Teradata
a division of NCR

Teradata® RDBMS

SQL Reference - Volume 5
Functions and Operators

V2R5.0

B035-1101-122A
December 2002

The product described in this book is a licensed product of NCR Corporation.

BYNET is an NCR trademark registered in the U.S. Patent and Trademark Office.
CICS, CICS/400, CICS/600, CICS/ESA, CICS/MVS, CICSPLEX, CICSVIEW, CICS/VSE, DB2, DFSMS/MVS, DFSMS/VM, IBM, NQS/MVS, OPERATING SYSTEM/2, OS/2, PS/2, MVS, QMS, RACF, SQL/400, VM/ESA, and VTAM are trademarks or registered trademarks of International Business Machines Corporation in the U. S. and other countries.
DEC, DECNET, MICROVAX, VAX and VMS are registered trademarks of Digital Equipment Corporation.
HEWLETT-PACKARD, HP, HP BRIO, HP BRIO PC, and HP-UX are registered trademarks of Hewlett-Packard Co.
KBMS is a trademark of Trinzic Corporation.
INTERTEST is a registered trademark of Computer Associates International, Inc.
MICROSOFT, MS-DOS, MSN, The Microsoft Network, MULTIPLAN, SQLWINDOWS, WIN32, WINDOWS, WINDOWS 2000, and WINDOWS NT are trademarks or registered trademarks of Microsoft Corporation.
SAS, SAS/C, SAS/CALC, SAS/CONNECT, and SAS/CPE are registered trademarks of SAS Institute Inc.
SOLARIS, SPARC, SUN and SUN OS are trademarks of Sun Microsystems, Inc.
TCP/IP protocol is a United States Department of Defense Standard ARPANET protocol.
TERADATA and DBC/1012 are registered trademarks of NCR International, Inc.
UNICODE is a trademark of Unicode, Inc.
UNIX is a registered trademark of The Open Group.
X and X/OPEN are registered trademarks of X/Open Company Limited.
YNET is a trademark of NCR Corporation.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS-IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. IN NO EVENT WILL NCR CORPORATION (NCR) BE LIABLE FOR ANY INDIRECT, DIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS OR LOST SAVINGS, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The information contained in this document may contain references or cross references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that NCR intends to announce such features, functions, products, or services in your country. Please consult your local NCR representative for those features, functions, products, or services available in your country.

Information contained in this document may contain technical inaccuracies or typographical errors. Information may be changed or updated without notice. NCR may also make improvements or changes in the products or services described in this information at any time without notice.

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.ncr.com

or write:

Information Engineering
NCR Corporation
100 North Sepulveda Boulevard
El Segundo, CA 90245-4361
U.S.A.

Any comments or materials (collectively referred to as "Feedback") sent to NCR will be deemed non-confidential. NCR will have no obligation of any kind with respect to Feedback and will be free to use, reproduce, disclose, exhibit, display, transform, create derivative works of and distribute the Feedback and derivative works thereof without limitation on a royalty-free basis. Further, NCR will be free to use any ideas, concepts, know-how or techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, or marketing products or services incorporating Feedback.

Copyright © 1996-2002, NCR Corporation
All Rights Reserved

Preface

Supported Software Release

This book supports Teradata RDBMS V2R5.0.

Changes to This Book

This book includes the following changes to support the current release:

Date	Description
December 2002	• New CASE_N and RANGE_N built-in functions for defining a partitioned primary index • New ROLE and PROFILE built-in functions to support roles and profiles • Multiple aggregate functions can use DISTINCT with different expressions • New rules and examples added to data type conversions to support new FORMAT phrase formatting characters • Teradata conversion syntax supported for ANSI DateTime types • Implicit conversion of CHARACTER to TIME and CHARACTER to TIMESTAMP now supported • "OLAP Functions" chapter renamed to "Ordered Analytical Functions" • Chapter on OLAP functions uses the term *ordered analytical functions* to discuss what was previously called *OLAP functions* • New ROW_NUMBER ordered analytical function • MAX and MIN now supported as window aggregate functions • Ordered analytical functions support an enhanced window specification
June 2001	• Volume renamed • New trigonometric and hyperbolic functions • New aggregate functions • New OLAP functions • Chapter on string functions rewritten • DR fixes and enhancement requests
September 2000	• Usability enhancements

Date	Description
June 2000	• New aggregates • New aggregate window functions and new OLAP functions • General reorganization and rewriting of material to enhance usability

About This Book

Purpose

Teradata RDBMS SQL Reference, Volume 5 describes the functions, operators, expressions, and predicates of Teradata Structured Query Language (SQL).

This preface describes the organization of *Teradata RDBMS SQL Reference, Volume 5* and identifies information you should know before using it. This book should be used in conjunction with the other volumes of *Teradata RDBMS SQL Reference*.

Audience

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

How This Book Is Organized

This books contains thirteen chapters and two appendixes.

- Chapter 1: "CASE Expressions" describes the valued and searched variants of the CASE operation and its two null-handling shorthand expressions COALESCE and NULLIF.
- Chapter 2: "Arithmetic Operators and Functions / Trigonometric / Hyperbolic Functions" describes the SQL arithmetic operators, the SQL arithmetic functions, the null conversion functions NULLIF ZERO and ZEROIFNULL, and trigonometric and hyperbolic functions.
- Chapter 3: "Comparison Operators" describes the SQL comparison operators, the [NOT] BETWEEN ... AND range comparison operator, and the OVERLAPS operator for DateTime and Interval data.
- Chapter 4: "Set Operators" describes the SQL set operators INTERSECT, MINUS, and UNION.
- Chapter 5: "DateTime and Interval Functions and Expressions" describes scalar operations on Teradata-style dates and ANSI-style dates, times, and intervals, and describes the ADD_MONTHS and EXTRACT functions
- Chapter 6: "Aggregate Functions" describes aggregate functions, with particular note to the way these functions handle nulls.
- Chapter 7: "Ordered Analytical Functions" describes the set of ordered analytical functions for analyzing series of column values, including cumulative sums, group sums, moving averages, moving differences, moving linear regressions, moving sums, QUANTILE and RANK functions, PERCENT_RANK, and ROW_NUMBER.

- Chapter 8: "String Functions" describes the ANSI character string functions and their Teradata extensions, including the TRANSLATE_CHK function for testing international character strings. Particular emphasis is placed on how the character string functions handle international character sets.
- Chapter 9: "Logical Predicates" describes the logical predicates and quantifiers.
- Chapter 10: "Attribute Functions" describes several functions and their Teradata extensions that return information about the attributes of their operands. CHAR2HEXINT, which is sometimes considered an attribute function, is documented as a character string function in Chapter 8: "String Functions."
- Chapter 11: "Hash-Related Functions" describes a set of functions that return information about how system hash buckets correspond to AMPs as well as a function to compute the row hash value for an expression.
- Chapter 12: "Built-In Functions" describes a set of niladic functions that return information about the system date, time, and timestamp, the user name of the current user, several variables associated with the current user including its account string, default database, and session number, and the role and profile of the current session.
- Chapter 13: "Data Type Conversions" describes the CAST function and a host of rules, both implicit and explicit, for converting one data type to another.
- Appendix A: "Notation Conventions" explains how to read syntax diagrams, character shorthand notation, and predicate calculus notation.
- Appendix B: "SQL Functions, Operators, Expressions, and Predicates" provides a very brief overview of the distinguishing characteristics of SQL functions, operators, expressions, and predicates.

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 the Teradata RDBMS, read *Introduction to Teradata RDBMS* before reading this book.

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

List of Acronyms

The following acronyms, listed in alphabetical order, apply to this release:

AMP	Access Module Processor vproc
ANSI	American National Standards Institute
BTEQ	Basic TEradata Query facility
BYNET	Banyan Network - High speed interconnect
CJK	Chinese, Japanese, and Korean
CLIv2	Call Level Interface Version 2
cs0, cs1, cs2, cs3	Four code sets (codeset 0, 1, 2, and 3) used in EUC encoding
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
LT/ST	Large Table/Small Table (join)
NPPI	Non-Partitioned Primary Index
NUPI	Non-Unique Primary Index
NUSI	Non-Unique Secondary Index
OLAP	On-Line Analytical Processing
OLTP	On-Line Transaction Processing
PDE	Parallel Database Extensions
PE	Parsing Engine vproc
PI	Primary Index
PK	Primary Key
PPI	Partitioned Primary Index
RDBMS	Relational Database Management System
SDF	Specification for Data Formatting

SQL	Structured Query Language
UCS-2	Universal Coded Character Set containing 2 bytes
UPI	Unique Primary Index
USI	Unique Secondary Index
vproc	Virtual Process

Technical Information on the Web

The NCR home page (http://www.ncr.com) provides links to numerous sources of information about Teradata. Among the links provided are sites that deal with the following subjects:

- Contacting technical support
- Enrolling in customer education courses
- Ordering and downloading product documentation
- Accessing case studies of customer experiences with Teradata
- Accessing third party industry analyses of Teradata data warehousing products
- Accessing white papers
- Viewing or subscribing to various online periodicals

Contents

Preface

About This Book iii
List of Acronyms v
Technical Information on the Web vii

Chapter 1:
CASE Expressions

CASE 1–2
Valued CASE Expression 1–3
Searched CASE Expression 1–7
Error Conditions 1–13
Rules Governing the Data Type of a CASE Expression 1–14
Output Format for a CASE Expression 1–21
CASE and Nulls 1–23
COALESCE Expression 1–25
NULLIF Expression 1–27

Chapter 2:
Arithmetic Operators and Functions / Trigonometric / Hyperbolic Functions

Arithmetic Operators 2–2
Binary Arithmetic Result Data Types 2–3
Structure of Arithmetic Expressions 2–7
Arithmetic Functions 2–9
ABS 2–10
CASE_N 2–12
EXP 2–17
LN 2–19
LOG 2–20
NULLIFZERO 2–21
RANDOM 2–23
RANGE_N 2–26
SQRT 2–33
WIDTH_BUCKET 2–34
ZEROIFNULL 2–37

Trigonometric Functions 2–39
COS 2–40
SIN 2–42
TAN 2–44
ACOS 2–46
ASIN 2–48
ATAN 2–50
ATAN2 2–52
Hyperbolic Functions 2–54
COSH 2–55
SINH 2–56
TANH 2–57
ACOSH 2–58
ASINH 2–60
ATANH 2–62

Chapter 3:
Comparison Operators
Comparison Operators 3–2
Comparison Operators in Logical Expressions 3–4
Comparisons That Produce TRUE Results 3–5
Methods of Data Type Evaluation 3–6
Rules for Comparisons 3–7
Time Zone Comparisons 3–9
Comparisons for the KANJI1 Character Data Type 3–11
Case Sensitivity in Comparisons 3–13

Chapter 4:
Set Operators
Overview of Set Operators 4–2
Rules for Set Operators 4–5
Precedence 4–6
Retaining Duplicate Rows Using the ALL Option 4–8
Attributes of a Set Result 4–9
Set Operators With Derived Tables 4–12
Set Operators in Subqueries 4–13
Set Operators in INSERT … SELECT Statements 4–15
Set Operators in View Definitions 4–17
Connecting Queries by Set Operators 4–19
INTERSECT Operator 4–22
MINUS/EXCEPT Operator 4–25

UNION Operator 4–27

Chapter 5:
DateTime and Interval Functions and Expressions

Overview 5–2
ANSI DateTime and Interval Data Type Assignment Rules 5–3
Scalar Operations on ANSI SQL-99 DateTime and Interval Values 5–5
ANSI DateTime Expressions 5–7
ANSI Interval Expressions 5–12
Arithmetic Operators 5–23
Aggregate Functions and ANSI DateTime and Interval Data Types 5–25
Scalar Operations and DateTime Functions 5–26
Teradata Date and Time Expressions 5–27
Scalar Operations on Teradata DATE Values 5–29
ADD_MONTHS 5–30
EXTRACT 5–36

Chapter 6:
Aggregate Functions

Aggregate Functions 6–2
AVG 6–5
CORR 6–7
COUNT 6–10
COVAR_POP 6–15
COVAR_SAMP 6–17
KURTOSIS 6–20
MAX 6–22
MIN 6–24
REGR_AVGX 6–26
REGR_AVGY 6–29
REGR_COUNT 6–32
REGR_INTERCEPT 6–34
REGR_R2 6–37
REGR_SLOPE 6–40
REGR_SXX 6–43
REGR_SXY 6–46
REGR_SYY 6–49
SKEW 6–52
STDDEV_POP 6–54
STDDEV_SAMP 6–57
SUM 6–60

