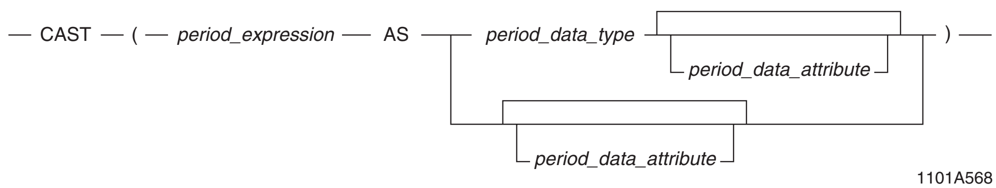
```
2005-02-02
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-Period Conversion

CAST Syntax



where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be converted.
period_data_type	the optional Period type to which *period_expression* is to be converted.
period_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Compatible Types

The following table describes the allowed combinations of source and target types when both the source and the target types are Period data types.

Source Type	Target Type
PERIOD(DATE)	PERIOD(DATE)
	PERIOD(TIMESTAMP[(m)] [WITH TIME ZONE])

Source Type	Target Type
PERIOD(TIME[(n)] [WITH TIME ZONE])	PERIOD(TIME[(m)] [WITH TIME ZONE]) where *m* is the target precision, *m* must be greater than or equal to the source precision *n*. The default for m is 6.
	PERIOD(TIMESTAMP[(m)] [WITH TIME ZONE]) where *m* is the target precision, *m* must be greater than or equal to the source precision *n*. The default for m is 6.
PERIOD(TIMESTAMP[(n)] WITH TIME ZONE)	PERIOD(DATE)
	PERIOD(TIME[(m)] [WITH TIME ZONE]) where *m* is the target precision, *m* must be greater than or equal to the source precision *n*. The default for m is 6.
	PERIOD(TIMESTAMP[(m)] [WITH TIME ZONE]) where *m* is the target precision, *m* must be greater than or equal to the source precision *n*. The default for m is 6.

PERIOD(DATE) to PERIOD(TIMESTAMP)

A PERIOD(DATE) value can be cast as PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

The UTC value of the result elements are obtained after adjustment with respect to the current session time zone from the timestamps created by setting the date portion to the corresponding source elements and the time portions to 0. If the target type is PERIOD(TIMESTAMP[(n)] WITH TIME ZONE), both result time zone fields are set to the current session time zone displacement. An exception to this is if the source ending bound is the maximum DATE value; in that case, the result ending bound is set to the maximum TIMESTAMP value.

PERIOD(TIME) to PERIOD(TIME)

A PERIOD(TIME(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) using the CAST function.

The UTC value of the source is copied to the UTC value in the result. If the target type specifies WITH TIME ZONE and the source contains time zones, the time zone displacements from the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements. For example, assume the current session time zone displacement is INTERVAL - "08:00" HOUR TO MINUTE and the source PERIOD(TIME(0) WITH TIME ZONE) has the value PERIOD '(12:12:12+08:00, 12:12:13+08:00)'. The UTC value of this source is ('04:12:12', '04:12:13'). The UTC value of the result is set to this value. On output of this result, the UTC value is adjusted to the current session time zone and the result is ('20:12:12', '20:12:13').

Note: This value is actually for a previous day and, assuming that the CURRENT_DATE at UTC is DATE '2006-07-28', the output beginning bound would be '2006-07-27 20:12:12' if it was a timestamp element.

If the target precision is higher than the source precision, trailing zeros are appended to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

PERIOD(TIME) to PERIOD(TIMESTAMP)

A PERIOD(TIME(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

The source time values get adjusted with respect to the session time zone displacement from the corresponding UTC value. The date portion of each result element is set to CURRENT_DATE. The hour, minute, and, second are copied from the source after the above adjustment and the timestamp value is converted to corresponding UTC value.

If the target type specifies WITH TIME ZONE and the source contains time zones, the time zone displacements from the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements.

If the target precision is higher than the source precision, trailing zeros are appended to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

PERIOD(TIMESTAMP) to PERIOD(DATE)

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(DATE) using the CAST function.

The result elements are each set to the date portion of the corresponding source bound after the source bound is adjusted according to the current session time zone (the adjustment is not done for the source ending bound if it is the maximum value). If the adjustment for time zone changes the date, the changed value is used. If the result date portions are the same, an error is reported.

PERIOD(TIMESTAMP) to PERIOD(TIME)

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) using the CAST function.

The date portion in the beginning and ending UTC values of the source must have the same DATE value. Otherwise, an error is reported. The time portions of the result elements are copied from the corresponding source time portions. If the target type specifies WITH TIME ZONE and the source also contains time zones, the source time zone displacements are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements.

If the target precision is higher than the source precision, trailing zeros are added to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

PERIOD(TIMESTAMP) to PERIOD(TIMESTAMP)

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

The result date and time portions are set to the corresponding source date and time portions. If the target type specifies WITH TIME ZONE and the source also contains time zones, the time zone displacements in the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements except if the source ending bound is the maximum value, the time zone for the result ending bound is +00:00.

If the target precision is higher than the source precision, trailing zeros are added in the fractional seconds. If the target precision is lower than the source precision, an error is reported.

Example 1: PERIOD(DATE) to PERIOD(TIMESTAMP)

Assume p is a PERIOD(DATE) column in table t1 with a value of PERIOD '(2005-02-02, 2006-02-03)' and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

In the following example, a PERIOD(DATE) column is cast as PERIOD(TIMESTAMP(6)). The date portion is obtained from the source for the corresponding result element and the time portions are set to zero.

```
SELECT CAST(p AS PERIOD(TIMESTAMP(6))) FROM t1;
```

The following is returned:

```
('2005-02-02 00:00:00.000000', '2006-02-03 00:00:00.000000')
```

Example 2: Least Significant Field in Source Lower Than Target

Assume p is a PERIOD(TIME(2)) column in table t with a value of PERIOD '(12:12:12.45, 13:12:12.67)' and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

In the following example, a PERIOD(TIME(2)) column is cast as PERIOD(TIME(6) WITH TIME ZONE). The time portion is obtained from the source with trailing zeros added to the fractional seconds to make the precision 6 for the corresponding result element and both result time zone fields are set to the current session time zone displacement.

```
SELECT CAST(p AS PERIOD(TIME(6)WITH TIME ZONE)) FROM t;
```

The following is returned:

```
('12:12:12.450000-08:00', '13:12:12.670000-08:00')
```

Related Topics

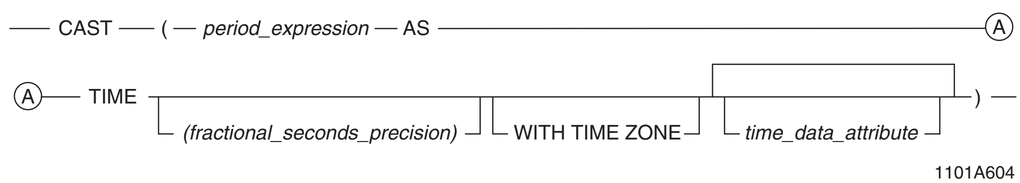
For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-TIME Conversion

Purpose

Converts Period data to a TIME value.

CAST Syntax



where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of TIME data attribute phrases.

Usage Notes

A PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as TIME[(n)] [WITH TIME ZONE] using the CAST function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is reported.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), the result time portion is obtained from time portion of the source beginning bound. If both the source and target type are WITH TIME ZONE, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is WITH TIME ZONE, the result time zone field is set to the current session time zone displacement.

If the source type is PERIOD(DATE), an error is reported.

Example

Assume pt is a PERIOD(TIME(2)) column in table t with a value of PERIOD '(12:12:12.34, 12:12:12.35)'.

In the following example, a PERIOD(TIME(2)) column is cast as TIME(6). The TIME(6) result is obtained from the source beginning element with trailing zeros added to the fractional seconds to make the precision 6.

```
SELECT CAST(pt AS TIME(6)) FROM t;
```

The following is returned:

```
12:12:12.340000
```

Related Topics

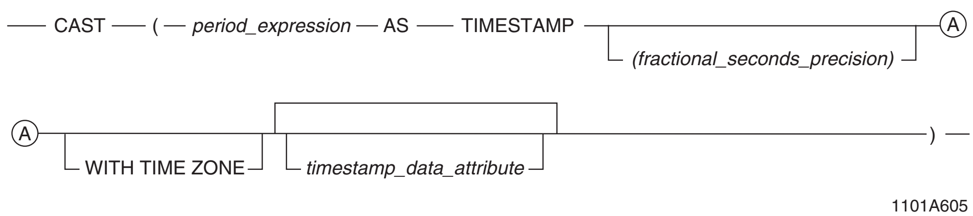
For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-TIMESTAMP Conversion

Purpose

Converts Period data to a TIMESTAMP value.

CAST Syntax



where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
timestamp_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting of DateTime data.

Usage Notes

A PERIOD(DATE), PERIOD(TIME(n) [WITH TIME ZONE]), or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as TIMESTAMP[(n)] [WITH TIME ZONE] using the CAST function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]):

- If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision.
- If the target precision is lower than the source precision, an error is reported.

If the source type is PERIOD(DATE), the result is formed from the source beginning bound and a time portion of 0 adjusted with respect to the current session time zone, and, if the target type is WITH TIME ZONE, the current session time zone displacement.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]), the source beginning bound (in UTC) is adjusted with respect to the current session time zone displacement. The timestamp portion of the result is formed from CURRENT_DATE and the time portion of the source beginning bound obtained after the above adjustment. The resulting timestamp value is converted to UTC. If both the source and target type are WITH TIME ZONE, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is WITH TIME ZONE, the result time zone field is set to the current session time zone displacement.

If the source type is PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), the result timestamp portion is the timestamp portion of the source beginning bound. If both the source and target type are WITH TIME ZONE, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is WITH TIME ZONE, the result time zone field is set to the current session time zone displacement.

Example

Assume pts is a PERIOD(TIMESTAMP(2)) column in table t with a value of PERIOD '(2005-02-03 12:12:12.34, 2005-02-03 12:12:12.35)'.

In the following example, column pts is cast as TIMESTAMP(6). The result is the source beginning bound with trailing zeros added to the fractional seconds to make the precision 6.

```
SELECT CAST(pts AS TIMESTAMP(6)) FROM t;
```

The following is returned:

```
2005-02-03 12:12:12.340000
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Signed Zone DECIMAL Conversion

Introduction

Teradata SQL can convert input data that is in signed zone (external) DECIMAL format to a NUMERIC data type, thus allowing numeric operations to be performed on row values. The column in which the signed zone decimal data is to be stored may be any numeric data type.

A FORMAT phrase incorporating the S sign character filters the data as it passes in and out of Teradata Database.

The rightmost character of the input data string is assumed to contain the zone (overpunch) bit.

The following table shows the characters representing zone-numeric combinations.

Last Character (Input String)	Numeric Conversion	Last Character (Input String)	Numeric Conversion	Last Character (Input String)	Numeric Conversion
{	n … 0	}	-n … 0	0	n … 0
A	n … 1	J	-n … 1	1	n … 1
B	n … 2	K	-n … 2	2	n … 2
C	n … 3	L	-n … 3	3	n … 3
D	n … 4	M	-n … 4	4	n … 4
E	n … 5	N	-n … 5	5	n … 5
F	n … 6	O	-n … 6	6	n … 6
G	n … 7	P	-n … 7	7	n … 7
H	n … 8	Q	-n … 8	8	n … 8
I	n … 9	R	-n … 9	9	n … 9

The sign FORMAT phrase can be included in a CREATE TABLE or ALTER TABLE statement when the column is defined, or in the INSERT statement when the data is loaded. The chosen method depends on how the stored value is to be used.

When a sign FORMAT phrase is specified at column creation time, it is considered attached to the column because it translates data at the column level; that is, both when the data is loaded and when it is retrieved.

Using FORMAT in CREATE TABLE

When the FORMAT phrase is used in the CREATE TABLE statement, as follows:

```
CREATE TABLE Test1 (Col1 DECIMAL(4) FORMAT '9999S');
```

then zoned input character strings can be loaded with standard INSERT statements, whether the data is defined:

```
INSERT INTO Test1 (Col1) VALUES ('123J');
```

or read from a client system data record via the USING modifier:

```
USING Ext1 (CHAR(4))
INSERT INTO Test1 (Col1)
VALUES (:Ext1);
```

The data record contains the string '123J'.

Subsequently, a simple select, such as:

```
SELECT Col1 FROM Test1;
```

returns:

```
Col1
----

123J
```

Using Another FORMAT in the SELECT Statement

To override an attached format, another FORMAT phrase is needed in the retrieval statement. Using the preceding table, one of the two following statements must be used to retrieve the numeric value:

```
SELECT Col1 (FORMAT '+9999') FROM Test1;
```

or

```
SELECT CAST (Col1 AS INTEGER) FROM Test1;
```

The result is as follows.

```
Col1
-----

-1231
```

If FORMAT is Not Attached to the Column

If the format is not attached to the column, the sign FORMAT phrase must be used each time signed zoned decimal data is loaded and each time the row value is to be retrieved in signed zoned decimal format.

For example, if a table is defined using a CREATE TABLE statement like this:

```
CREATE TABLE Test2 (Col2 DECIMAL(5));
```

then the sign FORMAT phrase must be included whenever signed zoned decimal strings are inserted.