VAR_POP 6–62
VAR_SAMP 6–64

Chapter 7:
Ordered Analytical Functions

Ordered Analytical Functions 7–2
How Ordered Analytical Functions Extend Teradata Queries 7–3
Syntax Alternatives for Ordered Analytical Functions 7–5
Ordered Analytical Computations 7–6
Window Feature 7–7
Applying Windows to Aggregate Functions 7–12
Characteristics of Ordered Analytical Functions 7–13
Nesting Aggregates in Ordered Analytical Functions 7–16
GROUP BY Clause 7–18
Using Ordered Analytical Functions: Some Examples 7–20
Ordered Analytical Functions 7–23
AVG 7–24
COUNT 7–30
CSUM 7–37
MAVG 7–40
MAX 7–43
MDIFF 7–48
MIN 7–51
MLINREG 7–56
MSUM 7–59
PERCENT_RANK 7–61
QUANTILE 7–64
RANK 7–67
RANK 7–70
ROW_NUMBER 7–72
SUM 7–74

Chapter 8:
String Functions

String Functions 8–2
Effects of Server Character Sets on Character String Functions 8–5
Concatenation Operator 8–6
CHAR2HEXINT 8–12
INDEX 8–15
LOWER 8–21
POSITION 8–23

SOUNDEX 8–26
SUBSTRING/SUBSTR 8–30
TRANSLATE 8–36
TRANSLATE_CHK 8–43
TRIM 8–47
UPPER 8–51
VARGRAPHIC 8–53
VARGRAPHIC Function Conversion Tables 8–56

Chapter 9:
Logical Predicates
Logical Predicates 9–2
Case Blind Comparisons 9–4
Comparisons Using Subqueries 9–8
BETWEEN/NOT BETWEEN 9–10
ANY/ALL/SOME 9–11
EXISTS/NOT EXISTS 9–14
IN/NOT IN 9–20
IS NULL/IS NOT NULL 9–25
Logical Operators and Search Conditions 9–27
LIKE 9–32
OVERLAPS 9–43

Chapter 10:
Attribute Functions
Attribute Functions 10–2
BYTES 10–3
CHARACTER_LENGTH 10–5
CHARACTERS 10–8
FORMAT 10–9
OCTET_LENGTH 10–10
TYPE 10–12

Chapter 11:
Hash-Related Functions
What are Hash-Related Functions? 11–2
HASHAMP 11–3
HASHBAKAMP 11–5
HASHBUCKET 11–7
HASHROW 11–9

Chapter 12:
Built-In Functions

What are Built-In Functions? 12–2
ACCOUNT 12–3
CURRENT_DATE 12–4
CURRENT_TIME 12–5
CURRENT_TIMESTAMP 12–7
DATABASE 12–9
DATE 12–10
PROFILE 12–12
ROLE 12–13
SESSION 12–14
TIME 12–15
USER 12–16

Chapter 13:
Data Type Conversions

Data Type Conversions 13–2
Implicit Type Conversions 13–3
CAST in Explicit Data Type Conversions 13–8
Teradata Conversion Syntax in Explicit Data Type Conversions 13–11
Data Conversions in Field Mode 13–13
Numeric Conversion 13–14
Numeric-to-Numeric Conversion 13–15
Numeric-to-Character Conversion 13–17
Numeric-to-DATE Conversion 13–21
Character Conversion 13–23
Character-to-Numeric Conversion 13–24
Character-to-Character Conversion 13–28
Character Data Type Assignment Rules 13–31
Implicit Character-to-Character Translation 13–32
BYTE Conversion 13–34
Teradata DATE Conversion 13–36
ANSI DateTime and Interval Conversion 13–39
DATE to TIMESTAMP Conversion 13–40
DATE-to-Character Conversion 13–41
TIME-to-TIME Conversion 13–43
TIME to TIMESTAMP Conversion 13–44
TIME-to-Character Conversion 13–45
TIMESTAMP-to-TIMESTAMP Conversion 13–47

TIMESTAMP-to-DATE Conversion 13–48
TIMESTAMP-to-TIME Conversion 13–49
TIMESTAMP-to-Character Conversion 13–50
INTERVAL-to-INTERVAL Conversion 13–52
INTERVAL-to-Exact Numeric Conversion 13–54
INTERVAL-to-Character Conversion 13–55
Character-to-DATE Conversion 13–56
Character-to-TIME Conversion 13–58
Character-to-TIMESTAMP Conversion 13–61
Character-to-INTERVAL Conversion 13–64
Exact Numeric-to-INTERVAL Conversion 13–65
Signed Zone DECIMAL Conversion 13–66

Appendix A:
Notation Conventions
Syntax Diagram Conventions A–2
Character Shorthand Notation Used In This Book A–6
Predicate Calculus Notation Used in This Book A–8

Appendix B:
SQL Functions, Operators, Expressions, and Predicates
SQL Functions B-2
SQL Operators B-3
SQL Expressions B-4
SQL Predicates B-5

Index Index-1

Chapter 1:

CASE Expressions

This chapter describes:

- CASE expression, including:
  - Valued CASE expression
  - Searched CASE expression
- The two special shorthand variants of CASE expression:
  - COALESCE
  - NULLIF

CASE

Purpose

Specifies alternate values for a conditional expression or expressions based on various equality and TRUTH conditions.

ANSI Compliance

CASE is ANSI SQL-99-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 the Teradata RDBMS.

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 1-3.
- Searched CASE is described under "Searched CASE Expression" on page 1-7.

CASE Shorthands for Handling Nulls

Two shorthand forms of CASE are provided to deal with nulls:

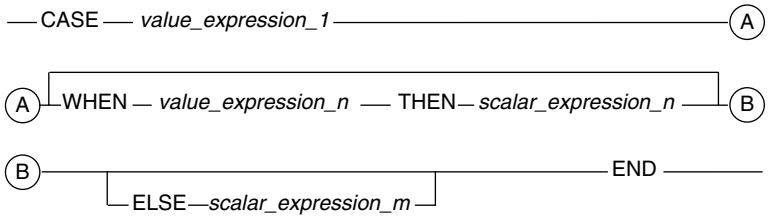
- COALESCE is described under "COALESCE Expression" on page 1-25.
- NULLIF is described under "NULLIF Expression" on page 1-27.

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



1101A012

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*. The data type for the value computed by *scalar_expression_n* is limited to one of the following. • Numeric • Character • Byte • DateTime • Interval

Syntax element ...	Specifies ...
scalar_expression_m	an expression whose value is to be returned as the result of the CASE expression if evaluation falls through to the ELSE clause. The data type for *scalar_expression_m* is limited to one of the following. • Numeric • Character • Byte • DateTime • Interval

ANSI Compliance

Valued CASE is ANSI SQL-99-compliant.

Teradata does not enforce the ANSI restriction that *value_expression_1* must be a deterministic function. In particular, Teradata 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*.

Definition

The following explains the Valued CASE syntax:

WHEN this condition is met ...	THEN the result is ...
value_expression_1 = *value_expression_n*	*scalar_expression_n*
value_expression_1 <> any *value_expression_n*	*scalar_expression_m*

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. Otherwise, an error is returned.

In a default title for an expression, the CASE expression appears as:

```
<CASE expression>
```

Using CASE in an Expression

Whenever you combine CASE with non-comparison operators, such as || or * or +, you must enclose the entire CASE expression in parentheses.

For an example, see "Example 2" on page 1-6.

This requirement is not ANSI SQL-99-compliant.

Parentheses are not required when you use CASE with comparison operators, Teradata conversion syntax, or alias syntax (AS).

For an example, see "Example 3" on page 1-6.

Additional Notes

For additional notes on ...	See ...
error conditions	"Error Conditions" on page 1-13.
the data type of a CASE expression	"Rules Governing the Data Type of a CASE Expression" on page 1-14
output format	"Output Format for a CASE Expression" on page 1-21
nulls and CASE expressions	"CASE and Nulls" on page 1-23

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

The following example shows how you must enclose CASE within parentheses when you combine it with a non-comparison operator.

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

Example 3

A CASE expression can be used in place of any *value-expression*.

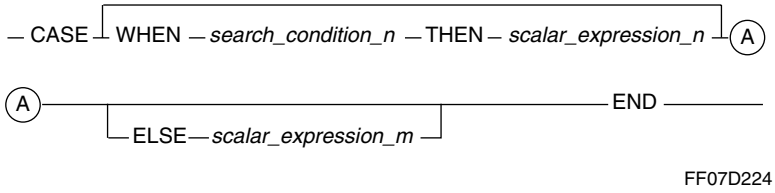
```
SELECT *
FROM t
WHERE x = CASE y
           WHEN 2
           THEN 1001
           WHEN 5
           THEN 1002
          END;
```

Searched CASE Expression

Purpose

Evaluates a search condition and returns one of a WHEN clause-defined set of scalar values whenever 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



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. The returned value has one of the following data types. • Numeric • Character • Byte • DateTime • Interval
scalar_expression_m	a scalar expression whose value is returned when no *search_condition_n* evaluates to TRUE. The returned value has one of the following data types. • Numeric • Character • Byte • DateTime • Interval

ANSI Compliance

Searched CASE is ANSI SQL-99-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.

In a default title for an expression, the CASE expression appears as:

```
<CASE expression>
```

Rules for WHEN Search Conditions

WHEN search conditions have the properties listed below:

- Can take the form of any comparison operator, such as LIKE, =, or <>.
- Can be a quantified predicate, such as ALL or ANY.
- 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.

Using CASE in an Expression

Whenever you combine CASE with non-comparison operators, such as || or * or +, you must enclose the entire CASE expression in parentheses.

For an example, see "Example 6" on page 1-12.

This requirement is not ANSI SQL-99-compliant.

Parentheses are not required when you use CASE with comparison operators, Teradata conversion syntax, or alias syntax (AS).

For examples, see "Example 2" on page 1-9 and "Example 3" on page 1-9.

Additional Notes

For additional notes on ...	See ...
error conditions	"Error Conditions" on page 1-13.
the data type of a CASE expression	"Rules Governing the Data Type of a CASE Expression" on page 1-14
output format	"Output Format for a CASE Expression" on page 1-21
nulls and CASE expressions	"CASE and Nulls" on page 1-23

Example 1

The following statement is equivalent to the first example of the valued form of CASE on page 1-6:

```
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 to the requestor.

```
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 MONTH, 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 MONTH;
```

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 'Under 8 Years'
            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

Example 6

The following example shows how you must enclose CASE within parentheses when you combine it with a non-comparison operator.

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

Error Conditions

The following conditions or expressions are considered illegal in a CASE expression:

- A condition after the keyword CASE is supplied.

  For example:

```
SELECT CASE a=1
         WHEN 1
         THEN 1
         ELSE 0
       END
FROM t;
```

- A non-valid WHEN expression is supplied in a valued CASE expression.

  For example:

```
SELECT CASE a
         WHEN a=1
         THEN 1
         ELSE 0
       END
FROM t;
```

- A non-valid WHEN condition is supplied in a searched CASE expression.

  For example:

```
SELECT CASE
         WHEN a
         THEN 1
         ELSE 0
       END
FROM t;


SELECT CASE
         WHEN NULL
         THEN 'NULL'
       END
FROM table_1;
```

- A subquery is specified in a WHEN condition of a searched CASE expression.

  For example:

```
SELECT CASE
         WHEN t.a IN
          (SELECT u.a
           FROM u)
         THEN 1
         ELSE 0
       END
FROM t;
```

Rules Governing the Data Type of a CASE Expression

Introduction

Because the values in CASE THEN/ELSE clauses can belong to different data types, the following rules apply to the data type of the result of the CASE expression.

Rules for Character Data Types

The rules for character data types are defined in the following table:

IF . . .	THEN . . .
all of the THEN/ELSE expressions belong to character data types	the CASE expression is also a character data type, with the length equal to the maximum of all of the formats of the different data types of the THEN/ELSE expressions. The character data type of the CASE expression is set to the character data type of the first THEN/ELSE character expression. When the character data types of the THEN/ELSE character expressions are different, they are translated to the character data type of the CASE expression.
the THEN/ELSE expression is a KANJI1 constant (for example _Kanji1 '<hex value>'XC)	the character data type of the CASE expression must be KANJI1. ELSE an error is generated.

Examples

For the following examples of CHARACTER data behavior, assume the session 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 result of the following query is in UNICODE:

```
SELECT i, CASE
           WHEN i=1
           THEN 3
           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 10
          END
FROM table_1
ORDER BY 1;
```

Example 2

The result of the following query is a failure because the THEN/ELSE expressions have a KANJI1 constant and the result of the CASE expression is LATIN:

```
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 10
          END
FROM table_1
ORDER BY 1;
```

Example 3

The following query fails because it cannot be converted to 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;
```

Example 4

The result of the following query is in KANJI1:

```
SELECT i, CASE
           WHEN i=1
           THEN column_k
           WHEN i=2
           THEN column_u
           WHEN i=3
           THEN 'abc'
           WHEN i=4
           THEN 8
           WHEN i=5
           THEN column_l
           ELSE 10
          END
FROM table_1
ORDER BY 1;
```

Example 5

The result of the following query is in KANJI1:

```
SELECT i, CASE
           WHEN i=1
           THEN column_k
           WHEN i=2
           THEN column_u
           WHEN i=3
           THEN column_j
           WHEN i=4
           THEN 8
           WHEN i=5
           THEN _Kanji1'4142'XC
           ELSE 10
          END
FROM table_1
ORDER BY 1;
```

Rules for Mixed Data

The rules for mixed data are defined in the following table:

IF the THEN/ELSE clause . . .	THEN . . .
contains a byte, DateTime, or Interval data type	all other expressions must have the same data type.
is composed only of DECIMAL data	the CASE expression returns a DECIMAL result. **Note:** A DECIMAL arithmetic result can have up to 18 digits. A result larger than 18 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 2-3.
is composed only of mixed DECIMAL and INTEGER data	all are implicitly converted to FLOAT and the CASE expression returns a FLOAT result. **Note:** Some inaccuracy is inherent and unavoidable when FLOAT data types are involved. Implicit conversion of DECIMAL and INTEGER values to FLOAT values may result in a loss of precision or produce a number that cannot be represented exactly. To avoid problems associated with conversion to FLOAT, use expressions of the same data type in every THEN/ELSE clause. When a clause uses an expression of one type, and a different type is needed, add an explicit conversion to the query, but carefully consider the properties and limitations of the conversion operation.
contains a FLOAT (approximate numeric) and no character strings	the CASE expression returns a FLOAT result.
is composed only of numeric and character data	the numeric data is converted to character. **Note:** An error is generated if the character data type is GRAPHIC.

Examples

For the following examples of numeric data behavior, assume the following table definition for the CASE examples:

```
CREATE TABLE dec15
(
 column_1 INTEGER,
 column_2 INTEGER,
 column_3 DECIMAL(15,2)
);
```

Example 1

In the following query, the CASE expression fails when column_3 contains the value 1234567890.12 because the result is a DECIMAL value that requires a minimum precision of 23 digits:

```
SELECT SUM (CASE
             WHEN column_2=1
             THEN column_3 * 6.1234500000
             ELSE column_3
            END )
FROM dec15;
```

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 * 6.12345
             ELSE column_3
            END )
FROM dec15;
```

Example 3

In the following query, the CASE expression returns a FLOAT result because its THEN and ELSE clauses contain both INTEGER and DECIMAL values. The result is then cast to DECIMAL using Teradata conversion syntax:

```
SELECT SUM (CASE
             WHEN column_2=1
             THEN column_3 * 6
             ELSE column_3
            END (DECIMAL(15,2)))
FROM dec15;
```

Example 4

In the following query, the CASE expression returns a CHARACTER result because its THEN and ELSE clause contains both FLOAT and CHARACTER values. The result is then implicitly cast to CHARACTER:

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

Example 5

Assume table *dec15* has the following data:

```
column_1     column_2     column_3
--------     --------     --------
       1          101         9.45
       2          101        86.78
       2          102        92.13
       4          102      -188.36
```

The expected answer to the following query is 0.0, the sum of *column_3* values:

```
SELECT SUM (CASE
             WHEN column_2=100
             THEN column_1
             ELSE column_3
            END )
FROM dec15;
```

However, the different data types in the THEN/ELSE clauses result in a conversion to FLOAT, and the actual answer may look like this:

```
Sum(<CASE expression>)
----------------------
-2.84217094304040E-014
```

One way to achieve the expected results is to use the same data types in the THEN/ELSE clauses. For example:

```
SELECT SUM (CASE
             WHEN column_2=100
             THEN (column_1 (DECIMAL(15,2)))
             ELSE column_3
            END )
FROM dec15;
```

Output Format for a CASE Expression

Default Output Format

The result of a CASE expression is displayed using the default output format for the resulting data type. The result of a CASE expression does not apply the explicit output format that may be defined for a column appearing in a THEN/ELSE expression.

For example, the following table definition has two DATE columns that are defined with different output formats:

```
CREATE TABLE duration
(
 i INTEGER,
 start_date DATE FORMAT 'EEEEBMMMBDD,BYYYY',
 end_date DATE FORMAT 'DDBM3BY4'
);
```

Assume the default output 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 output format, not the output 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;
```

Teradata Conversion Syntax

To modify the format of the result of a CASE expression, use the Teradata conversion syntax and specify the FORMAT clause.

Here is an example that changes the output format of the result of the CASE expression in the previous query:

```
SELECT i, ((CASE
             WHEN i=1
             THEN start_date
             WHEN i=2
             THEN end_date
            END) (FORMAT 'DDBM4BYYYY'))
FROM duration
ORDER BY 1;
```

For More Information

For information on ...	See ...
the default data type output formats and the FORMAT output phrase	Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.
using Teradata conversion syntax to convert the format of a data type	"Teradata Conversion Syntax in Explicit Data Type Conversions" on page 13-11.

CASE and Nulls

Introduction

The ANSI SQL-99 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 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.

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 1-25)
- NULLIF expression (see "NULLIF Expression" on page 1-27)

COALESCE Expression

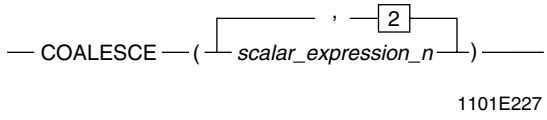
Purpose

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-99-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.

In a default title for an expression, a COALESCE expression appears as:

```
<CASE expression>
```

Using COALESCE in an Expression

Whenever you combine COALESCE with non-comparison operators, such as || or * or +, you must enclose the entire COALESCE function and its operands in parentheses.

For an example, see "Example 2" on page 1-26.

This requirement is not ANSI SQL-99-compliant.

Parentheses are not required when you use COALESCE with comparison operators, Teradata conversion syntax, alias syntax (AS), and so on.

For an example, see "Example 3" on page 1-26.

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 shows how you must enclose COALESCE and its arguments within parentheses when you combine it with a non-comparison operator.

```
SELECT (COALESCE(100,0)) * 100;
```

Example 3

The following example uses COALESCE with a comparison operator. In this case, you do not have to enclose COALESCE and its arguments within parentheses.

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

NULLIF Expression

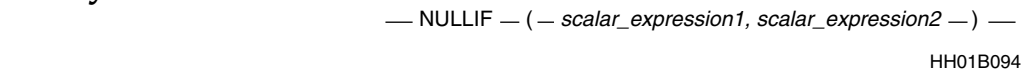
Purpose

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



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" on page 1-27.
scalar_expression_2	the scalar expression to the right of the = in the expanded CASE expression, as shown previously in "Purpose" on page 1-27.

ANSI Compliance

NULLIF is ANSI SQL-99-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*.

In a default title for an expression, a NULLIF expression appears as:

```
<CASE expression>
```

Using NULLIF in an Expression

Whenever you combine NULLIF with non-comparison operators, such as || or * or +, you must enclose the entire NULLIF function and its operands in parentheses.

For an example, see "Example 3" on page 1-28.

This requirement is not ANSI SQL-99-compliant.

Parentheses are not required when you use NULLIF with comparison operators, Teradata conversion syntax, alias syntax (AS), and so on.

Example 1

Column Age has type SMALLINT.

```
NULLIF (Age,0)
```

This is the ANSI-compliant form of the Teradata function NULLIFZERO(Age), and is more versatile.

Example 2

Column Code has type CHARACTER(4) and blanks indicate no value.

```
NULLIF (Code, '    ')
```

Example 3

The following example shows how you must enclose NULLIF and its arguments within parentheses when you combine it with a non-comparison operator.

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

Chapter 2:

Arithmetic Operators and Functions / Trigonometric / Hyperbolic Functions

This chapter describes SQL:

- Arithmetic operators
- Binary arithmetic result data types
- Format for an arithmetic expression
- Arithmetic functions
  - ABS
  - CASE_N
  - EXP
  - LN
  - LOG
  - RANDOM
  - RANGE_N
  - SQRT
  - WIDTH_BUCKET
- Functions designed for doing arithmetic in the presence of zeros and nulls
  - NULLIFZERO
  - ZEROIFNULL
- Trigonometric functions
  - COS
  - SIN
  - TAN
  - ACOS
  - ASIN
  - ATAN
  - ATAN2
- Hyperbolic functions
  - COSH
  - SINH
  - TANH
  - ACOSH
  - ASINH
  - ATANH

Arithmetic Operators

Arithmetic Operators

Teradata supports the following arithmetic operators:

Operator	Function
**	Exponentiate This is a Teradata extension to the ANSI SQL-99 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-99 standard.
+	Add
-	Subtract
+	Unary plus (positive value)
-	Unary minus (negative value)

ANSI Compliance

Except for MOD and **, the arithmetic operators are ANSI SQL-99-compliant.

Binary Arithmetic Result Data Types

Introduction

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:

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
BYTEINT SMALLINT INTEGER	BYTEINT SMALLINT INTEGER	* / + - MOD	INTEGER
	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]
FLOAT	BYTEINT SMALLINT INTEGER DECIMAL(k,j) FLOAT	* / + - MOD	FLOAT
	DATE	* / + - MOD	FLOAT[4]
	CHAR(n) VARCHAR(n)	* / + - MOD	FLOAT[3]

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	+ - *	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]
DATE	BYTEINT SMALLINT INTEGER	+ -	DATE[1]
		* / MOD	INTEGER[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]
CHAR(n) VARCHAR(n)	BYTEINT SMALLINT INTEGER DECIMAL(k,j) FLOAT CHAR(n) VARCHAR(n)	* / + - MOD	FLOAT[3]
	DATE	* / + - MOD	FLOAT[3,4]

Notes

<table>
<tr><th>Number</th><th>Notes</th></tr>
<tr><td>1</td><td>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.</td></tr>
<tr><td>2</td><td>Fractions of decimal values are truncated when added to or subtracted from date values.

Note 1 also applies.</td></tr>
<tr><td>3</td><td>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.</td></tr>
<tr><td>4</td><td>These operations on DATE do not report an error, but results are generally not meaningful.</td></tr>
<tr><td>5</td><td>The difference between two dates is the number of days between those dates.

Note that this is not the numeric difference between the values.</td></tr>
<tr><td>6</td><td>The value of p, the number of digits in the decimal result, depends on:
• The value specified for MaxDecimal in DBSControl.
For more information on DBSControl and MaxDecimal, see “DBS Control Utility” in Teradata RDBMS Utilities.
• 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.
<table>
<tr><th>IF MaxDecimal is …</th><th>AND the number of digits in the decimal operand is …</th><th>THEN p is …</th></tr>
<tr><td rowspan="2">0 or 15</td><td><= 15</td><td>15</td></tr>
<tr><td>> 15</td><td>18</td></tr>
<tr><td>18</td><td>any value</td><td>18</td></tr>
</table>
</td></tr>
<tr><td>7</td><td>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.
<table>
<tr><th>IF MaxDecimal is …</th><th>AND …</th><th>THEN p is …</th></tr>
<tr><td rowspan="2">0 or 15</td><td>m and k <= 15</td><td>15</td></tr>
<tr><td>m or k > 15</td><td>18</td></tr>
<tr><td>18</td><td>m and k = any value</td><td>18</td></tr>
</table>
</td></tr>
</table>

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 flag in DBSControl affects rounding, see "Decimal/Numeric Data Types" in *Teradata RDBMS SQL Reference, Volume 3* and "DBS Control Utility" in *Teradata RDBMS Utilities*.

Integer Division and Truncation

Integer division yields whole results, truncated toward zero.

Structure of Arithmetic Expressions

Order of Evaluation

In all cases, SQL performs expressions enclosed in parentheses first. The following table lists the order in which arithmetic expressions are performed when no parentheses are present. Operators of the same precedence are evaluated from left to right.

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)

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:

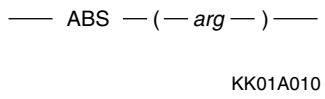
- ABS
- CASE_N
- EXP
- LN
- LOG
- NULLIFZERO
- RANDOM
- RANGE_N
- SQRT
- ZEROIFNULL

ABS

Purpose

Computes the absolute value of an argument.