This is true whether the definition is explicitly defined, as it is in Examples 1 and 2, or defined implicitly by being read from a client system data record as it is in Examples 3 and 4.

Example 1

```
INSERT INTO Test2 (Col2)
VALUES ('5678B' (DECIMAL(5), FORMAT '99999S'));
```

Example 2

```
INSERT INTO Test2 (Col2)
VALUES ('9012L' (DECIMAL(5), FORMAT '99999S'));
```

Example 3

```
USING Ext2 (CHAR(5))
INSERT INTO Test2 (Col2)
VALUES (:Ext2 (DECIMAL(5), FORMAT '99999S'));
```

Example 4

```
USING Ext2 (CHAR(5))
INSERT INTO Test2 (Col2)
VALUES (:Ext2 (DECIMAL(5), FORMAT '99999S'));
```

where Ext2 contains the strings '5678B' and '9012L'.

Because Col2 does not have an attached FORMAT phrase, a simple SELECT, such as the following example, returns the results as seen immediately following.

```
SELECT Col2 FROM Test2;

Col2
-------

 56782.
-90123.
```

A sign FORMAT phrase must be included in the SELECT statement in order to retrieve the values '5678B' and '9012L'.

It is important to remember this rule when manipulating signed zoned decimal values, especially when using sophisticated facilities like subqueries.

Example 5

This example is based on the data from Example 4.

Consider a column created with a CHARACTER data type.

```
CREATE TABLE Test3 (Col3 CHAR(5));
```

The column is loaded by selecting, without a sign FORMAT phrase, values from an "unattached" column, as follows.

```
INSERT INTO Test3 (Col3)
SELECT Col2 FROM Test2 ;
```

The values that are inserted are the following:

```
Col3
-----

 5678
-9012
```

The sign FORMAT phrase *must* be included in the query specification in order to insert the values '5678B' and '9012L'.

Related Topics

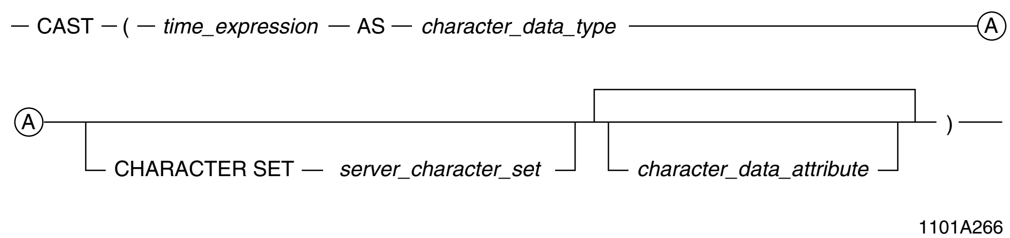
For information on data types, data type formats, formatting characters, and the FORMAT phrase, see *SQL Data Types and Literals*.

TIME-to-Character Conversion

Purpose

Convert TIME data to a character string.

CAST Syntax



where:

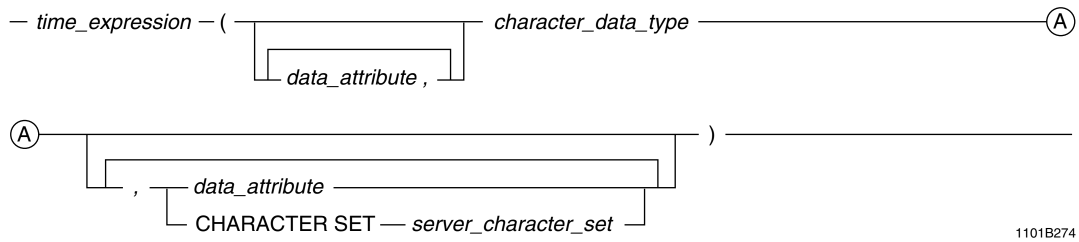
Syntax element …	Specifies …
time_expression	the TIME expression to be cast to a character type.
character_data_type	the character type to which the TIME expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting for the character representations of DateTime data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be cast to a character type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character type to which the TIME expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

When converting TIME to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the TIME value as represented by a character string literal.

IF the target data type is ...	AND *n* is ...	THEN ...
CHAR(*n*)	greater than the length of the TIME value as represented by a character string literal	trailing pad characters are added to pad the representation
	too small	a string truncation error is returned
VARCHAR(*n*)	greater than the length of the TIME value as represented by a character string literal	no blank padding is added to the character representation
	too small	a string truncation error is returned

TIME to CLOB conversion is not supported.

You cannot convert a TIME value to a character string when the server character set is GRAPHIC.

Forcing a FORMAT on CAST for Converting TIME to Character

The default format for TIME to character conversion is the format in effect for the TIME value.

You can convert a TIME value to a character string using a FORMAT phrase. The resulting format, however, is the same as the TIME value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from TIME to CHAR and force an explicit FORMAT on the result regardless of the format associated with the TIME value. This is because of the rules for matching FORMAT phrases to data types.

Example

Field T1 in the table INTTIME is a TIME(6) value with the explicit format 'HH:MI:SSDS(6)'. Assume that you want to convert this to a value of CHAR(6), and an explicit output format of 'HHhMIm'.

```
SELECT T1 FROM INTTIME ;
```

The result (without a type change) is the following report:

```
             T1
---------------
05:57:11.362271
```

Now use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```
SELECT
CAST( (CAST (T1 AS FORMAT 'HHhMim'))
AS CHAR(6))
FROM INTTIME;
```

The result after the nested CASTs is the following report.

```
T1
------
05h57m
```

The inner CAST establishes the display format for the TIME value and the outer CAST indicates the data type of the desired result.

Related Topics

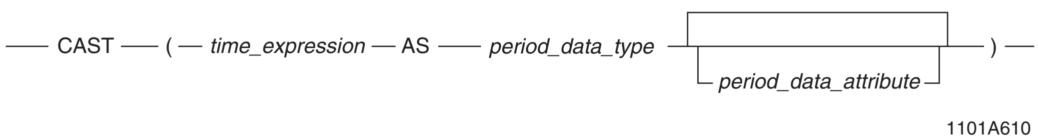
For details on data types and data attributes, see *SQL Data Types and Literals*.

TIME-to-Period Conversion

Purpose

Converts TIME data as PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)][WITH TIME ZONE]).

CAST Syntax



where:

Syntax element …	Specifies …
time_expression	the TIME data expression to be converted.
period_data_type	the target Period type to which *time_expression* is to be converted.
period_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Usage Notes

A TIME(n) [WITH TIME ZONE] value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

If the target precision is higher than the source precision, trailing zeros are added in the result bounds to adjust the precision. If the target precision is lower than the source precision, an error is reported.

If the TIME source value contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.

If the target type is PERIOD(TIME[(n)] [WITH TIME ZONE]), the result beginning element is set to the source value (in UTC). If the target type is PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]), the source time value get adjusted with respect to the current session time zone displacement from the corresponding UTC value; the date portion in the result beginning element is set to CURRENT_DATE, the time portion is set to the source value obtained after the above adjustment, and the resulting timestamp value is converted to UTC. If both the source and target are WITH TIME ZONE, the time zone field of the result beginning element is set to the source time zone field. If only the target has WITH TIME ZONE, the time zone field of the result beginning element is set to the current session time zone displacement. The result ending element is set to the result beginning bound plus one granule of the target type. If the result ending bound has a lower value than the result beginning bound for a target type of PERIOD(TIME[(n)] [WITH TIME ZONE) or the result ending element value exceeds the maximum corresponding TIMESTAMP value for a target type of PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE), an error is reported.

Note: If the target type is WITH TIME ZONE, the result beginning and ending bounds have the same time zones.

Also, note that the result has the same value for the beginning bound and last value.

Example

Assume pt is a TIME(0) column in table t with a value of TIME '12:12:12' and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

In the following example, a TIME(0) column is cast as PERIOD(TIME(4) WITH TIME ZONE). The result beginning bound is formed form the source (in UTC) with trailing zeros added to make the precision 4 and the current session time zone displacement. The result ending element is set to the result beginning bound plus INTERVAL '0.0001' SECOND.

Note: The time zones of the result beginning and ending elements are the same.

```
SELECT CAST(pt AS PERIOD(TIME(4) WITH TIME ZONE)) FROM t;
```

Returns a PERIOD(TIME(4) WITH TIME ZONE) value as follows:

```
('12:12:12.0000-08:00', '12:12:12.0001-08:00')
```

Related Topics

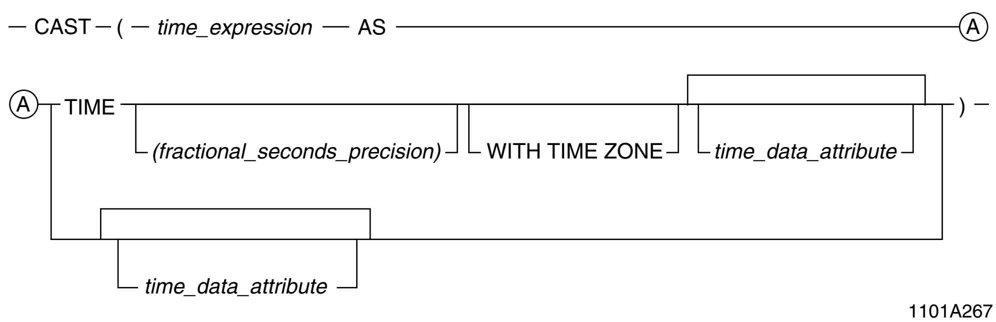
For details on data types and data attributes, see *SQL Data Types and Literals*.

TIME-to-TIME Conversion

Purpose

Converts TIME or TIME WITH TIME ZONE to TIME or TIME WITH TIME ZONE using optional data attributes.

CAST Syntax



where:

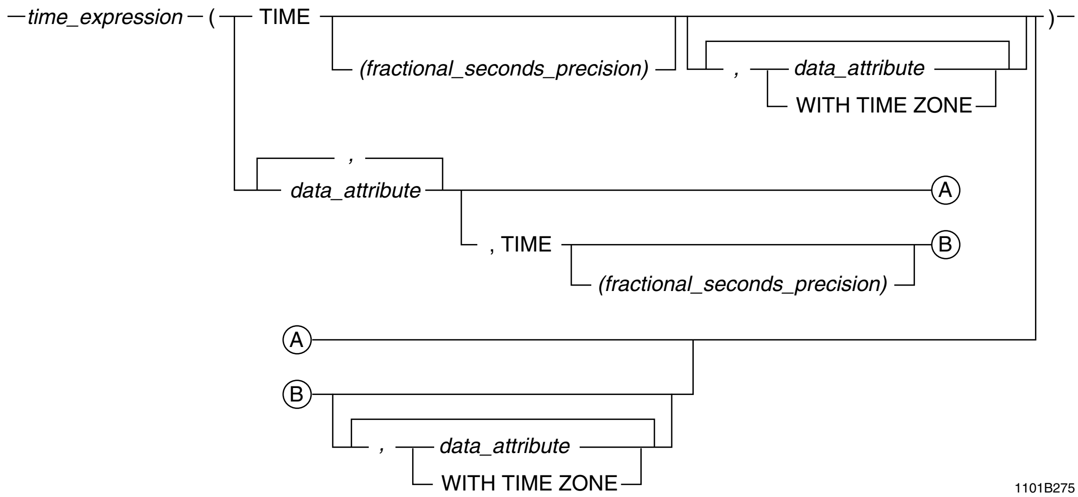
Syntax element …	Specifies …
time_expression	the TIME expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting for DateTime data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

If the target type is TIME WITH TIME ZONE, the explicit (or implicit) Time Zone displacement of the source value is included.

If the target type is TIME WITH TIME ZONE but the source value has the type TIME, then the appropriate WITH TIME ZONE offset is assigned to the target value using the current Time Zone displacement of the session.

Example

The following query returns '12:23:39.999900-08:00', applying the local Time Zone displacement of the session to complete the conversion.