Syntax



KK01A010

where:

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

ANSI Compliance

ABS is a Teradata extension to the ANSI SQL-99 standard.

Result Type and Attributes

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

Data Type	Format	Title
Same data type as *arg*[a]	IF the operand is ... / THEN the format is the default format for ... numeric / the resulting data type. character / FLOAT.	ABS(*arg*)

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

For information on the format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument is not numeric, it is converted to a numeric value, based on implicit type conversion rules. If *arg* is a character string, it is converted to a numeric value of the FLOAT data type. If the argument cannot be converted, an error is reported.

ABS cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Examples

Representative ABS arithmetic function expressions and the results are:

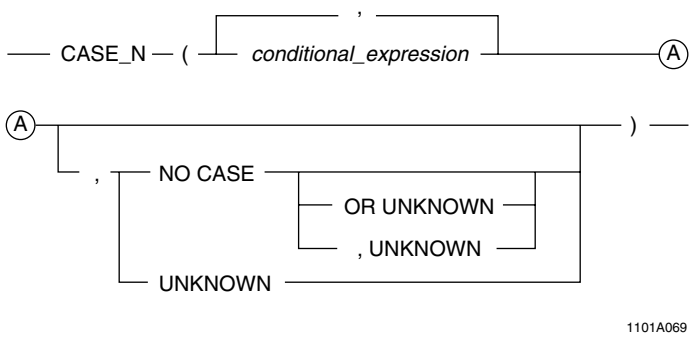
Expression	Result
ABS(-12)	12
ABS('23')	2.30000000000000E+001

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



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. A conditional expression that evaluates to NULL is considered to be 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-99 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.

IF ...	THEN ...
a *conditional_expression* evaluates to TRUE, and all prior conditions evaluate to FALSE	CASE_N returns the position of the *conditional_expression*.
a *conditional_expression* evaluates to UNKNOWN, and all prior conditions evaluate to FALSE	<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>
every *conditional_expression* evaluates to FALSE	<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>

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 the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Using CASE_N to Define Partitioned Primary Indexes

The primary index for a table controls the distribution of the data for that table 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, you specify the PRIMARY INDEX phrase in the CREATE TABLE 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 a partitioning expression that determines 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 2-26.

Restrictions

If CASE_N is used in a partitioning expression in a CREATE TABLE or ALTER TABLE statement, it:

- Must not involve character or graphic comparisons
- Can specify a maximum of 65533 conditions

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

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
```

For More Information

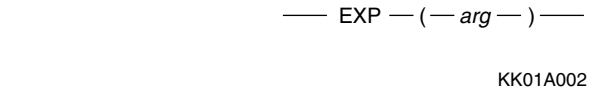
For information on ...	See ...
PPI properties and performance considerations	Chapter 9: "Primary Indexes" in *Teradata RDBMS Database Design*.
PPI considerations and capacity planning	Chapter 14: "Database-Level Considerations" in *Teradata RDBMS Database Design*.
the specification of a PPI for a table	CREATE TABLE in Chapter 1: "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.
the modification of the partitioning of the primary index for a table	ALTER TABLE in Chapter 1: "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.

EXP

Purpose

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

Syntax



KK01A002

where:

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

ANSI Compliance

EXP is a Teradata extension to the ANSI SQL-99 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 the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument is not FLOAT, it is converted to FLOAT, based on implicit type conversion rules. If the argument cannot be converted, an error is reported.

EXP cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Executing EXP may sometimes result in a numeric overflow error.

Examples

Representative EXP arithmetic function expressions and the results are:

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

LN

Purpose

Computes the natural logarithm of the argument.

Syntax



KK01A003

where:

Syntax element …	Specifies …
arg	a non-zero, positive numeric argument.

ANSI Compliance

LN is a Teradata extension to the ANSI SQL-99 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 the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported.

LN cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Examples

Representative LN arithmetic function expressions and the results are:

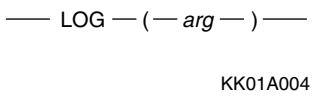
Expression	Result
LN(2.71828182845905)	1.00000000000000E+000
LN(0)	Error

LOG

Purpose

Computes the base 10 logarithm of an argument.

Syntax



KK01A004

where:

Syntax element …	Specifies …
arg	a non-zero, positive numeric argument.

ANSI Compliance

LOG is a Teradata extension to the ANSI SQL-99 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 the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported.

LOG cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Examples

Representative LOG arithmetic function expressions and the results are:

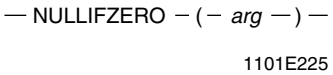
Expression	Result
LOG(50)	1.69897000433602E+000
LOG(100)	2.00000000000000E+000

NULLIFZERO

Purpose

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

Syntax



where:

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

ANSI Compliance

NULLIFZERO is a Teradata extension to the ANSI SQL-99 standard.

The ANSI form of this function is the CASE shorthand expression NULLIF. For more information, see "NULLIF Expression" on page 1-27.

Result Type and Attributes

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

Data Type	Format	Title
Same data type as *arg*[a]	IF the operand is ... / THEN the format is the ... numeric / same format as *arg*. character / default format for FLOAT.	NULLIFZERO(*arg*)

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

For information on the format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

IF the value of arg is ...	THEN NULLIFZERO returns ...
nonzero	the value of the numeric argument
null or zero	NULL

If the argument is not numeric, it is converted to a numeric value, based on implicit type conversion rules. If *arg* is a character string, it is converted to a numeric value of FLOAT data type. If the argument cannot be converted, an error is reported.

NULLIFZERO cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

For More Information

For additional expressions involving checks for nulls, see:

- "COALESCE Expression" on page 1-25
- "NULLIF Expression" on page 1-27
- "ZEROIFNULL" on page 2-37

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 '?'.

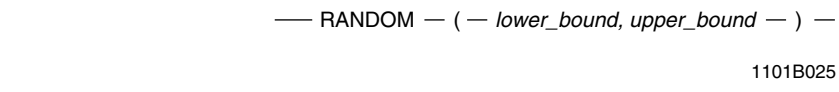
```
SELECT empno, NULLIFZERO(hcap)
FROM employee
WHERE empno = 10019 ;
```

RANDOM

Purpose

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

Syntax



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-99 standard.

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 used in the expression list of an INSERT statement to create a primary index column value, for example:

  ```
  INSERT t1 (RANDOM(1,10),...)
  ```

  RANDOM produces inconsistent results in this case.
- RANDOM cannot be nested inside aggregate or ordered analytical functions.

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.

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}

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 the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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.

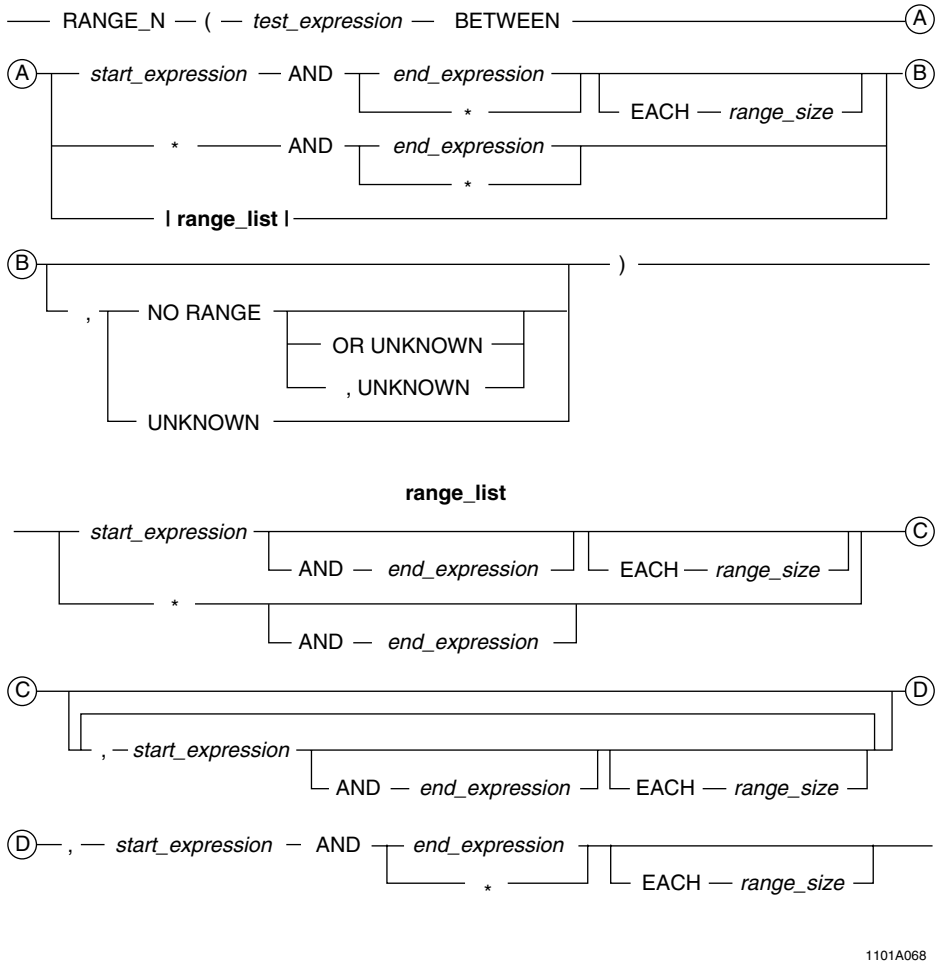
Store_ID	Product_ID	Sales	RANDOM(1,3)
1003	C	20000	1
1002	C	35000	2
1001	C	60000	2
1002	D	50000	3
1003	D	50000	2
1001	D	35000	3
1001	A	100000	2
1002	A	40000	1
1001	E	30000	2

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.
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. 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. The value of *range_size* must be greater than zero. 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.

Syntax element …	Specifies …
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-99 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.

IF …	THEN …
the result of *test_expression* is within a range	RANGE_N returns the position of the range.
test_expression is outside all the ranges in the list	**IF …** / **THEN RANGE_N returns …** NO RANGE or NO RANGE OR UNKNOWN is specified → *n* + 1. neither NO RANGE nor NO RANGE OR UNKNOWN is specified → NULL.

<table>
<tr><th>IF ...</th><th>THEN ...</th></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>

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 the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Using RANGE_N to Define Partitioned Primary Indexes

The primary index for a table controls the distribution of the data for that table 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, you specify the PRIMARY INDEX phrase in the CREATE TABLE 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 a partitioning expression that determines 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 2-12.

Restrictions

If RANGE_N is used in a partitioning expression in a CREATE TABLE or ALTER TABLE statement, it:

- Must not use character or graphic comparisons.
- Can specify a maximum of 65533 ranges

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 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 3

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 twelve 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
```

For More Information

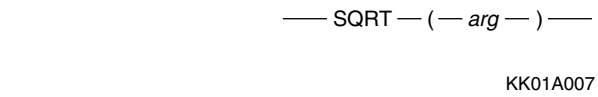
For information on ...	See ...
PPI properties and performance considerations	Chapter 9: "Primary Indexes" in *Teradata RDBMS Database Design*.
PPI considerations and capacity planning	Chapter 14: "Database-Level Considerations" in *Teradata RDBMS Database Design*.
the specification of a PPI for a table	CREATE TABLE in Chapter 1: "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.
the modification of the partitioning of the primary index for a table	ALTER TABLE in Chapter 1: "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*.

SQRT

Purpose

Computes the square root of an argument.

Syntax



KK01A007

where:

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

ANSI Compliance

SQRT is a Teradata extension to the ANSI SQL-99 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 the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported.

SQRT cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Examples

Representative SQRT arithmetic function expressions and the results are:

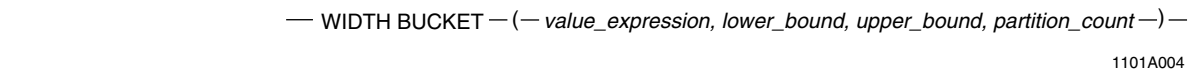
Expression	Result
SQRT(2)	1.41421356237309E+000
SQRT(-2)	Error

WIDTH_BUCKET

Purpose

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

Syntax



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-99 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 the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

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. WIDTH_BUCKET also accepts character strings that represent

numeric values, and converts them to the appropriate numeric type. If an argument cannot be converted, an error is reported.

WIDTH_BUCKET cannot be applied to arguments of type TIME, TIMESTAMP, INTERVAL, BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

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.

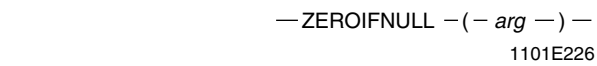
salary	Width_bucket(salary,70000,200000,4)	Count(salary)
50000	0	1
70000	1	3
120000	2	1
150000	3	1
199999	4	1
220000	5	1

ZEROIFNULL

Purpose

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

Syntax



where:

Syntax element …	Specifies …
arg	a numeric argument.

ANSI Compliance

ZEROIFNULL is a Teradata extension to the ANSI SQL-99 standard.

Result Type and Attributes

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

Data Type	Format	Title
Same data type as *arg*[a]	**IF the operand is …** / **THEN the format is the …**: numeric → same format as *arg*.; character → default format for FLOAT.	ZEROIFNULL(*arg*)

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

For information on the format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

IF the value of arg is …	THEN ZEROIFNULL returns …
not null	the value of the numeric argument
null or zero	zero

If the argument is not numeric, it is converted to a numeric value according to implicit type conversion rules. If *arg* is a character string, it is converted to a numeric value of FLOAT data type. If the argument cannot be converted, an error is reported.

ZEROIFNULL cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

For More Information

For additional expressions involving checks for nulls, see:

- "COALESCE Expression" on page 1-25
- "NULLIF Expression" on page 1-27
- "NULLIFZERO" on page 2-21

Example

In this example, you can test the Salary column for null.

```
SELECT empno, ZEROIFNULL(salary)
FROM employee ;
```

A non-zero value is returned for each employee number, indicating that no nulls exist in the Salary column.

Trigonometric Functions

The next sections describe trigonometric functions. Teradata supports the standard set of trigonometric and inverse trigonometric functions:

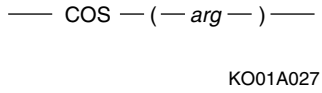
- COS
- SIN
- TAN
- ACOS
- ASIN
- ATAN
- ATAN2

COS

Purpose

Returns the cosine of an argument.

Syntax



KO01A027

where:

Syntax element …	Specifies …
arg	any valid number expression that expresses an angle in radians

ANSI Compliance

COS is a Teradata extension to the ANSI SQL-99 standard.

Definition

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.

Result Type and Attributes

Here are the default attributes for the result of COS(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Cos(arg)

For information on the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The result of COS lies in the range -1 to 1, inclusive.

If the argument is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported.

COS cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative COS function expression and result:

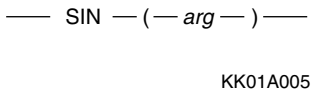
Expression	Result
COS(5-4)	5.40302305868140E -001

SIN

Purpose

Returns the sine of an argument.

Syntax



KK01A005

where:

Syntax element …	Specifies …
arg	any valid number expression that expresses an angle in radians

ANSI Compliance

SIN is a Teradata extension to the ANSI SQL-99 standard.

Definition

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.

Result Type and Attributes

Here are the default attributes for the result of SIN(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Sin(arg)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The result of SIN lies in the range -1 to 1, inclusive.

If the argument is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported.

SIN cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative SIN function expression and result:

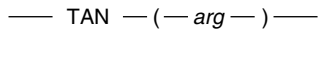
Expression	Result
SIN(LOG(0.5))	-2.96504042171437E -001

TAN

Purpose

Returns the tangent of an argument.

Syntax



KK01A008

where:

Syntax element ...	Specifies ...
arg	any valid number expression that expresses an angle in radians

ANSI Compliance

TAN is a Teradata extension to the ANSI SQL-99 standard.

Definition

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 attributes for the result of TAN(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Tan(arg)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported.

TAN cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative TAN function expression and result:

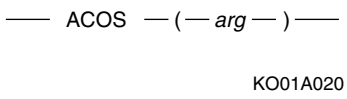
Expression	Result
TAN(ABS(-3))	-1.42546543074278E -001

ACOS

Purpose

Returns the arccosine of an argument.

Syntax



where:

Syntax element ...	Specifies ...
arg	the cosine of the returned angle. The values of *arg* must be between -1 and 1, inclusive.

ANSI Compliance

ACOS is a Teradata extension to the ANSI SQL-99 standard.

Definition

The arccosine is the angle whose cosine is the argument.

Result Type and Attributes

Here are the default attributes for the result of ACOS(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	ArcCos(arg)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The returned angle is in the range 0 to π radians, inclusive.

If the argument 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.

ACOS cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative ACOS function expression and result:

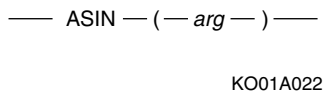
Expression	Result
ACOS(-0.5)	2.09439510239320E 000

ASIN

Purpose

Returns the arcsine of an argument.

Syntax



KO01A022

where:

Syntax element …	Specifies …
arg	the sine of the returned angle. The values of *arg* must be between -1 and 1, inclusive.

ANSI Compliance

ASIN is a Teradata extension to the ANSI SQL-99 standard.

Definition

The arcsine is the angle whose sine is the argument.

Result Type and Attributes

Here are the default attributes for the result of ASIN(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	ArcSin(arg)

For information on the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The returned angle is in the range $-\pi/2$ to $\pi/2$ radians, inclusive.

If the argument 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.

ASIN cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative ASIN function expression and result:

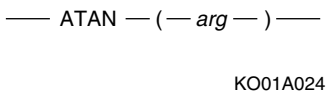
Expression	Result
ASIN(1)	1.57079632679490E 000

ATAN

Purpose

Returns the arctangent of an argument.

Syntax



KO01A024

where:

Syntax element ...	Specifies ...
arg	the tangent of the returned angle.

ANSI Compliance

ATAN is a Teradata extension to the ANSI SQL-99 standard.

Definition

The arctangent is the angle whose tangent is the argument.

Result Type and Attributes

Here are the default attributes for the result of ATAN(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	ArcTan(arg)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The returned angle is in the range $-\pi/2$ to $\pi/2$ radians, inclusive.

If the argument 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.

ATAN cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative ATAN function expression and result:

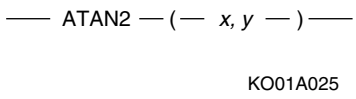
Expression	Result
ATAN(1+2)	1.24904577239825E 000

ATAN2

Purpose

Returns the arctangent of the specified x and y coordinates.

Syntax



KO01A025

where:

Syntax element ...	Specifies ...
x	the x-coordinate of a point to use in the calculation.
y	the y-coordinate of a point to use in the calculation.

ANSI Compliance

ATAN2 is a Teradata extension to the ANSI SQL-99 standard.

Definition

The arctangent is the angle whose tangent is the argument: the angle from the x-axis to a line containing the origin (0,0) and a point with coordinates (x,y).

Result Type and Attributes

Here are the default attributes for the result of ATAN2(*x,y*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Atan2(x,y)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The returned angle is 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.

Note: 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.

Arguments that are not FLOAT are converted to FLOAT values according to implicit type conversion rules. If an argument cannot be converted, an error is reported.

If both x and y are 0, an error is returned.

ATAN2 cannot be applied to arguments of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative ATAN2 function expression and result:

Expression	Result
ATAN2(1,1)	7.85398163397448E -001

Hyperbolic Functions

The next sections describe the following hyperbolic functions:

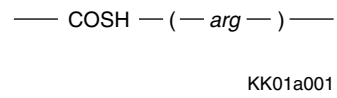
- COSH
- SINH
- TANH
- ACOSH
- ASINH
- ATANH

COSH

Purpose

Returns the hyperbolic cosine of an argument.

Syntax



KK01a001

where:

Syntax element …	Specifies …
arg	any real number.

ANSI Compliance

COSH is a Teradata extension to the ANSI SQL-99 standard.

Result Type and Attributes

Here are the default attributes for the result of COSH(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Hyperbolic Cos(arg)

For information on the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument 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.

COSH cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative COSH function expression and result:

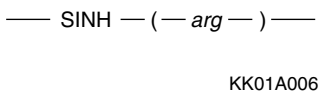
Expression	Result
COSH(EXP(1))	7.61012513866229E 000

SINH

Purpose

Returns the hyperbolic sine of an argument.

Syntax



KK01A006

where:

Syntax element ...	Specifies ...
arg	any real number.

ANSI Compliance

SINH is a Teradata extension to the ANSI SQL-99 standard.

Result Type and Attributes

Here are the default attributes for the result of SINH(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Hyperbolic Sin(arg)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument 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.

SINH cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative SINH function expression and result:

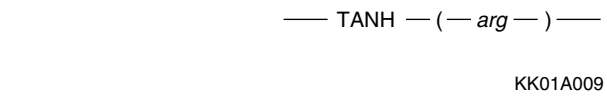
Expression	Result
SINH(1)	1.17520119364380E 000

TANH

Purpose

Returns the hyperbolic tangent of an argument.

Syntax



where:

Syntax element ...	Specifies ...
arg	any real number.

ANSI Compliance

TANH is a Teradata extension to the ANSI SQL-99 standard.

Result Type and Attributes

Here are the default attributes for the result of TANH(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Hyperbolic Tan(arg)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

If the argument 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.

TANH cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative TANH function expression and result:

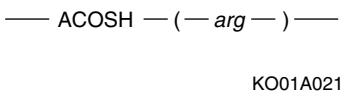
Expression	Result
TANH(0)	0.00000000000000E 000

ACOSH

Purpose

Returns the inverse hyperbolic cosine of an argument.

Syntax



KO01A021

where:

Syntax element …	Specifies …
arg	any real number equal to, or greater than, 1.

ANSI Compliance

ACOSH is a Teradata extension to the ANSI SQL-99 standard.

Result Type and Attributes

Here are the default attributes for the result of ACOSH(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Hyperbolic ArcCos(arg)

For information on the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The inverse hyperbolic cosine is the value whose hyperbolic cosine is a number so that:

```
acosh(cosh(arg)) = arg
```

If the argument 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.

ACOSH cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative ACOSH function expression and result:

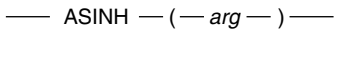
Expression	Result
ACOSH(3)	1.76274717403909E 000

ASINH

Purpose

Returns the inverse hyperbolic sine of an argument.

Syntax



KO01A023

where:

Syntax element ...	Specifies ...
arg	any real number.

ANSI Compliance

ASINH is a Teradata extension to the ANSI SQL-99 standard.

Result Type and Attributes

Here are the default attributes for the result of ASINH(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Hyperbolic ArcSin(arg)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The inverse hyperbolic sine is the value whose hyperbolic sine is a number so that:

```
asinh(sinh(arg)) = arg
```

If the argument 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.

ASINH cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative ASINH function expression and result:

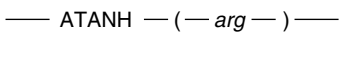
Expression	Result
ASINH(LOG(0.1))	-8.81373587019543E -001

ATANH

Purpose

Returns the inverse hyperbolic tangent of an argument.

Syntax



KO01A026

where:

Syntax element ...	Specifies ...
arg	any real number between 1 and -1, excluding 1 and -1.

ANSI Compliance

ATANH is a Teradata extension to the ANSI SQL-99 standard.

Result Type and Attributes

Here are the default attributes for the result of ATANH(*arg*):

Data Type	Format	Title
FLOAT	Default format for FLOAT	Hyperbolic ArcTan(arg)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The inverse hyperbolic tangent is the value whose hyperbolic tangent is a number so that:

```
atanh(tanh(arg)) = arg
```

If the argument 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.

ATANH cannot be applied to an argument of type BYTE, VARBYTE, or CHARACTER or VARCHAR if the server character set is GRAPHIC.

Example

The following is a representative ATANH function expression and result:

Expression	Result
ATANH(LN(0.5))	-8.53988047997524E -001

Chapter 3:

Comparison Operators

This chapter describes SQL comparison operators, including:

- General comparison rules
- Comparison evaluation by data type
- Time zone comparisons
- Comparisons for the KANJI1 character data type
- Case sensitivity in comparisons
- OVERLAPS operator

Comparison Operators

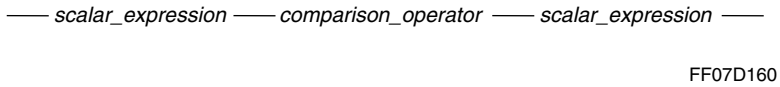
Purpose

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, and REPEAT statements in stored procedures
- WHEN clauses in searched CASE expressions
- WHERE, ON, and HAVING clauses to qualify or disqualify rows in a SELECT statement

Syntax



FF07D160

where:

Syntax element …	Specifies …
scalar_expression	an expression to be evaluated in comparison with a second *scalar_expression*.
comparison_operator	the type of comparison to be evaluated for truth. The supported comparisons are defined in the table following this one.