```
SELECT CAST(TIME '12:23:39.9999' AS TIME(6) WITH TIME ZONE);
```

Related Topics

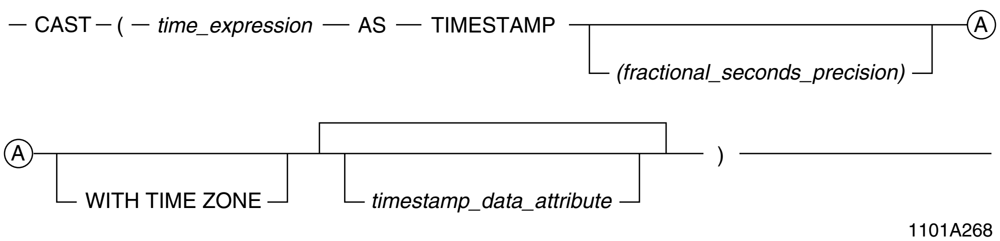
For details on data types and data attributes, see *SQL Data Types and Literals*.

TIME-to-TIMESTAMP Conversion

Purpose

Converts TIME or TIME WITH TIME ZONE to TIMESTAMP or TIMESTAMP WITH TIME ZONE using optional data attributes.

CAST Syntax



where:

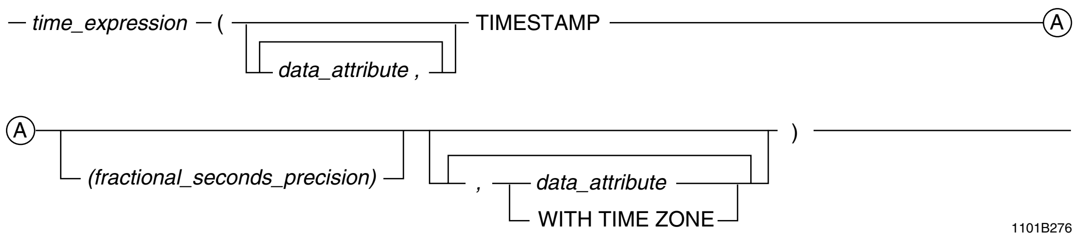
Syntax element ...	Specifies ...
time_expression	the TIME expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting of DateTime data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Implicit TIME-to-TIMESTAMP Conversion

Teradata Database performs implicit conversion from TIME to TIMESTAMP data types in some cases. However, implicit conversion from TIME to TIMESTAMP is not supported for comparisons. See "Implicit Conversion of DateTime types" on page 580.

The following conversions are supported:

From source type...	To target type...
TIME	TIMESTAMP
	TIMESTAMP WITH TIME ZONE
TIME WITH TIME ZONE	TIMESTAMP
	TIMESTAMP WITH TIME ZONE

Conversion from TIME to TIMESTAMP types is performed as follows:

- The YEAR, MONTH, and DAY fields are set to the values of CURRENT_DATE.
- The HOUR, MINUTE, and SECOND fields are set to the values from the source TIME record.
- If the target type is specified WITH TIME ZONE, then the time zone fields of the target are set to the explicit or implicit time zone values from the source TIME type. If the source type is not specified WITH TIME ZONE, the time zone displacement of the current session is used.
- When converting from TIME WITH TIME ZONE to TIMESTAMP, the necessary adjustments for time zone displacement are made, which may change the YEAR, MONTH, DAY, HOUR, and MINUTE fields of the TIMESTAMP value.

Example

If the current date is July 10, 1999, the following query returns '1999-10-07 17:30:48'

```
SELECT CAST(CURRENT_TIME(0) AS TIMESTAMP(0));
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIME-to-UDT Conversion

Purpose

Converts TIME data to UDT data.

CAST Syntax

```
── CAST ── ( time_expression ── AS ── UDT_data_definition ) ──
```

1101A340

where:

Syntax element ...	Specifies ...
time_expression	a TIME expression to be cast to a UDT.
UDT_data_definition	the UDT type, followed by any optional FORMAT, NAMED, or TITLE data attribute phrases, to which *time_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit TIME-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit TIME-to-UDT Conversion

Teradata Database performs implicit TIME-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

If no TIME-to-UDT implicit cast definition exists, Teradata Database looks for a CHAR-to-UDT or VARCHAR-to-UDT implicit cast definition that can substitute for the TIME-to-UDT implicit cast definition. Substitutions are valid because Teradata Database can implicitly cast a TIME type to the character data type, and then use the implicit cast definition to cast from the character data type to the UDT. If multiple character-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.

Related Topics

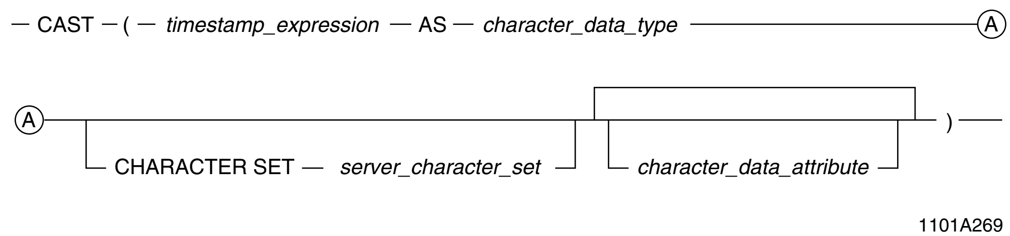
For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-Character Conversion

Purpose

Convert TIMESTAMP data to a character string.

CAST Syntax



where:

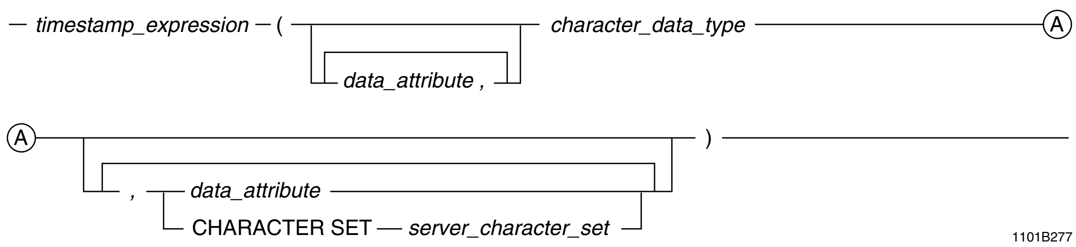
Syntax element …	Specifies …
timestamp_expression	the TIMESTAMP expression to be cast to a character type.
character_data_type	the character type to which the TIMESTAMP expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be cast to a character type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character type to which the TIMESTAMP expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

When converting TIMESTAMP to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the TIMESTAMP value as represented by a character string literal.

IF the target data type is ...	AND *n* is ...	THEN ...
CHAR(*n*)	greater than the length of the TIMESTAMP value as represented by a character string literal	trailing pad characters are added to pad the representation.
	too small	a string truncation error is returned.

IF the target data type is …	AND *n* is …	THEN …
VARCHAR(*n*)	greater than the length of the TIMESTAMP value as represented by a character string literal	no blank padding is added to the character representation.
	too small	a string truncation error is returned.

TIMESTAMP to CLOB conversion is not supported.

You cannot convert a TIME value to a character string if the server character set is GRAPHIC.

Forcing a FORMAT on CAST for Converting TIMESTAMP to Character

The default format for TIMESTAMP to character conversion is the format in effect for the TIMESTAMP value.

To override the format, you can convert a TIMESTAMP value to a string using a FORMAT phrase. The resulting format, however, is the same as the TIMESTAMP value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from TIMESTAMP to CHAR and force an explicit FORMAT on the result regardless of the format associated with the TIMESTAMP value. This is because of the rules for matching FORMAT phrases to data types.

Example

Field TS1 in the table INTTIMESTAMP is a TIMESTAMP value with the explicit format 'Y4-MM-DDBHH:MI:SSDS(6)'. Assume that you want to convert this to a value of CHAR(19), and an explicit output format of 'M3BDD,BY4BHHhMIm'.

```
SELECT TS1 FROM INTTIMESTAMP;
```

The result (without a type change) is the following report:

```
                       TS1
--------------------------
1900-12-31 08:25:37.899231
```

Now use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```
SELECT
CAST( (CAST (TS1 AS FORMAT 'M3BDD,BY4BHHhMIm'))
AS CHAR(19))
FROM INTTIMESTAMP;
```

The result after the nested CASTs is the following report.

```
TS1
-------------------
Dec 31, 1900 08h25m
```

The inner CAST establishes the display format for the TIMESTAMP value and the outer CAST indicates the data type of the desired result.

Related Topics

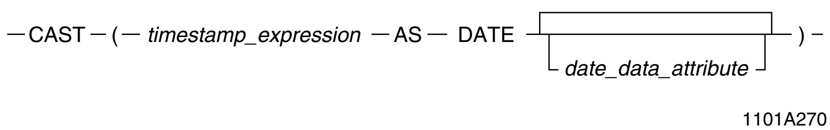
For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-DATE Conversion

Purpose

Convert TIMESTAMP data to a DATE value.

CAST Syntax



where:

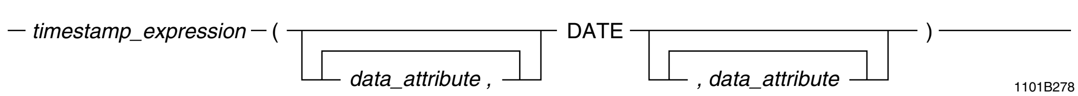
Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.
date_data_attribute	any of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of DATE data attribute phrases, such as FORMAT that enables an alternative format.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.

Syntax element ...	Specifies ...
data_attribute	any of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Implicit TIMESTAMP-to-DATE Conversion

Teradata Database performs implicit conversion from TIMESTAMP types to DATE in some cases. See "Implicit Conversion of DateTime types" on page 580.

The following conversions are supported:

From source type...	To target type...
TIMESTAMP	DATE[a]
TIMESTAMP WITH TIME ZONE	

a. ANSIDate dateform mode or IntegerDate dateform mode

The YEAR, MONTH, and DAY fields are set to the values from the source TIMESTAMP record after any necessary adjustments for time zone displacement are made. The time zone adjustment may change the YEAR, MONTH, and DAY fields of the DATE value.

The TIMESTAMP value is always converted to DATE in case of comparison.

Example

A single column table has three rows of type TIMESTAMP(0) WITH TIME ZONE.

A query that requests the field values and CASTs them as DATE is performed during a session that has its Local Time Zone defined as -'08:00'.

The results table is as follows.

```
TimeStampWithTimeZone     CastAsDate
--------------------------------------------------
1997-10-07 15:43:00+08:00 1997-10-06
1997-10-07 15:47:52-08:00 1997-10-07
1997-10-07 15:43:00-00:00 1997-10-07
```

Notice that the difference between the stored Time Zone and the Local Time Zone is 16 hours in the first row, but at the same time the TimeStamp value is 15:43, which is less than 16.

This puzzling result can be clarified using a similar query that casts TIMESTAMP(0) WITH TIME ZONE as TIMESTAMP(0), omitting the Time Zone information.

The results table for this query is as follows.

```
TimeStampWithTimeZone     CastAsTimeStamp
-------------------------------------------------
1997-10-07 15:43:00+08:00 1997-10-06 23:43:00
1997-10-07 15:47:52-08:00 1997-10-07 15:47:52
1997-10-07 15:43:00-00:00 1997-10-07 07:43:00
```

After the CAST, the values are all displayed at Local Time Zone, and the value in the first row indicates that the 16 hour adjustment rolled the date back 1, to a time near the end of that date.

Related Topics

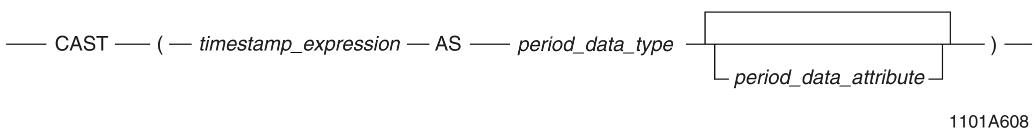
For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-Period Conversion

Purpose

Converts a TIMESTAMP value as PERIOD(DATE), PERIOD(TIME[(n)][WITH TIME ZONE]), or PERIOD(TIMESTAMP[(n)][WITH TIME ZONE]).

CAST Syntax



where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP data expression to be converted.
period_data_type	the target Period type to which *timestamp_expression* is to be converted.
period_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Usage Notes

A TIMESTAMP(n) [WITH TIME ZONE] value can be cast as PERIOD(DATE), PERIOD(TIME[(n)] [WITH TIME ZONE]), or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

If the target type is PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]):

- If the target precision is higher than the source precision, trailing zeros are added in the result bounds to adjust the precision.
- If the target precision is lower than the source precision, an error is reported.

If the target type is PERIOD(DATE), the result beginning bound is the date portion of the source beginning bound adjusted to the current session time zone.

If the target type is PERIOD(TIME[(n)]), the result beginning bound is the time portion of the source value (in UTC).

If the target type is PERIOD(TIME[(n)] WITH TIME ZONE), the result beginning bound is formed from the time portion of the source value (in UTC) and, if the source type is WITH TIME ZONE, the source time zone displacement and, if not, the current session time zone displacement.

If the target type is PERIOD(TIMESTAMP[(n)]), the result beginning bound is the timestamp portion of the source value (in UTC).

If the target type is PERIOD(TIMESTAMP[(n)] WITH TIME ZONE), the result beginning bound is formed from the timestamp portion of the source value (in UTC) and, if the source type is WITH TIME ZONE, the source time zone displacement and, if not, the current session time zone displacement.

If the TIMESTAMP source value contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.

The result ending element is set to the result beginning bound plus one granule of the target type. If the result ending bound exceeds the maximum allowed DATE or TIMESTAMP value for a target type of PERIOD(DATE) or PERIOD(TIMESTAMP[(n)]), respectively, or the ending bound has a lower value than the result beginning bound in their UTC forms for a target type of PERIOD(TIME[(n)]), an error is reported.

Note: If the target type is WITH TIME ZONE, the result beginning and ending bounds have the same time zones.

Also, note that the result has the same value for the beginning bound and last value.

Example

In the following example, a TIMESTAMP(6) literal is cast as PERIOD(DATE). The result beginning element is set to the date portion of the source value. The result ending element is set to result beginning bound plus INTERVAL '1' DAY.