ANSI Compliance

The following comparison operators are ANSI SQL-99-compliant.

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

The following comparison operators are Teradata extensions to the ANSI SQL-99 standard. Their use is deprecated.

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

Supported Comparison Operators

The Teradata RDBMS uses the comparison operators described in the following table.

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.

The results of character comparisons depends on the collation sequence of the character set in use.

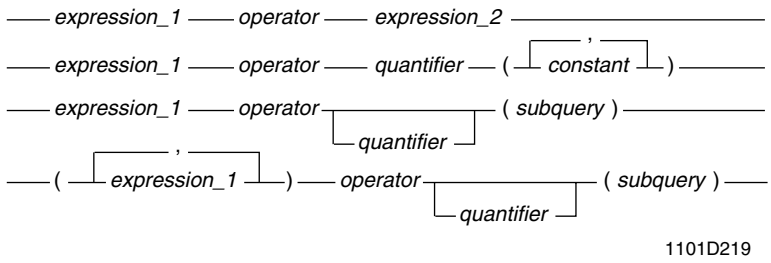
Further Information on Predicates

FOR more information on ...	SEE ...
other logical predicates, including: • [NOT] BETWEEN … AND … • OVERLAPS • LIKE • [NOT] IN • ALL • ANY • SOME • [NOT] EXISTS	Chapter 9: “Logical Predicates”.
using predicates in conditional expressions in searched CASE expressions	Chapter 1: “CASE Expressions”.
using predicates in conditional expressions in WHERE, ON, or HAVING clauses in SELECT statements	Chapter 1: “The SELECT Statement” in *Teradata RDBMS SQL Reference, Volume 6*.
using predicates in conditional expressions in IF, WHILE, or REPEAT statements in stored procedures	Chapter 9: “SQL Stored Procedures” in *Teradata RDBMS SQL Reference, Volume 6*.

Comparison Operators in Logical Expressions

Syntax

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



1101D219

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 See "ANY/ALL/SOME" on page 9-11.
constant	one or more constant values. The comparison operation may compare an expression against a list of explicit constants. A constant may be any of the following: • Defined value • Macro parameter • Built-in value such as TIME, DATE, or USER The data types of *expression* and *constant* must be compatible.
subquery	an SQL SELECT statement.

Results

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

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 rth components of *expression_1* and *expression_2* are $expression\_1_r$ and $expression\_2_r$.

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

The data types of *expression_1* and *expression_2* must be compatible.

For an explanation of the symbols used in this table, see "Predicate Calculus Notation Used in This Book" on page A-8.

This comparison . . .	Is TRUE iff . . .
expression_1 = *expression_2*	$\forall$ *r*, $expression\_1_r = expression\_2_r$ is TRUE.
expression_1 <> *expression_2*	$\exists$ δ such that $expression\_1_\delta <> expression\_2_\delta$ is TRUE.
expression_1 < *expression_2*	$\exists$ δ 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*	$\exists$ δ 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*	*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.

Methods of 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:

<table>
<tr><th>This data type ...</th><th>Is evaluated in this way ...</th></tr>
<tr><td>NUMERIC</td><td>Algebraically, with negatives considered to be smaller irrespective of their absolute value.</td></tr>
<tr><td>CHARACTER</td><td>Character-by-character from left to right. Exact comparisons depend on the collation sequence assigned.
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><tr><td>two strings of different length are compared</td><td>the shorter string is padded to the right with spaces to make the lengths equal prior to making the comparison.</td></tr></table></td></tr>
<tr><td>DateTime</td><td>Chronologically.</td></tr>
<tr><td>INTERVAL</td><td>According to sign and magnitude.</td></tr>
<tr><td>• BYTE
• VARBYTE</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>
</table>

Rules for Comparisons

Comparison of Different Data Types

Expression operands must be of the same data type before a comparison operation can occur.

If numeric values of different underlying representations are being compared, they are implicitly converted to the same representation before the comparison occurs.

For example, numeric values of decimal and integer representations are converted and compared as follows.

Expression_1	Expression_2	Data Compared As
INTEGER	DECIMAL(m,n)	DECIMAL(18, n)
DECIMAL(m,n)	DECIMAL(k,j)	DECIMAL(18,max(n,j))

Thus, if an INTEGER value is compared to a DECIMAL(5,2) value, the INTEGER value is first converted to DECIMAL(18,2).

If operand data types differ, then an implicit conversion is performed, as shown in the following table.

	NUMERIC	CHARACTER	BYTE	DATE
NUMERIC	NUMERIC	NUMERIC	error	NUMERIC
DATE	NUMERIC	DATE	error	DATE
CHARACTER	NUMERIC[1]	CHARACTER	error	DATE[1]
BYTE	error	error	BYTE	error

[1]Returns an error for character data with GRAPHIC type.

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

Implicit conversions are Teradata extensions to the ANSI SQL-99 standard.

Character String Comparisons

Comparison rules for string comparisons appear below.

- If character strings of unequal length are being compared, the shorter of the two is padded on the right with blanks before the comparison occurs.

  If two byte strings of unequal length are compared, the shorter string is padded on the right with binary zeros.

- When comparing character strings, data characters must have the same character data type. If they do not, then the system translates them using the implicit translation rules described in "Implicit Character-to-Character Translation" on page 13-32.

In this session mode ...	Strings are compared in this way ...
ANSI	case specific without regard to how the columns are specified in the table definition. To make non-case specific comparisons, you must explicitly specify the (NOT CASESPECIFIC) option to the fields being compared.
Teradata	not case specific unless otherwise specified with the (CASESPECIFIC) option.

  The values used depend on the collation in effect for the session.
- 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.

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:

```
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).

Time Zone Comparisons

Introduction

Comparisons between time zones are performed chronologically, using the same time zone to make the comparisons.

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
```

Comparisons for the KANJI1 Character Data Type

Introduction

For KANJI1 characters, the Teradata RDBMS makes comparisons as described in the following sections:

Equality Comparison

The handling of an equality comparison depends on whether the expressions are CHARACTER or GRAPHIC strings, as explained below.

- 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 character data types, then they are converted to the same type (see "Implicit Character-to-Character Translation" on page 13-32).
  - 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.
  -

In this mode ...	Comparisons are ...
ANSI	CASESPECIFIC by default without regard to how the columns are specified in the table definition.
Teradata	NOT CASESPECIFIC by default. 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.

  - Trailing pad characters are ignored.

Nonequality Comparison: Collation Sequences

Nonequality comparisons involve the collation in effect for the session. Five collations are available: EBCDIC, ASCII, MULTINATIONAL, CHARSET_COLL, and 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 information on collation sequences, see the chapter "International Character Support" in *Teradata RDBMS SQL Reference, Volume 1.*

Comparing Nonequality Character Strings

Nonequality comparisons are handled as explained below.

<table>
<tr><th>Stage</th><th>Process</th></tr>
<tr><td>1</td><td>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.</td></tr>
<tr><td>2</td><td>Session mode is identified.

<table><tr><th>In this mode ...</th><th>String comparisons are ...</th></tr><tr><td>ANSI</td><td>CASESPECIFIC by default without regard to how the columns are specified in the table definition.</td></tr><tr><td>Teradata</td><td>NOT CASESPECIFIC by default.

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.</td></tr></table></td></tr>
<tr><td>3</td><td>Characters identified as single byte characters under the current character set are converted according to the collation sequence in effect for the session.</td></tr>
<tr><td>4</td><td>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.</td></tr>
<tr><td>5</td><td>The prepared strings are compared and any trailing pad characters are ignored.</td></tr>
</table>

Case Sensitivity in Comparisons

Introduction

All character data accessed in the performance of a Teradata SQL statement has an attribute of CASESPECIFIC or NOT CASESPECIFIC, either by default or by explicit designation. This is not an ANSI SQL-99-compatible attribute—ANSI does *all* character comparisons as the equivalent of CASESPECIFIC.

Table columns carry the attribute assigned at the time the column was defined unless a non-ANSI modification phrase is used in their access.

You can declare a session to operate in ANSI or Teradata mode when you log on or you can use the .SET SESSION command in BTEQ or the equivalent session control command in other applications.

IF a session is in this mode . . .	THEN the default for character data in a CREATE or ALTER statement is . . .
ANSI	CASESPECIFIC
Teradata	NOT CASESPECIFIC The exception is CHARACTER data of type GRAPHIC, which is always CASESPECIFIC. This attribute persists unless the column definition is modified by a later ALTER statement.

Column case specificity only has meaning in Teradata mode. If you create a table having columns that are specified to be NOT CASESPECIFIC, those columns are treated with case specificity in any comparison made on them in ANSI mode. You can only make non-case specific comparisons in ANSI mode if you explicitly declare the relevant column names and values in the query to be (NOT CASESPECIFIC).

Defaults

The defaults for character literal strings in queries, view and macro definitions, CHECK constraints, and data accessed by a USING phrase are determined at session logon unless you modify them explicitly in your query text. The default case comparison attribute exists only for the life of the session.

Domain Checking

Expression operands must be of the same data type before a comparison can occur. If the operands data types are different, one of them must be converted.

If you find that your applications are comparing data from different domains, you need to review your database design to ensure that column data are properly typed.

Defaults for Non-English Collation

On a European or a Japanese language site, 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 attribute of your site, the availability of diacritical or Japanese character sets, and your default collation sequence are under the control of your database administrator.

For collation details, see:

- "SET SESSION COLLATION" in Chapter 1: "SQL Data Definition Language Statement Syntax" in *Teradata RDBMS SQL Reference, Volume 4*
- *Teradata RDBMS International Character Set Support*
- "ORDER BY Clause" in Chapter 1: "The SELECT Statement" in *Teradata RDBMS SQL Reference, Volume 6*

Comparing Character Strings

If character strings of unequal length are compared on a European Teradata RDBMS, the shorter field is padded with blanks on the right prior to the comparison. If Japanese character strings of unequal length are compared, the shorter string is padded with one or more pad characters.

For NON CASESPECIFIC 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.

Comparison Rules

- Because of the long persistence of the CASESPECIFIC attribute for columns, and the shorter persistence of attributes carried in the source text of queries, the same statement can have different results for a user logged on in ANSI versus Teradata mode.
-

IF . . .	THEN the comparison is . . .
either argument is CASESPECIFIC	case specific.
both arguments are NOT CASESPECIFIC	case blind for simple Latin letters.

- Default comparisons depend on the mode of the session and on the comparison option specified for the data.
- If the comparison specification is NON CASESPECIFIC, lowercase and uppercase Cyrillic, Greek, and fullwidth ASCII are equivalent.

- To distinguish lowercase and uppercase Cyrillic, Greek, and fullwidth ASCII letters you must explicitly declare CASESPECIFIC comparison.
- A comparison qualified as NOT CASESPECIFIC always considers lowercase and uppercase Cyrillic, Greek and full-width ASCII letters to be equivalent.
- These options work for the KanjiSJIS character data type as if the data were first converted to the Unicode type and then the options applied.

Example

Consider the following query.

The behavior of its comparison `WHERE firstname = 'George'` under different case specificity attributes and session modes is described in the table that follows.

```
SELECT *
FROM STUDENTS
WHERE firstname = 'George';
```

IF column firstname is . . .	AND the session is in this mode . . .	THEN the match succeeds for rows with firstname containing . . .
CASESPECIFIC	either	'George'
NOT CASESPECIFIC	ANSI	'George'
NOT CASESPECIFIC	Teradata	'george' 'GEORGE' 'George' and any other combination of cases spelling out the name George.

If you want to ensure case blind comparisons and have an ANSI-compliant application, the recommended approach is to define FIRSTNAME with ANSI default (CASESPECIFIC).

Comparisons on GRAPHIC character data are always CASESPECIFIC regardless of the session mode.

Chapter 4:

Set Operators

This chapter describes SQL set operators. Topics include:

- Overview of set operators
- Rules for Set Operators
- Precedence
- Retaining duplicate rows using the ALL option
- Attributes of a set result
- Set operators with derived tables
- Set operators in subqueries
- Set operators in INSERT...SELECT statements
- Set operators in view definitions
- Connecting queries by set operators
- INTERSECT operator
- MINUS/EXCEPT operator
- UNION operator

Overview of Set Operators

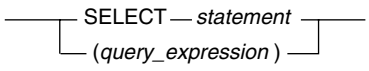
Introduction

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.