```
SELECT CAST(TIMESTAMP '2005-02-03 12:12:12.340000' AS PERIOD(DATE));
```

The following is returned:

```
('2005-02-03', '2005-02-04')
```

Related Topics

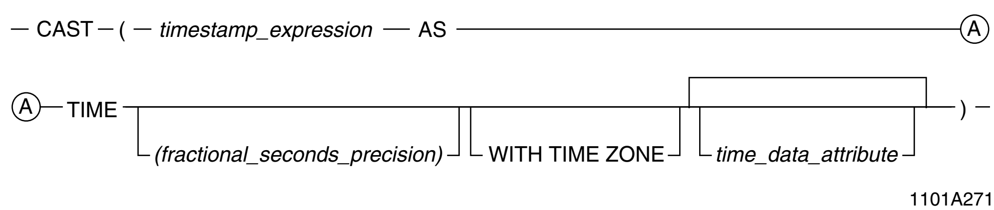
For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-TIME Conversion

Purpose

Convert TIMESTAMP data to a TIME value.

CAST Syntax



where:

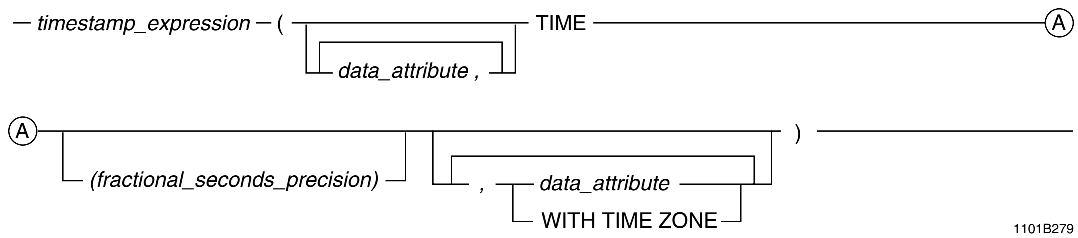
Syntax element …	Specifies …
timestamp_expression	the TIMESTAMP expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of TIME data attribute phrases.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Implicit TIMESTAMP-to-TIME Conversion

Teradata Database performs implicit conversion from TIMESTAMP to TIME data types in some cases. However, implicit conversion from TIMESTAMP to TIME is not supported for comparisons. See "Implicit Conversion of DateTime types" on page 580.

The following conversions are supported:

From source type...	To target type...
TIMESTAMP	TIME
	TIME WITH TIME ZONE
TIMESTAMP WITH TIME ZONE	TIME
	TIME WITH TIME ZONE

Conversion from TIMESTAMP to TIME types is performed as follows:

- The HOUR, MINUTE, and SECOND fields are set to the values from the source TIMESTAMP record.
- If the target type is specified WITH TIME ZONE, then the time zone fields of the target are set to the explicit or implicit time zone values from the source TIMESTAMP type. If the source type is not specified WITH TIME ZONE, the time zone displacement of the current session is used.
- When converting from TIMESTAMP WITH TIME ZONE to TIME, the necessary adjustments for time zone displacement are made, which may change the HOUR and MINUTE fields of the TIME value.

Example

The following query returns '23:59:59.00000-08:00'.

```
   SELECT CAST(TIMESTAMP '1997-12-31 23:59:59' AS TIME(5) WITH TIME
ZONE);
```

Related Topics

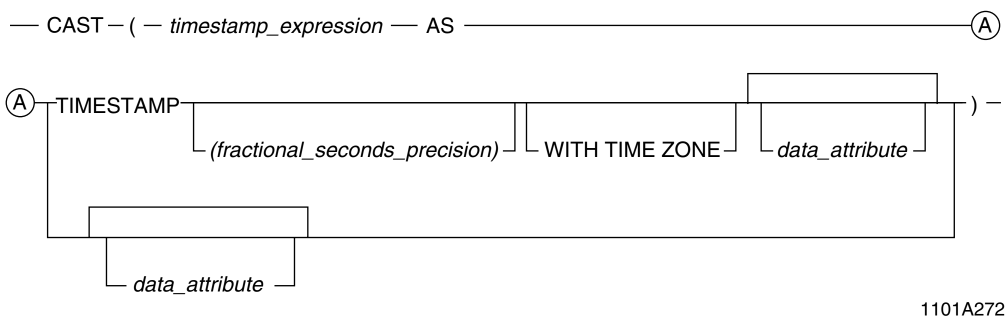
For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-TIMESTAMP Conversion

Purpose

Convert TIMESTAMP data to a TIMESTAMP value with different precision information or WITH TIME ZONE definition.

CAST Syntax



where:

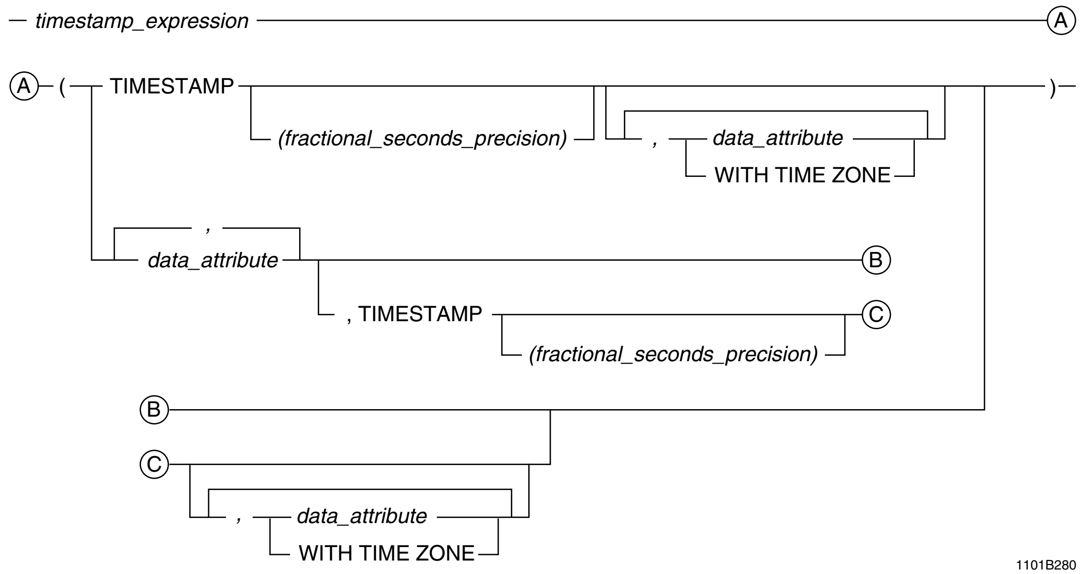
Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting for the character representations of DateTime and Interval data.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

If both source and target data types are the same with respect to having a WITH TIME ZONE definition, then no conversion of internal data is performed; however, precision information *might* change in such a case.

Because internal values for TIMESTAMPs carry full precision, the data need not be changed by a CAST. The number of decimal digits displayed when the value is output, however, *can* be affected by the conversion.

When the source and target differ with respect to their WITH TIME ZONE definitions, conversion is required. Notice that CASTing TIMESTAMP WITH TIME ZONE to TIMESTAMP can change the values of fields in the result.

For an example of how field values can change unexpectedly as a result of CASTing WITH TIME ZONE values, see "TIMESTAMP-to-DATE Conversion" on page 705.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-UDT Conversion

Purpose

Converts TIMESTAMP data to UDT data.

CAST Syntax

```
── CAST ── ( timestamp_expression ── AS ── UDT_data_definition ) ──
```

1101A341

where:

Syntax element …	Specifies …
timestamp_expression	a TIMESTAMP expression to be cast to a UDT.
UDT_data_definition	the UDT type, followed by any optional FORMAT, NAMED, or TITLE data attribute phrases, to which *timestamp_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit TIMESTAMP-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit TIMESTAMP-to-UDT Conversion

Teradata Database performs implicit TIMESTAMP-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

If no TIMESTAMP-to-UDT implicit cast definition exists, Teradata Database looks for a CHAR-to-UDT or VARCHAR-to-UDT implicit cast definition that can substitute. Substitutions are valid because Teradata Database can implicitly cast a TIMESTAMP type to the character data type, and then use the implicit cast definition to cast from the character data type to the UDT. If multiple character-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.

Related Topics

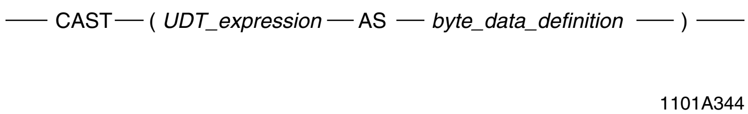
For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-Byte Conversion

Purpose

Converts a UDT expression to a byte data type.

CAST Syntax



where:

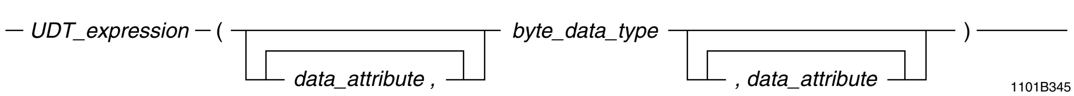
Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
byte_data_definition	the BLOB, BYTE or VARBYTE byte type followed by optional FORMAT, NAMED, or TITLE attribute phrases to which *UDT_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
byte_data_type	the BLOB, BYTE or VARBYTE byte type to which UDT_expression is to be converted.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-byte conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit UDT-to-byte data type conversion requires a cast definition (see "Usage Notes") that specifies the following:

- the AS ASSIGNMENT clause
- a BYTE, VARBYTE, or BLOB target data type

  The target data type of the cast definition does not have to be an exact match to the target of the implicit type conversion.

If multiple implicit cast definitions exist for converting the UDT to different byte types, Teradata Database uses the implicit cast definition for the byte type with the highest precedence. The following list shows the precedence of byte types in order from lowest to highest precedence:

- BYTE
- VARBYTE
- BLOB

Example

Consider the following table definition, where image is a UDT:

```
CREATE TABLE history
   (id INTEGER
   ,information image );
```

Assuming an appropriate cast definition exists for the image UDT, the following statement converts the values in the information column to BYTE:

```
SELECT CAST (information AS BYTE(20))
FROM history
WHERE id = 100121;
```

Related Topics

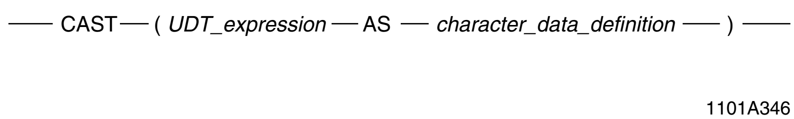
For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-Character Conversion

Purpose

Converts a UDT expression to a character data type.

CAST Syntax



where:

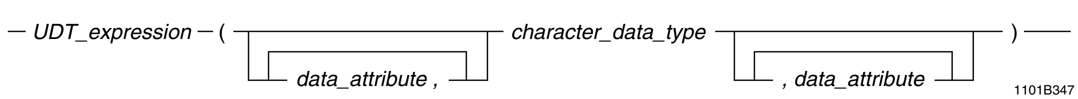
Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
character_data_definition	the target character type, for example CHAR or VARCHAR, followed by optional FORMAT, NAMED, or TITLE attribute phrases.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the target character type, for example CHAR or VARCHAR.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-character conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

The target character type of the cast definition does not have to be an exact match to the target character type of the implicit conversion. Teradata Database can use an implicit cast definition that specifies a CHAR, VARCHAR, or CLOB target type.

If multiple implicit cast definitions exist for converting the UDT to different character types, Teradata Database uses the implicit cast definition for the character type with the highest precedence. The following list shows the precedence of character types in order from lowest to highest precedence:

- CHAR
- VARCHAR
- CLOB

If no UDT-to-character implicit cast definitions exist, Teradata Database looks for other cast definitions that can substitute for the UDT-to-character implicit cast definition:

IF the following combination of implicit cast definitions exists …				THEN Teradata Database …
UDT-to-numeric	UDT-to-DATE	UDT-to-TIME	UDT-to-TIMESTAMP	
X				uses the UDT-to-numeric implicit cast definition. If multiple UDT-to-numeric implicit cast definitions exist, then Teradata Database returns an SQL error.
	X			uses the UDT-to-DATE implicit cast definition.
		X		uses the UDT-to-TIME implicit cast definition.
			X	uses the UDT-to-TIMESTAMP implicit cast definition.
X	X			reports an error.
X		X		
X			X	
	X	X		
	X		X	
		X	X	
X	X	X		
X	X		X	
X		X	X	
	X	X	X	
X	X	X	X	

Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a character type.

Example

Consider the following table definition, where euro is a UDT:

```
CREATE TABLE euro_sales_table
   (quarter INTEGER
   ,region VARCHAR(20)
   ,sales euro );
```

Assuming an appropriate cast definition exists for the euro UDT, the following statement converts the values in the sales column to CHAR(10):

```
SELECT region, CAST (sales AS CHAR(10))
FROM euro_sales_table
WHERE quarter = 1;
```

Related Topics

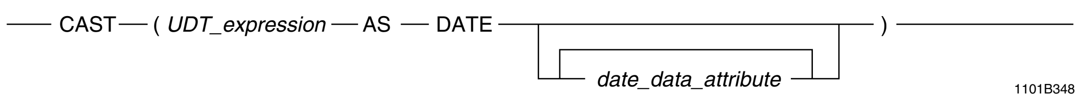
For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-DATE Conversion

Purpose

Converts a UDT expression to a DATE data type.

CAST Syntax



where:

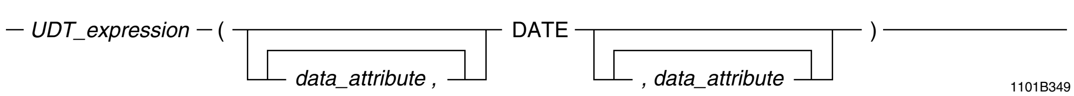
Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
date_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

Teradata Database performs implicit UDT-to-DATE conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

If no UDT-to-DATE implicit cast definition exists, Teradata Database looks for other cast definitions that can substitute for the UDT-to-DATE implicit cast definition:

IF the following combination of implicit cast definitions exists ...		THEN Teradata Database ...
UDT-to-Numeric	UDT-to-Character (non-CLOB)	
X		uses the UDT-to-numeric implicit cast definition. If multiple UDT-to-numeric implicit cast definitions exist, then Teradata Database returns an SQL error.
	X	uses the UDT-to-character implicit cast definition. If multiple UDT-to-character implicit cast definitions exist, then Teradata Database returns an SQL error.
X	X	reports an error.

Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a DATE type.

Example

Consider the following table definition, where datetime_record is a UDT:

```
CREATE TABLE support
   (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime_record UDT, the following statement converts the values in the information column to DATE:

```
SELECT id, CAST (information AS DATE) FROM support;
```

Related Topics

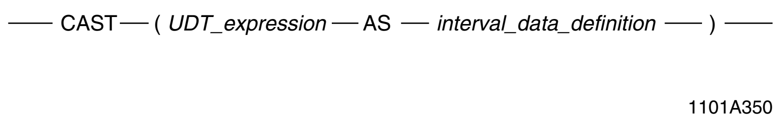
For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-INTERVAL Conversion

Purpose

Converts a UDT expression to an INTERVAL data type.

CAST Syntax



where:

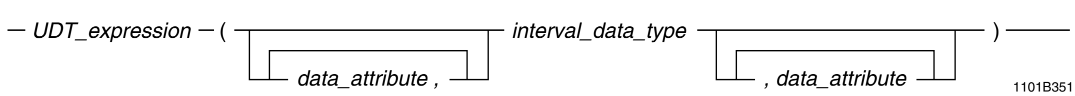
Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
interval_data_definition	the target predefined interval type followed by optional NAMED or TITLE attribute phrases.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • NAMED • TITLE
interval_data_type	the target predefined interval type to which *UDT_expression* is to be converted.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Performing an implicit data type conversion requires a cast definition (see "Usage Notes") that specifies the following:

- the AS ASSIGNMENT clause
- a target data type that is in the same INTERVAL family as the target of the implicit cast:

This INTERVAL data type ...	Belongs to this INTERVAL family ...
• INTERVAL YEAR • INTERVAL YEAR TO MONTH • INTERVAL MONTH	Year-Month
• INTERVAL DAY • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE • INTERVAL MINUTE TO SECOND • INTERVAL SECOND	Day-Time

The target data type of the cast definition does not have to be an exact match to the target of the implicit type conversion.

Teradata Database performs implicit UDT-to-INTERVAL conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Example

Consider the following table definition, where datetime_record is a UDT:

```
CREATE TABLE support
   (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime_record UDT, the following statement converts the values in the information column to INTERVAL MONTH:

```
SELECT id, CAST (information AS INTERVAL MONTH) FROM support;
```

Related Topics

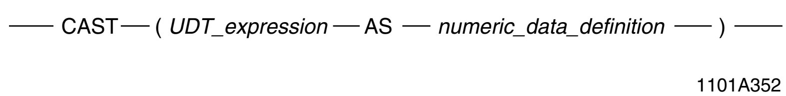
For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-Numeric Conversion

Purpose

Converts a UDT expression to a numeric data type.

CAST Syntax



where:

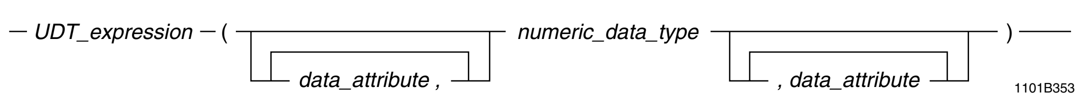
Syntax element …	Specifies …
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see “Scalar UDF Expression” on page 552.
numeric_data_definition	the target predefined numeric type followed by any optional FORMAT, NAMED, or TITLE attribute phrases.

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax



where:

Syntax element …	Specifies …
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see “Scalar UDF Expression” on page 552.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
numeric_data_type	a predefined numeric type to which *UDT_expression* is to be converted.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-numeric conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

The target numeric type of the cast definition does not have to be an exact match to the target numeric type of the implicit conversion. Teradata Database can use an implicit cast definition that specifies a BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, or REAL/FLOAT/DOUBLE target type.

If multiple implicit cast definitions exist for converting the UDT to different numeric types, Teradata Database uses the implicit cast definition for the numeric type with the highest precedence. The following list shows the precedence of numeric types in order from lowest to highest precedence:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT

- DECIMAL/NUMERIC
- REAL/FLOAT/DOUBLE

If no UDT-to-numeric implicit cast definitions exist, Teradata Database looks for other cast definitions that can substitute for the UDT-to-character implicit cast definition:

IF the following combination of implicit cast definitions exists ...		THEN Teradata Database ...
UDT-to-DATE	UDT-to-Character[a]	
X		uses the UDT-to-DATE implicit cast definition.
	X	uses the UDT-to-character implicit cast definition. If multiple UDT-to-character implicit cast definitions exist, then Teradata Database returns an SQL error.
X	X	reports an error.

a. a non-CLOB character type

Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a numeric type.

Example

Consider the following table definition, where euro is a UDT:

```
CREATE TABLE euro_sales_table
   (quarter INTEGER
   ,region VARCHAR(20)
   ,sales euro );
```

Assuming an appropriate cast definition exists for the euro UDT, the following statement converts the values in the sales column to DECIMAL(10,2):

```
SELECT SUM (CAST (sales AS DECIMAL(10,2))) FROM euro_sales_table;
```

Related Topics

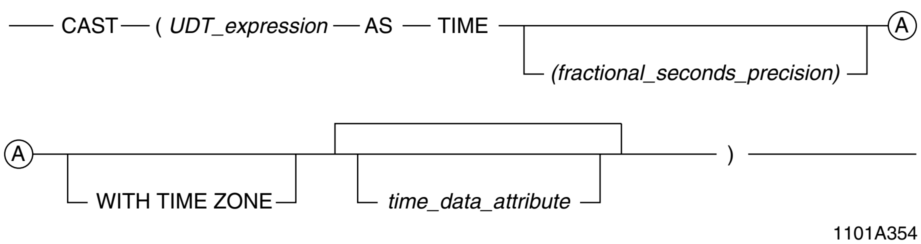
For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-TIME Conversion

Purpose

Converts a UDT expression to a TIME data type.

CAST Syntax



where:

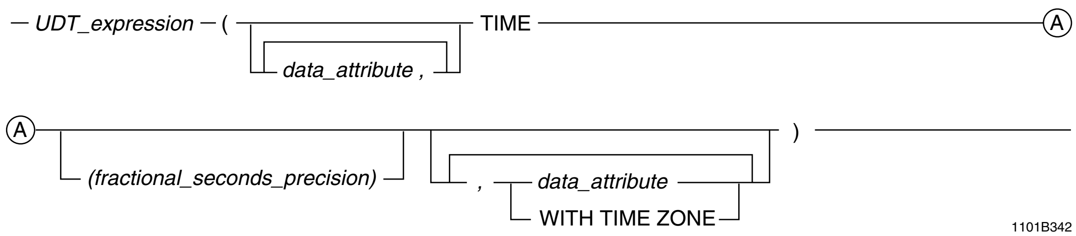
Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-TIME conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

If no UDT-to-TIME implicit cast definition exists, Teradata Database looks for a UDT-to-CHAR or UDT-to-VARCHAR cast definition that can substitute for the UDT-to-TIME implicit cast definition. Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to a character data type, and then implicitly cast the character data type to a DATE type. If multiple UDT-to-character implicit cast definitions exist, then Teradata Database returns an SQL error.

Example

Consider the following table definition, where datetime_record is a UDT:

```
CREATE TABLE support
   (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime_record UDT, the following statement converts the values in the information column to TIME WITH TIME ZONE:

```
SELECT id, CAST (information AS TIME WITH TIME ZONE) FROM support;
```

Related Topics

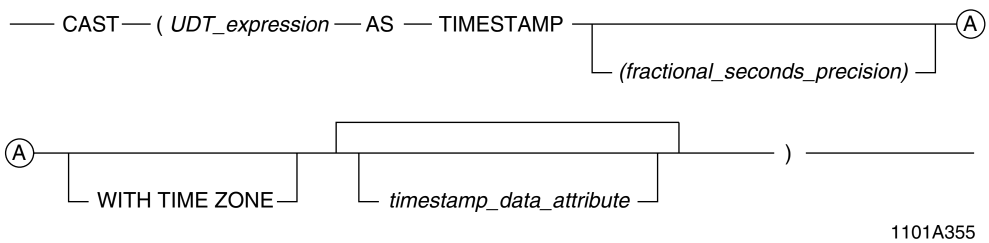
For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-TIMESTAMP Conversion

Purpose

Converts a UDT expression to a TIMESTAMP data type.

CAST Syntax



where:

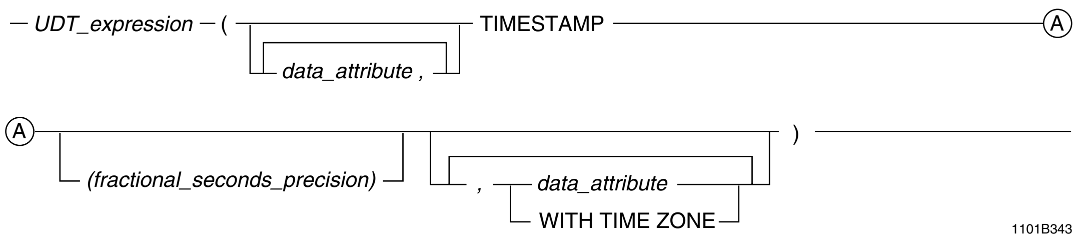
Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
timestamp_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax



where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL:2008 standard.

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-TIMESTAMP conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs

- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

If no UDT-to-TIMESTAMP implicit cast definition exists, Teradata Database looks for a UDT-to-CHAR or UDT-to-VARCHAR cast definition that can substitute for the UDT-to-TIMESTAMP implicit cast definition. Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to a character data type, and then implicitly cast the character data type to a TIMESTAMP type. If multiple UDT-to-character implicit cast definitions exist, then Teradata Database returns an SQL error.

Example

Consider the following table definition, where datetime_record is a UDT:

```
CREATE TABLE support
   (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime_record UDT, the following statement converts the values in the information column to TIMESTAMP:

```
SELECT id, CAST (information AS TIMESTAMP) FROM support;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-UDT Conversion

Purpose

Converts a UDT expression to a different UDT type.

CAST Syntax

```
—— CAST — ( UDT_expression — AS — UDT_data_definition — ) ——
```

1101A356

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "Scalar UDF Expression" on page 552.
UDT_data_definition	a UDT type to which *UDT_expression* is to be converted, followed by any of the following optional attribute phrases: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2008 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit UDT-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-UDT casts for the following operations:

- UPDATE
- INSERT

- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

An implicit data type conversion involving a UDT can only be performed if the cast definition specifies the AS ASSIGNMENT clause. For more information, see "CREATE CAST" in *SQL Data Definition Language*.