What is a Query Expression?

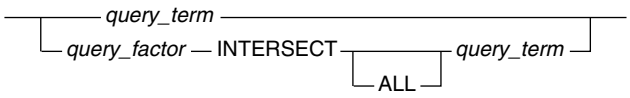
A query expression is a set of queries combined by the set operators INTERSECT, MINUS/EXCEPT, and UNION.

Syntax for *query_term*



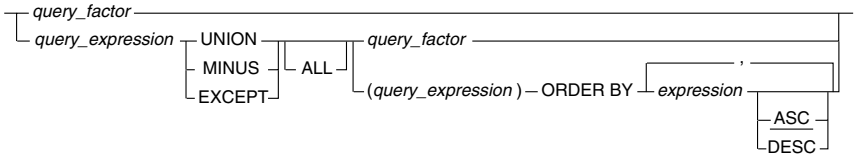
FF07D178

Syntax for *query_factor*



HH01A061

Syntax for *query_expression*



FF07D179

where:

Syntax Element . . .	Specifies . . .
query_term	
SELECT *statement*	a SELECT statement. For details, see Chapter 1: "The SELECT Statement" of *Teradata RDBMS SQL Reference, Volume 6*.
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 "ORDER BY Clause" in Chapter 1: "The SELECT Statement" of *Teradata RDBMS SQL Reference, Volume 6*.
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-99-compliant.

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

The Teradata Set Operators

The set operators are listed and defined in the following table.

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.

Rules for Set Operators

The following rules apply to the use of set operators:

- By default, duplicate rows are not returned.

  To permit duplicate rows to be returned, specify the ALL option.
- You can use set operators within the following operations:
  - Simple queries
  - Derived tables
  - 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.
- A query expression can include only one ORDER BY specification, at the end.

  A query expression including the UNION operator, for example, can contain another query expression as the left operand of the UNION.

Precedence

The precedence for processing set operators is shown in the following table:

Precedence	Set Operator
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 Optimizer instructs the database manager to perform the specified operations in the order described in the following table.

Stage	Process
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

Precedence can be overridden by using parentheses. The operations in the parentheses are performed first.

For example, when the following form is used:

```
(
(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 table explains the precedence of operators for this example.

Stage	Process
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

Introduction

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 ;
```

Set Operators in Subqueries

Introduction

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

Introduction

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

Introduction

Set operators are permitted within view definitions.

The following example set provides 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_1 AS

   SELECT x1,y1
   FROM table_1
   UNION
   SELECT x2,y2
   FROM table_2;
```

Example 3

```
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 4

```
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;
```

Connecting Queries by Set Operators

Restrictions

Certain restrictions apply to SELECT statements connected by set operators that might not apply elsewhere. Therefore, in order for a group of queries to be connected by set operators, the queries must satisfy the following rules:

- All SELECT statements in a multiple query 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 clauses cannot be used in the SELECT statements.
- GROUP BY clauses are allowed in individual SELECT statements but apply only to that SELECT statement and not to the result set.
- ORDER BY clauses are allowed only in the last SELECT statement 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.

  GROUP BY and ORDER BY operations can be used within views with set operators. 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
);
```

- Each SELECT statement must identify the table that the data is to come from even if all SELECT statements reference the same table.
- The data types of corresponding items in each SELECT statement must be *compatible*.

  For example, if the first field in the first SELECT statement is character data type, then the first field in each succeeding SELECT statement must be character data type.

  If a field in the first SELECT statement is numeric, then the corresponding field in succeeding SELECT statements must also be numeric.

  It is not necessary that the numeric data types be the same; that is, one field can be defined as INTEGER and the corresponding field can be defined as SMALLINT.

  All succeeding corresponding numeric fields will be converted to the numeric data type of the first SELECT statement.

- For view definitions, the following three limitations are placed on the use of set operators:
  - 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.
    ```

  - 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.
    ```

  - 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 causes the following error message:

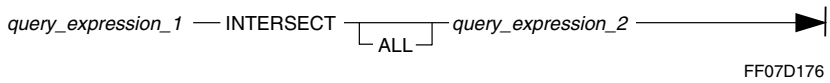
    ```
    ***Failure 3823 VIEW 'v' may not be used for Help
    Index/Constraint/Statistics, Update, Delete or Insert.
    ```

INTERSECT Operator

Purpose

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

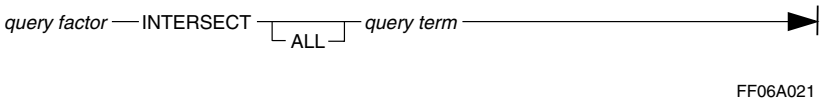
Syntax



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 4-2.
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 4-2.

Semantics



ANSI Compliance

INTERSECT is ANSI SQL-99-compliant.

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

Rules for INTERSECT

The following rules apply to the use of INTERSECT.

- Besides simple queries, INTERSECT can be used within the following operations:
  - Derived tables
  - 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.

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 4-32.

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), the following request could be used:

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

The result of this request is:

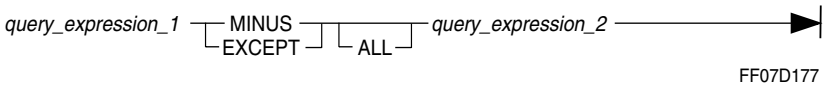
```
SuppNo
   101
```

MINUS/EXCEPT Operator

Purpose

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

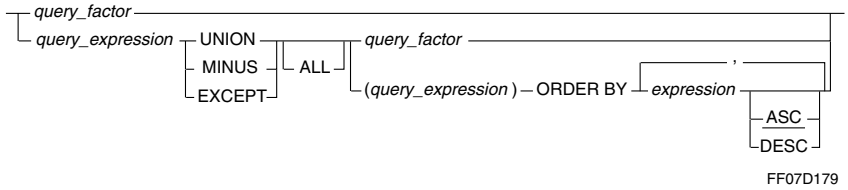
Syntax



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*.

Semantics



ANSI Compliance

EXCEPT is ANSI SQL-99-compliant.

MINUS and the ALL option are Teradata extensions to the ANSI SQL-99 standard.

Usage Notes

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

- Derived tables
- Subqueries
- INSERT … SELECT statements
- View definitions

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.

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 4-32.

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.

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

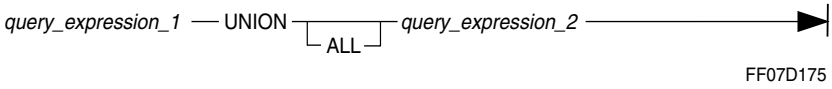
If the ALL option is specified, duplicate rows are retained. The ALL option can be specified for as many MINUS operators as are used in a multistatement query.

UNION Operator

Purpose

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

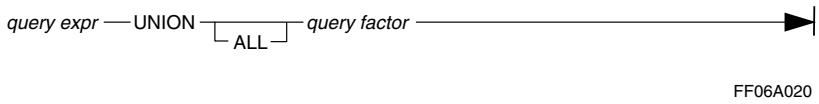
Syntax



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 4-2.
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 4-2.

Semantics



ANSI Compliance

UNION is ANSI SQL-99-compliant.

Valid UNION Operations

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

- Derived tables
- Subqueries
- INSERT … SELECT statements
- View definitions

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 4-34.

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 4-32.

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.

Sorting Rows for UNION

Before performing the sort operation used to check for duplicates in some union operations, Teradata 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.

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 above 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—INTEGER—through to the result.

You can confirm the unconverted data type for NULL and 144.87 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 this function is specified in the first SELECT . . .	The result data type is . . .	AND the value returned as the result is . . .
AVG	REAL	1
MIN	INTEGER	truncated to 0

Chapter 5:

DateTime and Interval Functions and Expressions

This chapter describes the DateTime and Interval functions and expressions, including:

- ANSI DateTime and Interval data type assignment rules
- Scalar operations on ANSI SQL-99 DateTime and Interval values
- ANSI DateTime expressions
- ANSI Interval expressions
- Arithmetic operators
- Aggregate functions and ANSI SQL-99 DateTime and Interval Data Types
- Scalar Operations and DateTime Functions
- Teradata Date and Time expressions
- Scalar operations on Teradata DATE values
- ADD_MONTHS function
- EXTRACT function

Overview

Introduction

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-99 standard.

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

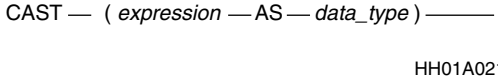
Source and Target Data Types Must be Compatible

In assignments involving ANSI DateTime or Interval data types, source values must be compatible with target data types. This differs from the Teradata convention of performing implicit conversions of source values to target data types for assignment.

Data Type Conversion

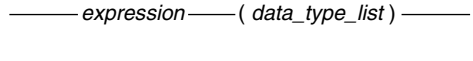
To perform conversions on ANSI DateTime or Interval data types, use one of the following forms:

- ANSI SQL-99-compliant CAST



HH01A021

- Teradata explicit conversion



HH01A019

For more information on the syntax, see "CAST in Explicit Data Type Conversions" on page 13-8 and "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 13-11.

DateTime Data Type Compatibility Rules

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

Data Type	Compatibility Rules
DATE	All DATEs are compatible with one another. 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	All TIMEs are compatible with one another. The Teradata system value TIME is encoded as a REAL and is not compatible with ANSI TIME or TIME WITH TIME ZONE.
TIMESTAMP	All TIMESTAMPs are compatible with one another.

Interval Data Type Compatibility Rules

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

Data Type	Compatibility Rules
Year-Month INTERVAL	All Year-Month INTERVALs are compatible with one another. • When the types match, assignment is straightforward. • When the source is INTERVAL YEAR and the target is INTERVAL YEAR TO MONTH, then the value for MONTH in the target is set to zero. • When the source is INTERVAL MONTH and the target is INTERVAL YEAR TO MONTH, then 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'. • When 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'. • When 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'.
Day-Time INTERVAL	All Day-Time INTERVALs are compatible with one another. • When the types match, assignment is straightforward. • When 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'. • When the target has fields of higher significance than the most significant field of the source, then 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'. • When 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-99 DateTime and Interval Values

Introduction

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

Scalar operations include:

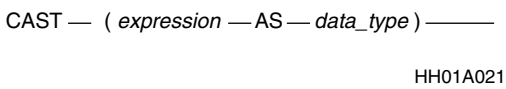
Operation	Description
DateTime Expressions	Expressions providing a result that is a DateTime value. DateTime expressions have arguments that are also DateTime or Interval expressions.
Interval Expressions	Expressions providing a result that is an Interval. Interval expressions may include components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

Data Type Compatibility

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

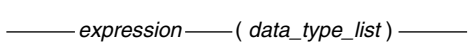
To perform conversions on ANSI DateTime or Interval expressions, use one of the following forms:

- ANSI SQL-99-compliant CAST



HH01A021

- Teradata explicit conversion



HH01A019

For more information on the syntax, see "CAST in Explicit Data Type Conversions" on page 13-8 and "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 13-11.

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

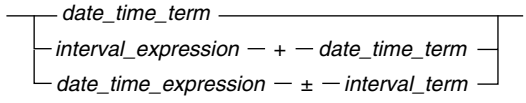
Purpose

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

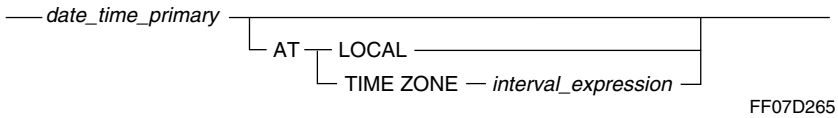
Definition

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

date_time_expression Syntax



date_time_term Syntax



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*.

Syntax element . . .	Specifies . . .
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 5-12.
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 Chapter 2: "Data Literals" in *Teradata RDBMS SQL Reference Volume 3*. • 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.

IF the type of the *date_time_primary* is . . .	THEN the time zone specifier is adjusted to . . .
DATE	an error. You cannot specify a TIME ZONE with a DATE value.
TIME	TIME WITH TIME ZONE
TIMESTAMP	TIMESTAMP WITH TIME ZONE
The time zone value is explicit in both cases.	
TIME WITH TIME ZONE	the specified TIME ZONE, replacing the previous “current” value.
TIMESTAMP WITH TIME ZONE	

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
```

ANSI Interval Expressions

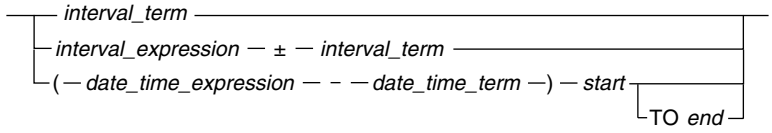
Purpose

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

Definition

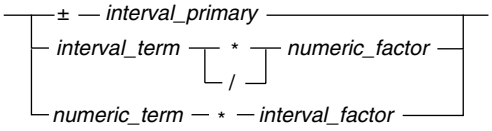
An interval expression is any expression that returns a result that is an INTERVAL value.

interval_expression Syntax



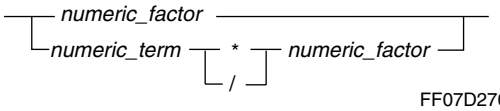
1101A010

interval_term Syntax



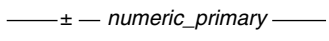
FF07D268

numeric_term Syntax



FF07D270

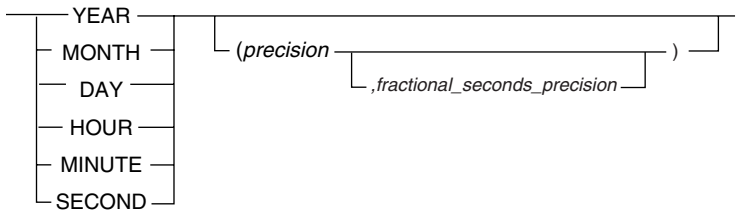
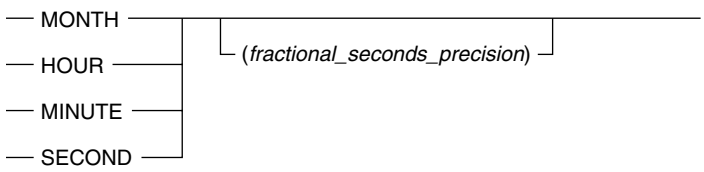
numeric_factor Syntax



FF07D269

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 5-7.
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 5-7.