Example

Consider the following table definitions, where euro and us_dollar are UDTs:

```
CREATE TABLE euro_sales_table
   (euro_quarter INTEGER
   ,euro_region VARCHAR(20)
   ,euro_sales euro );

CREATE TABLE us_sales_table
   (us_quarter INTEGER
   ,us_region VARCHAR(20)
   ,us_sales us_dollar );
```

Assuming an appropriate cast definition exists for converting the euro UDT to a us_dollar UDT, the following statement performs a us_dollar UDT to euro UDT conversion:

```
INSERT INTO euro_sales_table
   SELECT us_quarter, us_region, CAST (us_sales AS euro)
   FROM us_sales_table;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

APPENDIX A Notation Conventions

This appendix describes the notation conventions used in this book.

This book uses three conventions to describe the SQL syntax and code:

<table>
<tr><th>Convention</th><th>Description</th></tr>
<tr><td>Syntax diagrams</td><td>Describes SQL syntax form, including options.
For details, see “Syntax Diagram Conventions” on page 743.</td></tr>
<tr><td>Square braces in the text</td><td>Represent options. The indicated parentheses are required when you specify options.
For example:
DECIMAL [(n[,m])] means the decimal data type can be defined optionally:
without specifying the precision value n or scale value m
specifying precision (n) only
specifying both values (n,m)
You cannot specify scale without first defining precision.
CHARACTER [(n)] means that use of (n) is optional.
The values for n and m are integers in all cases.</td></tr>
<tr><td>Japanese character code shorthand notation</td><td>Represent unprintable Japanese characters.
For details, see “Character Shorthand Notation Used In This Book” on page 748.</td></tr>
</table>

Symbols from the predicate calculus are also used occasionally to describe logical operations. For details, see “Predicate Calculus Notation Used In This Book” on page 750.

Syntax Diagram Conventions

Notation Conventions

Item	Definition / Comments
Letter	An uppercase or lowercase alphabetic character ranging from A through Z.
Number	A digit ranging from 0 through 9. Do not use commas when typing a number with more than 3 digits.

Item	Definition / Comments
Word	Keywords and variables. • UPPERCASE LETTERS represent a keyword. Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase. • lowercase letters represent a keyword that you must type in lowercase, such as a UNIX command. • *lowercase italic letters* represent a variable such as a column or table name. Substitute the variable with a proper value. • **lowercase bold letters** represent an excerpt from the diagram. The excerpt is defined immediately following the diagram that contains it. • UNDERLINED LETTERS represent the default value. This applies to both uppercase and lowercase words.
Spaces	Use one space between items such as keywords or variables.
Punctuation	Type all punctuation exactly as it appears in the diagram.

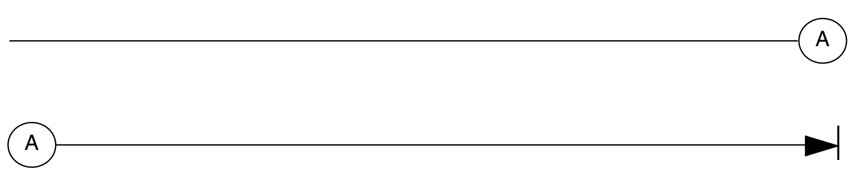
Paths

The main path along the syntax diagram begins at the left with a keyword, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

Continuation Links

Paths that are too long for one line use continuation links. Continuation links are circled letters indicating the beginning and end of a link:



FE0CA002

When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue reading.

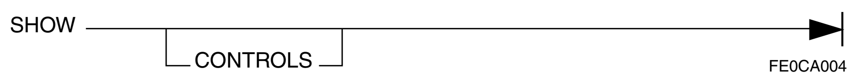
Required Entries

Required entries appear on the main path:

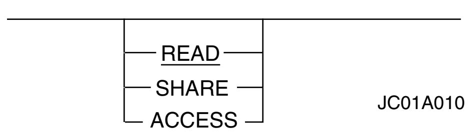
If you can choose from more than one entry, the choices appear vertically, in a stack. The first entry appears on the main path:

Optional Entries

You may choose to include or disregard optional entries. Optional entries appear below the main path:



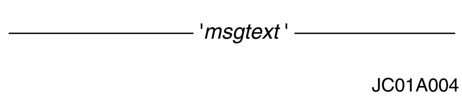
If you can optionally choose from more than one entry, all the choices appear below the main path:



Some commands and statements treat one of the optional choices as a default value. This value is UNDERLINED. It is presumed to be selected if you type the command or statement without specifying one of the options.

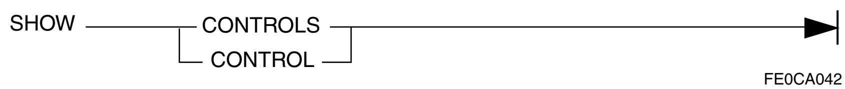
Strings

String literals appear in single quotes:



Abbreviations

If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.

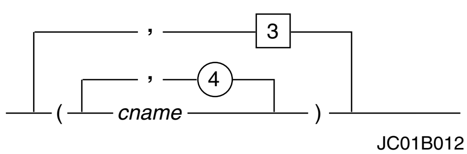


In the above syntax, the following formats are valid:

- SHOW CONTROLS
- SHOW CONTROL

Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat:



Read loops from right to left.

The following conventions apply to loops:

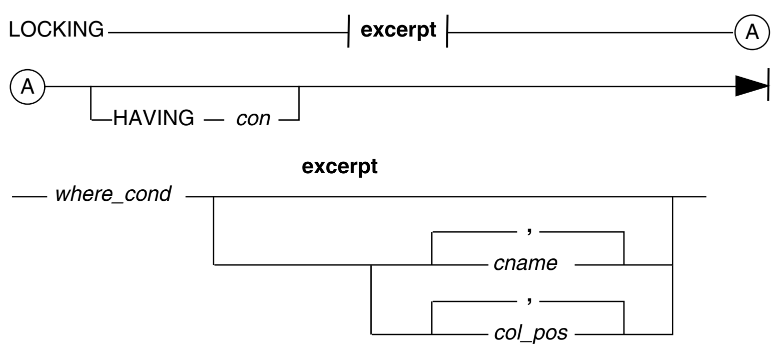
IF...	THEN...
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may type *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must type at least three groups of column names.
a separator character is required between entries	the character appears on the return path. If the diagram does not show a separator character, use one blank space. In the example, the separator character is a comma.

IF...	THEN...
a delimiter character is required around entries	the beginning and end characters appear outside the return path. Generally, a space is not needed between delimiter characters and entries. In the example, the delimiter characters are the left and right parentheses.

Excerpts

Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by (|) terminators on each side of the break. The name for the excerpted piece appears between the terminators in boldface type.

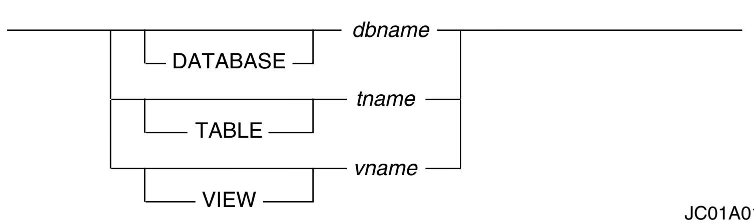
The boldface excerpt name and the excerpted phrase appears immediately after the main diagram. The excerpted phrase starts and ends with a plain horizontal line:



JC01A014

Multiple Legitimate Phrases

In a syntax diagram, it is possible for any number of phrases to be legitimate:



JC01A016

In this example, any of the following phrases are legitimate:

- *dbname*
- DATABASE *dbname*
- *tname*

- TABLE *tname*
- *vname*
- VIEW *vname*

Sample Syntax Diagram

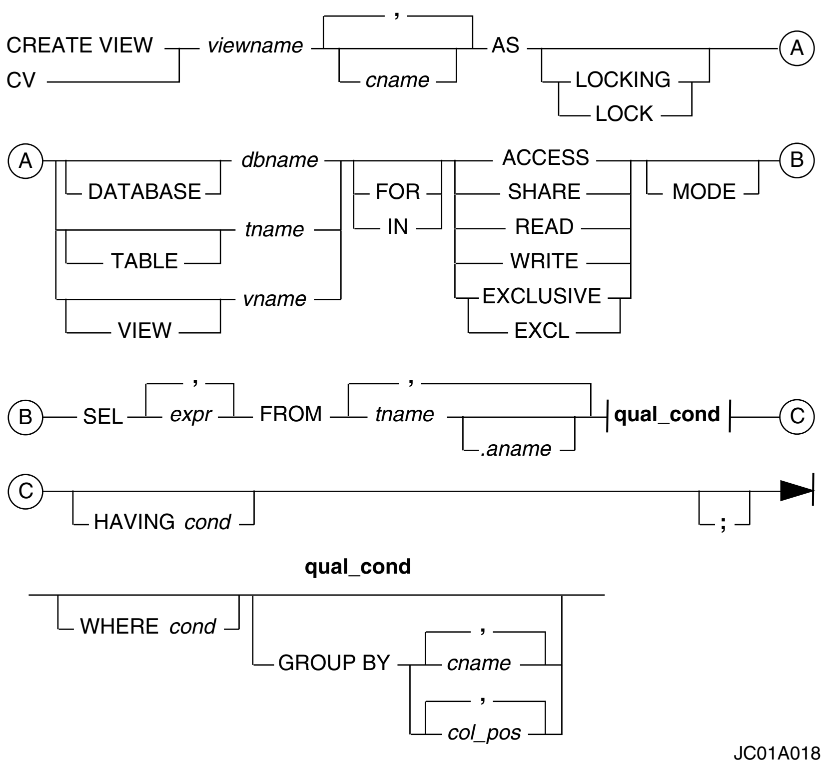


Diagram Identifier

The alphanumeric string that appears in the lower right corner of every diagram is an internal identifier used to catalog the diagram. The text never refers to this string.

Character Shorthand Notation Used In This Book

Introduction

This book uses the Unicode naming convention for characters. For example, the lowercase character ‘a’ is more formally specified as either LATIN SMALL 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 book, 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 *International Character Set Support*.

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.
$\underline{a}$–$\underline{z}$ $\underline{A}$–$\underline{Z}$ $\underline{0}$–$\underline{9}$	Unicode compatibility zone	Any fullwidth Latin letter or digit.
<	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 sometimes preceded by "ss_3".
$\underline{\mathbf{I}}$	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by "ss_2", 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 server character sets.

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

Predicate Calculus Notation Used In This Book

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this book uses the following predicate calculus notation to explain concepts.

This symbol ...	Represents this phrase ...
iff	If and only if
$\forall$	For all
$\exists$	There exists

Glossary

AMP Access Module Process

ANSI American National Standards Institute

BLOB Binary Large Object

BTEQ Basic Teradata Query

BYNET Banyan Network

CJK Chinese, Japanese, and Korean

CLIv2 Call Level Interface Version 2

CLOB Character Large Object

cs0, cs1, cs2, cs3 Four code sets (codeset 0, 1, 2, and 3) used in EUC encoding.

distinct type A UDT that is based on a single predefined data type

E2I External-to-Internal

EUC Extended UNIX Code

FK Foreign Key

HI Hash Index

I2E Internal-to-External

JI Join Index

JIS Japanese Industrial Standards

LOB Large Object

LT/ST Large Table/Small Table (join)

NPPI Nonpartitioned Primary Index

NUPI Nonunique Primary Index

NUSI Nonunique Secondary Index

OLAP OnLine Analytical Processing

OLTP OnLine Transaction Processing

PDE Parallel Database Extensions

PE Parsing Engine vproc

PI Primary Index

PK Primary Key

PPI Partitioned Primary Index

predefined type Teradata Database system type such as INTEGER and VARCHAR

RDBMS Relational Database Management System

SDF Specification for Data Formatting

structured type A UDT that is a collection of one or more fields called attributes, each of which is defined as a predefined data type or other UDT (which allows nesting)

UCS Universal Coded Character Set, specified by International Standard ISO/IEC 10646

UDF User-Defined Function

UDM User-Defined Method

UDT User-Defined Type

UDT expression An expression that returns a distinct or structured UDT data type

UPI Unique Primary Index

USI Unique Secondary Index

vproc Virtual Process

Index

Symbols

||, concatenation operator 373

A

ABS function 54
ACCOUNT function 528
Account string, get account string 528
ACOS inverse trigonometric function 91
ACOSH hyperbolic function 97
ADD_MONTHS function 174
Addition operator 46
Aggregate functions
  AVG 220
  constant expressions and 216
  CORR 223
  COUNT 226
  COVAR_POP 231
  COVAR_SAMP 234
  date and 216
  DateTime types and 170
  DISTINCT option and 219
  floating point data and 218
  GROUP BY clause and 215
  GROUPING 237
  HAVING clause and 218
  interval types and 170
  KURTOSIS 240
  LOB data types and 218
  MAX 242
  MIN 245
  nesting 217
  nulls and 217
  Period data types and 218
  recursive queries and 218
  REGR_AVGX 248
  REGR_AVGY 251
  REGR_COUNT 254
  REGR_INTERCEPT 258
  REGR_R2 262
  REGR_SLOPE 266
  REGR_SXX 270
  REGR_SXY 273
  REGR_SYY 276
  select list containing 215
  SKEW 279
  STDDEV_POP 282
  STDDEV_SAMP 285
  SUM 288
  VAR_POP 291
  VAR_SAMP 294
  when expression evaluates to zero 217
  WHERE clause and 218
ALL predicate quantifier 445
AMP
  identify maximum number in configuration 516
  identify with HASHAMP 516
AND logical operator 490
  truth table 492
ANY predicate quantifier 445
Arithmetic functions
  ABS function 54
  DEGREES function 94
  EXP function 62
  LN function 64
  LOG function 66
  RADIANS function 94
  RANDOM function 71
  SQRT function 82
Arithmetic operators 212
  - 46
  * 46
  ** 46
  + 46
  / 46
  addition operator 46
  division operator 46
  exponentiate 46
  LOB data types and 46
  MOD operator 46
  multiplication 46
  Period data types and 46
  subtraction operator 46
  unary minus operator 46
  unary plus operator 46
ASIN inverse trigonometric function 91
ASINH hyperbolic function 97
ATAN inverse trigonometric function 91
ATAN2 inverse trigonometric function 91
ATANH hyperbolic function 97
Attribute functions 495
  BYTES 496
  CHARACTER_LENGTH 498
  CHARACTERS 501

DEFAULT 503
FORMAT 507
MCHARACTERS 495, 498
OCTET_LENGTH 508
TITLE 511
TYPE 512
AVERAGE aggregate function. See AVG aggregate function.
AVG aggregate function
DateTime types and 170
described 220
Interval types and 170
AVG window function, described 320

B

BEGIN function 186
BETWEEN predicate 450
Blank, as used in strings 473
BLOB data types
aggregate functions and 218
arithmetic operators and 46
comparison operators and 101
predicates and 442
Bound functions
BEGIN function 186
End function 188, 190
Built-in functions 527
ACCOUNT 528
CURRENT_DATE 529
CURRENT_TIME 533
CURRENT_TIMESTAMP 535
CURRENT_USER 537
DATABASE function 538
DATE function 539
PROFILE 541
ROLE 531, 542
SESSION 545
TIME 546
USER 548
Byte conversion 588
HASHBUCKET function and 523
Byte functions, TRIM 420
BYTES function 496

C

CALENDAR system view
cumulative sum, and calculating 339
moving difference, and calculating 345
CASE expression and nulls 40
CASE operation
COALESCE expression 40
data type of, rules governing 32
defined 23
NULLIF expression 42
searched 27
valued 24
Case sensitivity in comparisons 113
CASE_N function 56
CAST
data type conversions 666
DECIMAL data type 666
CAST expression 582, 585
ANSI DateTime conversion 650
CAST, in data type conversions 582
CHAR function. See CHARACTERS function.
CHAR2HEXINT function 379
Character
assignability rules for 627
conversion to formatted DATE conversion 599
translation 595
translation (internal to external) 372
Character conversion 583
character-to-character 592
character-to-numeric 608
character-to-UDT 625
string functions and 372
Character string functions. See String functions
Character to character conversion 592
CHARACTER_LENGTH function 498
CHARACTERS function 501
ANSI equivalent 498
Character-to-DATE conversion 597
Character-to-INTERVAL conversion 603
Character-to-numeric conversion 608
Character-to-Period conversion 605
Character-to-TIME conversion 614
Character-to-TIME conversion, implicit 579, 615, 621
Character-to-TIME WITH TIME ZONE conversion 614
Character-to-TIMESTAMP conversion 620
Character-to-TIMESTAMP conversion, implicit 579, 615, 621
Character-to-UDT conversion 625
CHARS function. See CHARACTERS function.
CLOB data types
aggregate functions and 218
arithmetic operators and 46
comparison operators and 101
predicates and 442
COALESCE expression 40
Comparison evaluations by data type 106
Comparison operators
= 102
> 102
>= 102
GE 102
general rules 105
GT 102
Japanese character sets 115

LE 102
LOB data types and 101
LT 102
NE 102
Period data types 118
results 105
Comparison rules
floating point data and 106
string 112
Concatenation operator 373
Conditional expressions 490
Constant expressions, aggregate functions and 216
CONTAINS predicate 451
Conversion
byte 588
byte to INTEGER using HASHBUCKET 523
character to character 592
character to DATE 597
character to formatted date 599
character to INTERVAL 603
character to numeric 608
character to Period 605
Character to TIME 614
Character to TIME WITH TIME ZONE 614
Character to TIME, implicit 579, 615, 621
character to TIMESTAMP 620
character to TIMESTAMP, implicit 579, 615, 621
character to UDT 625
data type 577
DATE to character 628
DATE to Period 637
DATE to TIMESTAMP 639
DATE to UDT 642
explicit, using CAST 582
explicit, using Teradata syntax 585
field mode 587
implicit 577
Interval to character 644
INTERVAL to INTERVAL 646
interval to numeric 650
interval to UDT 652
numeric 664
numeric to character 654
numeric to INTERVAL 662
numeric to UDT 668
Period to character 670
Period to DATE 673
Period to Period 675
Period to TIME 680
Period to TIMESTAMP 682
rounding and truncation rules 665
signed zone decimal 684
table showing supported types 578
Teradata DATE 632
TIME to character 688
TIME to Period 691
TIME to TIME 693
TIME to TIMESTAMP 696
TIME to UDT 699
TIMESTAMP to character 701
TIMESTAMP to DATE 705
TIMESTAMP to Period 708
TIMESTAMP to TIMESTAMP 713
Timestamp to timestamp 710
TIMESTAMP to UDT 716
using CAST to convert 582
using Teradata syntax to convert 585
CORR aggregate function, described 223
CORR window function, described 320
COS trigonometric function 91
COSH hyperbolic function 97
COUNT aggregate function, described 226
COUNT function
DateTime types and 170
Interval types and 170
COUNT window function, described 320
COVAR_POP aggregate function, described 231
COVAR_POP window function, described 320
COVAR_SAMP aggregate function, described 234
COVAR_SAMP window function, described 320
CSUM function, described 338
CUBE grouping set, GROUPING aggregate function and 237
Cumulative sum
CALENDAR view, computing with 339
computing 338
CURRENT_DATE function 529
CURRENT_TIME function 533
CURRENT_TIMESTAMP function 535
CURRENT_USER function 537

D

Data conversion rules
rounding and truncation 665
Data conversion rules, explicit 585
Data conversion rules, implicit 577
Data definition
byte conversion 588
byte to INTEGER conversion using HASHBUCKET 523
CAST in data type conversions 582
character conversion
character to character 592
character-to-numeric 608
character-to-UDT 625
character-to-DATE conversion 597
character-to-formatted DATE conversion 599
character-to-INTERVAL conversion 603
character-to-Period 605

character-to-TIME conversion 614
character-to-TIMESTAMP conversion 620
conversion rules
implicit rules 577
data type conversions 577
DATE conversion (Teradata) 632
DATE-to-character conversion 628
DATE-to-Period conversion 637
DATE-to-TIMESTAMP conversion 639
DATE-to-UDT conversion 642
Exact numeric-to-INTERVAL conversion 662
explicit type conversion rules 585
Interval-to-character conversion 644
interval-to-exact numeric conversion 650
INTERVAL-to-INTERVAL conversion 646
interval-to-UDT conversion 652
numeric conversion
numeric-to-character 654
numeric-to-numeric 664
numeric-to-UDT conversion 668
Period-to-character conversion 670
Period-to-DATE conversion 673
Period-to-Period conversion 675
Period-to-TIME conversion 680
Period-to-TIMESTAMP conversion 682
signed zone decimal conversion 684
TIMESTAMP-to-character conversion 701
TIMESTAMP-to-DATE conversion 705
TIMESTAMP-to-Period conversion 708
TIMESTAMP-to-TIMESTAMP conversion 710, 713
TIMESTAMP-to-UDT conversion 716
TIME-to-character conversion 688
TIME-to-Period conversion 691
TIME-to-TIME conversion 693
TIME-to-TIMESTAMP conversion 696
TIME-to-UDT conversion 699
Data type, conversions 577
Database, get default database 538
Date
get current date (Teradata) 539
get system date 529
DATE conversion
DATE-to-UDT 642
DATE conversion (Teradata) 632
Date expressions, Teradata 171
DATE values, scalar operations on 173
Date, aggregate operations and 216
DATE, as logical predicate 111
DATE, conversion to character 628
DateTime expressions 155
rules for, ANSI 157
DateTime functions, and scalar operations 171
DateTime scalar operations
arithmetic 168
restrictions on 155
DateTime types
aggregate functions and 170
assignment rules 152, 153
DATE-to-Period conversion 637
DATE-to-TIMESTAMP conversion 639
DATE-to-UDT conversion 642
DECIMAL(18)
CAST to DECIMAL(18) with a DECIMAL(15) default 666
DECIMAL/NUMERIC types
arithmetic expression and rounding 51
DEFAULT function 503
DEGREES function 94
DISTINCT, SELECT option 219
Division operator 46

E

End function 188, 190
ESCAPE, with LIKE predicate 473, 478
Exact numeric-to-INTERVAL conversion 662
EXCEPT operator 140
EXISTS predicate 453
EXP function 62
Exponentiation operator 46
Expressions, defined 19
EXTRACT function 180

F

Fallback AMP
identify maximum number with HASHBAKAMP 519
identify with HASHBAKAMP 519
FALSE 491
Field mode, data type conversions and 587
FLOAT data types
aggregate functions and 218
comparison operations and 106
FORMAT phrase 507
Functions
defined 17
types of 17

G

GROUP BY clause
aggregate functions and 215
rules for aggregate functions and constant expressions 216
Group count, example 332
GROUPING aggregate function
CUBE and 237
described 237
GROUPING SET and 237
ROLLUP and 237
GROUPING SET grouping set, GROUPING aggregate

function and 237

H

Hash index, ordered analytical functions and 310
HASHAMP function 516
HASHBAKAMP function 519
HASHBUCKET function 522
Hash-related functions 515
  HASHAMP 516
  HASHBAKAMP 519
  HASHBUCKET 522
  HASHROW 525
HASHROW function 525
Hyperbolic functions 97
  ACOSH 97
  ASINH 97
  ATANH 97
  COSH 97
  SINH 97
  TANH 97

I

Implicit type conversion 577
  byte-to-UDT 590
  character-to-UDT 625
  comparison operators and 108
  DATE-to-UDT 642, 699, 716
  interval-to-UDT 652
  numeric-to-UDT 668
IN predicate 459
INDEX function 382
  ANSI equivalent 370
INTERSECT operator 137
Interval conversion
  interval-to-character 644
  interval-to-interval 646
  interval-to-UDT 652
Interval expressions 160
  rules for, ANSI 167
INTERVAL function 192
Interval scalar operations
  arithmetic 168
  restrictions on 155
Interval types
  aggregate functions and 170
  assignment rules 152, 153
Interval-to-character conversion 644
Interval-to-exact numeric conversion 650
INTERVAL-to-INTERVAL conversion 646
Interval-to-UDT conversion 652
Inverse trigonometric functions 91
  ACOS 91
  ASIN 91
  ATAN 91
  ATAN2 91
IS NOT NULL predicate 466
IS NOT UNTIL_CHANGED predicate 468
IS NULL predicate 466
IS UNTIL_CHANGED predicate 468

J

Japanese character code notation, how to read 748
Join index, ordered analytical functions and 310

K

KURTOSIS aggregate function, described 240

L

LDIFF operator 201
Least squares, computing 347
LIKE predicate 470
Linear regression, computing 347
LN function 64
LOG function 66
Logical expressions
  BETWEEN predicate 450
  FALSE result 491
  NOT BETWEEN predicate 450
  TRUE result 491
  UNKNOWN result 491
Logical operators
  AND 490
  defined 490
  NOT 490
  OR 490
  search conditions and 490
Logical predicate
  conditional expression as 441
  DATE as 111
  DEFAULT function and 117, 444
  defined 441
  LOB data types and 442
  order of evaluation 491
  primitives, tabular summary of 442
  SQL use of 441
LOWER function 388

M

MAVG function, described 341
MAX aggregate function
  DateTime types and 170
  described 242
  Interval types and 170
MAX window function, described 320

MAXIMUM aggregate function. See MAX aggregate function.
MCHARACTERS function 495, 498
  ANSI equivalent 495
MDIFF function, described 344
MEETS predicate 480
MIN aggregate function
  DateTime types and 170
  described 245
  Interval types and 170
MIN window function, described 320
MINDEX function 370, 391
  ANSI equivalent 370
MINIMUM aggregate function. See MIN aggregate function.
MINUS operator 140
MLINREG function, described 347
MOD operator 46
Moving average, computing 341
Moving difference
  CALENDAR view, computing with 345
  computing 344
Moving sum, computing 350
MSUBSTR function 370, 403
  ANSI equivalent 370
MSUM function, described 350
Multiplication operator 46
mutator methods 572

N

Name, get user name 537, 548
NEW expression 566
NEXT function 197
NOT BETWEEN predicate 450
NOT EXISTS predicate 453
NOT IN predicate 459
  NOT EXISTS predicate and 454
  recursive queries and 464
NOT logical operator 490
NULLIF expression 42
NULLIFZERO function 68
Nulls
  aggregate operations and 217
  CASE expression and 40
  searching for/excluding 466
Numeric conversion
  numeric-to-character 654
  numeric-to-date 660
  numeric-to-numeric 664
  numeric-to-UDT 668
Numeric-to-character conversion 654
Numeric-to-date conversion 660
Numeric-to-numeric conversion 664
Numeric-to-UDT conversion 668

O

observer methods 572
OCTET_LENGTH function 508
OLAP functions. See Ordered analytical functions.
Operator, defined 18
operators
  arithmetic operators 212
  LDIFF operator 201
  P_INTERSECT operator 199
  P_NORMALIZE operator 207
  RDIFF operator 204
OR logical operator 490
  truth table 492
ORDER BY clause
  ordered analytical functions and 302, 310
  window specification and 310
Order of evaluation. See Logical predicate
Ordered analytical functions 297
  aggregates and 311
  AVG window function 320
  common characteristics of 308
  CORR window function 320
  COUNT window function 320
  COVAR_POP window function 320
  COVAR_SAMP window function 320
  CSUM 338
  derived tables and 313
  description 298
  examples 316
  extending Teradata queries 298
  GROUP BY clause 314
  hash indexes and 310
  HAVING clause 311
  join indexes and 310
  MAVG 341
  MAX window function 320
  MDIFF 344
  MIN window function 320
  MLINREG 347
  MSUM 350
  ORDER BY clause 302, 310
  PARTITION BY clause 301, 311
  PERCENT_RANK window function 352
  Period data types and 310
  QUALIFY clause 308, 311
  QUANTILE 356
  RANK 359
  RANK window function 362
  recursive queries and 310
  REGR_AVGX window function 320
  REGR_AVGY window function 320
  REGR_COUNT window function 320
  REGR_INTERCEPT window function 320

REGR_R2 window function 320
REGR_SLOPE window function 320
REGR_SXX window function 320
REGR_SXY window function 320
REGR_SYY window function 320
result order 310
ROW_NUMBER window function 365
ROWS clause 306
STDDEV_POP window function 320
STDDEV_SAMP window function 320
SUM window function 320
syntax alternatives for 299
Teradata OLAP functions 300
Teradata Warehouse Miner and 298
VAR_POP window function 320
VAR_SAMP window function 320
views and 313
window 300
window functions 300
OVERLAPS predicate 482

P

P_INTERSECT operator 199
P_NORMALIZE operator 207
PARTITION BY clause
affect on spool space 311
ordered analytical functions and 301, 311
Partitioned primary index. See PPI.
PERCENT_RANK window function, described 352
Period data types
logical predicates and 443
Period Value Constructor 210
Period-to-character conversion 670
Period-to-DATE conversion 673
Period-to-Period conversion 675
Period-to-TIME conversion 680
Period-to-TIMESTAMP conversion 682
POSITION function 370, 391
PPI
defined 58, 78
maximum partitions when using CASE_N 58
maximum partitions when using RANGE_N 79
multilevel 58, 78
system-derived columns 58, 79
PPI functions
CASE_N 56
RANGE_N 75
Precedence
arithmetic expressions 51
logical operators 491
operator 51
set operators 124
PRECEDES predicate 486
Predicate quantifiers
ALL 445
ANY 445
SOME 445
Predicates
BETWEEN 450
CONTAINS 451
DEFAULT function and 117, 444
defined 20
EXISTS 453
IN 459
IS NOT NULL 466
IS NOT UNTIL_CHANGED 468
IS NULL 466
IS UNTIL_CHANGED 468
LIKE 470
logical 441
MEETS 480
NOT BETWEEN 450
NOT EXISTS 453
NOT IN 459
OVERLAPS 482
PRECEDES 486
quantifiers 445
SUCCEEDS 488
PRIOR function 195
PROFILE function 541
Profiles, getting the current profile 541
proximity functions
NEXT function 197
PRIOR function 195

Q

QUALIFY clause, ordered analytical functions and 308
Quantifiers
ALL 445
ANY 445
SOME 445
QUANTILE function, described 356

R

RADIANS function 94
RANDOM function 71
RANDOM function, and valued CASE 24
RANGE_N function 75
RANK function, described 359
RANK window function, described 362
RDIFF operator 204
REGR_AVGX aggregate function, described 248
REGR_AVGX window function, described 320
REGR_AVGY aggregate function, described 251
REGR_AVGY window function, described 320
REGR_COUNT aggregate function, described 254

REGR_COUNT window function, described 320
REGR_INTERCEPT aggregate function, described 258
REGR_INTERCEPT window function, described 320
REGR_R2 aggregate function, described 262
REGR_R2 window function, described 320
REGR_SLOPE aggregate function, described 266
REGR_SLOPE window function, described 320
REGR_SXX aggregate function, described 270
REGR_SXX window function, described 320
REGR_SXY aggregate function, described 273
REGR_SXY window function, described 320
REGR_SYY aggregate function, described 276
REGR_SYY window function, described 320
Remaining average, example 310
Remaining count, example 332
Remaining sum, example 337
ROLE function 531, 542
Roles, getting the current role 531, 542
ROLLUP grouping set, GROUPING aggregate function and 237
Rounding
  arithmetic operators and DECIMAL/NUMERIC data 51
  data type conversion rules 665
Row length errors
  UNION operator 143
ROW_NUMBER window function, described 365
Rowhash, identify with HASHROW function 525
ROWNUM. See ROW_NUMBER window function.
ROWNUMBER. See ROW_NUMBER window function.
ROWS clause
  defined 306
  ordered analytical functions and 306

S

Scalar, converting scalar value expressions 582
SDF
  and data type default formats 613
  Currency 611
  CurrencyName 611
  GroupingRule 611
  GroupSeparator 611
  RadixSeparator 611
  relationship to FORMAT phrase 611
Search conditions
  definition 490
  logical operators and 490
SESSION function 545
Session, get session number 545
Set operators
  ALL option 125
  derived tables and 127
  described 121
  EXCEPT operator 140
  INSERT...SELECT statements containing 130
  INTERSECT operator 137
  MINUS operator 140
  precedence 124
  rules for 123
  rules for connecting queries by 133
  set result, attributes of 125
  subqueries containing 128
  UNION operator 142
  view definitions containing 132
Signed zone decimal conversion 684
SIN trigonometric function 91
SINH hyperbolic function 97
SKEW aggregate function, described 279
SOME predicate quantifier 445
SOUNDEX function, described 394
Specification for data formatting, see SDF
SQL expressions
  aggregate functions
    AVG 220
    CORR 223
    COUNT 226
    COVAR_POP 231
    COVAR_SAMP 234
    DISTINCT option 219
    GROUPING 237
    HAVING clause and 218
    KURTOSIS 240
    MAX 242
    MIN 245
    REGR_AVGX 248
    REGR_AVGY 251
    REGR_COUNT 254
    REGR_INTERCEPT 258
    REGR_R2 262
    REGR_SLOPE 266
    REGR_SXX 270
    REGR_SXY 273
    REGR_SYY 276
    SKEW 279
    STDDEV_POP 282
    STDDEV_SAMP 285
    SUM 288
    VAR_POP 291
    VAR_SAMP 294
    WHERE clause and 218
  arithmetic functions
    ABS function 54
    DEGREES function 94
    EXP function 62
    LN function 64
    LOG function 66
    NULLIFZERO function 68
    RADIANS function 94

- RANDOM function 71
- SQRT function 82
- ZEROIFNULL function 88
- arithmetic operators
  - addition operator 46
  - division operator 46
  - exponentiation 46
  - MOD operator 46
  - multiplication operator 46
  - precedence 51
  - subtraction operator 46
  - unary minus operator 46
  - unary plus operator 46
- CASE operation 23
  - COALESCE expression 40
  - NULLIF expression 42
  - searched CASE 27
  - valued CASE 24
- comparison operators
  - = 102
  - > 102
  - >= 102
  - EQ 102
  - GE 102
  - GT 102
  - Japanese character set comparison operators 115
  - LE 102
  - LT 102
  - NE 102
  - Period data type comparison operators 118
- conditional expressions 490
- hyperbolic functions 97
  - ACOSH 97
  - ASINH 97
  - ATANH 97
  - COSH 97
  - SINH 97
  - TANH 97
- inverse trigonometric functions 91
  - ACOS 91
  - ASIN 91
  - ATAN 91
  - ATAN2 91
- logical expressions
  - BETWEEN 450
  - NOT BETWEEN 450
- trigonometric functions 91
  - COS 91
  - SIN 91
  - TAN 91

SQL functions
- attribute functions 495
  - BYTES 496
  - CHARACTER_LENGTH 498
  - CHARACTERS 501
  - DEFAULT 503
  - FORMAT 507
  - MCHARACTERS 495, 498
  - OCTET_LENGTH 508
  - TITLE 511
  - TYPE 512
- built-in functions
  - ACCOUNT 528
  - CURRENT_DATE 529
  - CURRENT_TIME 533
  - CURRENT_TIMESTAMP 535
  - CURRENT_USER 537
  - DATABASE 538
  - DATE 539
  - PROFILE 541
  - ROLE 531, 542
  - SESSION 545
  - TIME 546
  - USER 548
- byte strings
  - BYTES 496
  - TRIM 420
- hash-related functions 515
  - HASHAMP 516
  - HASHBAKAMP 519
  - HASHBUCKET 522
  - HASHROW 525
- Ordered analytical functions
  - AVG window function 320
  - CORR window function 320
  - COUNT window function 320
  - COVAR_POP window function 320
  - COVAR_SAMP window function 320
  - CSUM 338
  - MAVG 341
  - MAX window function 320
  - MDIFF 344
  - MIN window function 320
  - MLINREG 347
  - MSUM 350
  - PERCENT_RANK window function 352
  - QUANTILE 356
  - RANK 359
  - RANK window function 362
  - REGR_AVGX window function 320
  - REGR_AVGY window function 320
  - REGR_COUNT window function 320
  - REGR_INTERCEPT window function 320
  - REGR_R2 window function 320
  - REGR_SLOPE window function 320
  - REGR_SXX window function 320
  - REGR_SXY window function 320
  - REGR_SYY window function 320

ROW_NUMBER window function 365
STDDEV_POP window function 320
STDDEV_SAMP window function 320
SUM window function 320
VAR_POP window function 320
VAR_SAMP window function 320
partitioning functions
CASE_N 56
RANGE_N 75
string functions 369
CHAR2HEXINT 379
concatenation operator 373
INDEX 382
LOWER 388
MINDEX 391
MSUBSTR 403
POSITION 391
SOUNDEX 394
STRING_CS 398
SUBSTR 401, 403
SUBSTRING 401
TRANSLATE 407
TRANSLATE_CHK 416
TRIM 420
TRIM and concatenation 422
UPPER 424
VARGRAPHIC 427
WIDTH_BUCKET function 84
SQRT function 82
STDDEV_POP aggregate function, described 282
STDDEV_POP window function, described 320
STDDEV_SAMP aggregate function, described 285
STDDEV_SAMP window function, described 320
String functions
CHAR2HEXINT 379
implicit character type conversion 372
INDEX 382
LOWER 388
MINDEX 370, 391
MSUBSTR 370, 403
POSITION 391
rules 372
server character sets and 372
SOUNDEX 394
STRING_CS 398
SUBSTR 401, 403
SUBSTRING 401
TRANSLATE 407
TRANSLATE_CHK 416
TRIM 420
UPPER 424
VARGRAPHIC 427
STRING_CS function, described 398
Subqueries, comparison operators and 104
SUBSTR function 401, 403
ANSI equivalent 370
SUBSTRING function 370, 401
Subtraction operator 46
SUCCEEDS predicate 488
SUM aggregate function, described 288
SUM function, Interval types and 170
SUM window function, described 320
Syntax, how to read 743
SYS_CALENDAR system database
CALENDAR view
cumulative sum, and calculating 339
moving difference, and calculating 345

T

TAN trigonometric function 91
TANH hyperbolic function 97
Teradata conversion syntax 585
Teradata OLAP functions. See Ordered analytical functions.
Teradata Warehouse Miner 298
Time
get current time (Teradata) 546
get system time 533
TIME conversion
TIME-to-UDT 699
Time expressions, Teradata 171
TIME function 546
Time stamp, get system time stamp 535
Time zone comparisons 159
Time zone, get time zone displacement 533
TIME, conversion to character 688
TIMESTAMP
arithmetic 166
conversion to character 701
TIMESTAMP conversion
TIMESTAMP-to-UDT 716
TIMESTAMP-to-DATE conversion 705
TIMESTAMP-to-Period conversion 708
TIMESTAMP-to-TIMESTAMP conversion 710, 713
TIMESTAMP-to-UDT conversion 716
TIME-to-Period conversion 691
TIME-to-TIME conversion 693
TIME-to-TIMESTAMP conversion 696
TIME-to-UDT conversion 699
TITLE function 511
TRANSLATE function 407
TRANSLATE_CHK function 416
Translation, character 595
Trigonometric functions 91
COS 91
SIN 91
TAN 91
TRIM function 420

TRIM function and concatenation 422
TRUE 491
Type conversion, implicit 577
TYPE function 512

U

UDF expression
  aggregate 555
  scalar 552
  table 558
UDM invocation 572
UDT data types
  aggregate functions and 221
  arithmetic operators and 47
  CASE expression and 25, 28, 32
  COALESCE expression and 41
  comparison operators and 107
  conversion 718, 721, 725, 728, 731, 734, 737, 740
  hash-related functions and 517, 520
  implicit type conversions and 579
  logical predicates and 443
  method invocation 572
  mutator methods 572
  NEW expression 562, 566
  NULL value 217
  NULLIF expression and 43
  observer methods 572
  ordered analytical functions and 310
  set operators and 124
  string functions and 374
UDT expression 562
UDT-to-byte type conversion 718
UDT-to-character type conversion 721
UDT-to-DATE type conversion 725
UDT-to-INTERVAL type conversion 728
UDT-to-numeric type conversion 731
UDT-to-TIME type conversion 734
UDT-to-TIMESTAMP type conversion 737
UDT-to-UDT type conversion 740
Unary minus operator 46
Unary plus operator 46
UNION operator 142
  Effect of sort key and row size on ability to perform 143
  outer join and 146
  reason for unexpected row length errors 143
Universal Coordinated Time, see UTC
UNKNOWN 491
UPPER function 424
USER function 548
User-defined types. See UDT data types
Username, get user name 537, 548
UTC
  and time conversions 616, 622
UTF16 client character set
  KANJI1 translation
    internal to external 372
  OCTET_LENGTH and 509
UTF8 client character set
  KANJI1 translation 372
  OCTET_LENGTH and 509

V

VAR_POP aggregate function, described 291
VAR_POP window function, described 320
VAR_SAMP aggregate functions, described 294
VAR_SAMP window function, described 320
VARGRAPHIC function 427
VARGRAPHIC function conversion tables 430

W

WIDTH_BUCKET function, described 84
Wildcards, used with LIKE predicate 471
Window aggregate functions
  defined 307
  difference between aggregate functions and 307
Window functions. See Ordered analytical functions.
Window, defined 300

Z

ZEROIFNULL function 88