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 Chapter 2: "Data Literals" in *Teradata RDBMS SQL Reference Volume 3*. • 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`.

Stage	Process
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’.

Examples 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`.

Stage	Process
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)`.

Stage	Process
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'.

Examples 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`.

Stage	Process
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`.

Stage	Process
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`.

Stage	Process
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.

Stage	Process
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.

Examples 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`.

Stage	Process
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)`.

Stage	Process
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;
```

Stage	Process
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.

Maximum and Minimum Values for Intervals With Multiple Fields

The following table lists the permissible minima and maxima for multiple-field intervals with a defined precision:

Multiple-Field Interval Type	Maximum Value	Minimum Value
YEAR(1) TO MONTH	‘9-11’	-’9-11’
YEAR(2) TO MONTH	‘99-11’	-’99-11’
YEAR(3) TO MONTH	‘999-11’	-’999-11’
YEAR(4) TO MONTH	‘9999-11’	-’9999-11’
DAY(1) TO HOUR	‘9 23’	-’9 23’
DAY(2) TO HOUR	‘99 23’	-’99 23’
DAY(3) TO HOUR	‘999 23’	-’999 23’
DAY(4) TO HOUR	‘9999 23’	-’9999 23’
DAY(1) TO MINUTE	‘9 23:59’	-’9 23:59’
DAY(2) TO MINUTE	‘99 23:59’	-’99 23:59’
DAY(3) TO MINUTE	‘999 23:59’	-’999 23:59’
DAY(4) TO MINUTE	‘9999 23:59’	-’9999 23:59’

Multiple-Field Interval Type	Maximum Value	Minimum Value
DAY(1) TO SECOND	‘9 23:59:59’	-’9 23:59:59’
DAY(2) TO SECOND	‘99 23:59:59’	-’99 23:59:59’
DAY(3) TO SECOND	‘999 23:59:59’	-’999 23:59:59’
DAY(4) TO SECOND	‘9999 23:59:59’	-’9999 23:59:59’
HOUR(1) TO MINUTE	‘9:59’	-’9:59’
HOUR(2) TO MINUTE	‘99:59’	-’99:59’
HOUR(3) TO MINUTE	‘999:59’	-’999:59’
HOUR(4) TO MINUTE	‘9999:59’	-’9999:59’
HOUR(1) TO SECOND	‘9:59:59’	-’9:59:59’
HOUR(2) TO SECOND	‘99:59:59’	-’99:59:59’
HOUR(3) TO SECOND	‘999:59:59’	-’999:59:59’
HOUR(4) TO SECOND	‘9999:59:59’	-’9999:59:59’
MINUTE(1) TO SECOND	‘9:59’	-’9:59’
MINUTE(2) TO SECOND	‘99:59’	-’99:59’
MINUTE(3) TO SECOND	‘999:59’	-’999:59’
MINUTE(4) TO SECOND	‘9999:59’	-’9999:59’

DAY TO SECOND, HOUR TO SECOND, and MINUTE TO SECOND multiple-field intervals can also specify a precision for fractional seconds, where the precision is a value from zero to six. For example, MINUTE(4) TO SECOND(6).

When multiple-field intervals specify fractional second precision, the maximum and minimum values in the preceding table are adjusted by the following minimum and maximum fractional values:

Fractional Seconds Precision	Maximum Fractional Value	Maximum Fractional Value
SECOND(1)	'.9'	-'.9'
SECOND(2)	'.99'	-'.99'
SECOND(3)	'.999'	-'.999'
SECOND(4)	'.9999'	-'.9999'
SECOND(5)	'.99999'	-'.99999'
SECOND(6)	'.999999'	-'.999999'

Arithmetic Operators

Introduction

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 extends the ANSI SQL-99 standard to allow the operations of adding or subtracting a number of days from an ANSI DATE value.

Teradata treats the number as an INTERVAL DAY value.

For more information, see "DATE and Integer Arithmetic" on page 5-27.

Example 1

Here is an example that subtracts two TIMESTAMP types.

First define a table:

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

Now, determine the difference, specifying an Interval unit of DAY TO SECOND for the result:

```
SELECT (end_time - start_time) DAY(4) TO SECOND
FROM BillDateTime;
```

The DAY(4) specifies four digits of precision, and allows for a maximum of 9999 days, or approximately 27 years.

The result is an Interval that looks like this:

```
5 16:49:20.340000
```

Example 2

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-99 DateTime types.

For this function ...	The result is ...		For more information, see ...
AVG(*arg*)	the type of the argument.		"AVG" on page 6-5
COUNT(*arg*)	INTEGER, if the mode is Teradata.		"COUNT" on page 6-10
	DECIMAL(*n*,0), if the mode is ANSI, where:		
	***n* is ...**	**if MaxDecimal in DBSControl is ...**	
	15	0 or 15	
	18	18	
MAX(*arg*)	the type of the argument, based on the comparison rules for DateTime types.		"MAX" on page 6-22
MIN(*arg*)			"MIN" on page 6-24

Interval Data Types

The following aggregate functions are valid for Interval types.

For this function ...	The result is ...		For more information, see ...
AVG(*arg*)	the type of the argument.		"AVG" on page 6-5
COUNT(*arg*)	INTEGER, if the mode is Teradata.		"COUNT" on page 6-10
	DECIMAL(*n*,0), if the mode is ANSI, where:		
	***n* is ...**	**if MaxDecimal in DBSControl is ...**	
	15	0 or 15	
	18	18	
MAX(*arg*)	the type of the argument, based on the comparison rules for DateTime types.		"MAX" on page 6-22
MIN(*arg*)			"MIN" on page 6-24
SUM(*arg*)	the type of the argument.		"SUM" on page 6-60

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;
```

For More Information

For more information on ...	See ...
CURRENT_DATE	“CURRENT_DATE” on page 12-4
CURRENT_TIME	“CURRENT_TIME” on page 12-5
CURRENT_TIMESTAMP	“CURRENT_TIMESTAMP” on page 12-7
EXTRACT	“EXTRACT” on page 5-36

Teradata Date and Time Expressions

Introduction

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-99 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 5-3 and "Scalar Operations on ANSI SQL-99 DateTime and Interval Values" on page 5-5.

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 table:

Stage	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 table:

Stage	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-99 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

Introduction

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-99 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-99 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 5-30.

EXTRACT Function

The EXTRACT function can be used to get the year, month, or day from a date. The result is an integer.

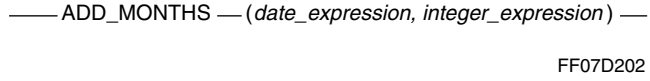
For details, see "EXTRACT" on page 5-36.

ADD_MONTHS

Introduction

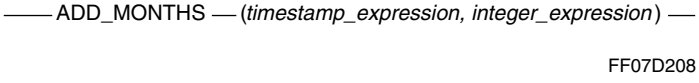
Adds an integer number of months to a DATE or TIMESTAMP expression and normalizes the result.

Date Syntax



FF07D202

Timestamp Syntax



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 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 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-99 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 non-valid dates are normalized to ensure that all reported dates are valid.

The Problem of Valid Results With Scalar Arithmetic on Months

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 5-8.

The ADD_MONTHS function uses an algorithm that permits you to 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 ADD_MONTHS function or ANSI SQL-99 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-defined Calendar view for date arithmetic. For information, see “CALENDAR View” in the “DateTime and Interval Data Types” chapter in *Teradata RDBMS SQL Reference, Volume 3*.

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-99-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 non-valid date value is returned by the function.

- ANSI SQL-99 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-99 dates and intervals as follows.

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

The equation is true unless a non-valid date such as 1999-09-31 results, in which case the ANSI expression traps the non-valid 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 5-34 with the results of "Example 7" on page 5-35.

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-99 DateTime interval arithmetic instead. For more information, see "ANSI Interval Expressions" on page 5-12.

Remember that ADD_MONTHS changes the DAY value of the result *only* when a non-valid *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 5-34.

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 5-34.

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 is probably not 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);
```

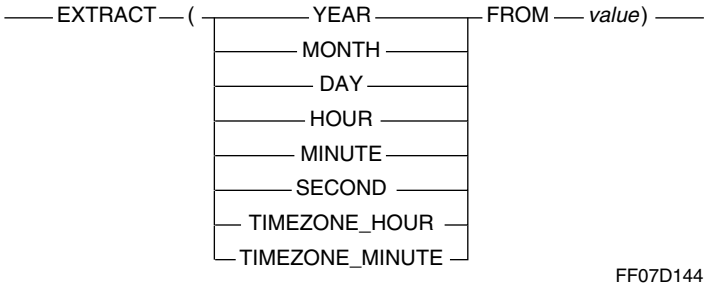
ADD_MONTHS returns the result 1999-05-30.

EXTRACT

Introduction

Extracts a single specified full ANSI SQL-99 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	a DateTime value expression or Interval value expression.

With the exception of non-ANSI Teradata definitions, all the expressions defined by the keyword list above are ANSI SQL-99 DateTime or Interval value expressions.

IF *value* is . . .	THEN . . .
a character string literal	the literal must be of the form 'YYYY-MM-DD-hh:mm-ss'. When *value* is a literal, its validity is checked as part of the function
not a character string literal	it must be the value of an expression that results in a Teradata DATE data type.
NULL	the result is NULL.
not NULL	the result is an integer.

ANSI Compliance

EXTRACT is partially ANSI SQL-99-compliant. Full ANSI EXTRACT permits any field in the specified value to be extracted.

Teradata EXTRACT Is A Subset of ANSI SQL-99 EXTRACT

Full ANSI SQL-99 EXTRACT allows the extraction of any field contained in any DateTime or Interval value.

In addition to the ANSI SQL-99 extract function, Teradata also supports EXTRACT for YEAR, MONTH, or DAY extracted from a Teradata DATE value, or HOUR, MINUTE, or SECOND extracted from a Teradata REAL value.

ANSI SQL-99 EXTRACT Results

EXTRACT returns an exact numeric value for ANSI SQL-99 DateTime values. The precision of the response is implementation-defined.

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.

For values other than SECOND, the result is always of type INTEGER with 32 bits of precision.

For values of SECOND, if the SECOND field of *data_value* has a fractional precision of zero, the result is type INTEGER, else the result is DECIMAL with the scaling as specified for the SECOND field in its data description.

Example1

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

```
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.

```
EXTRACT (MONTH FROM PurchaseDate+90)
```

Example 3

The following returns 12 as an integer.

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

Example 4

This example returns an error, found by trying to evaluate the character string literal as a date.

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

Example 5

The following returns an error, found by trying to evaluate the character string literal as a date. In this case, the reason is that EXTRACT uses the ANSI SQL-99 format for recognizing a date as a character string literal.

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

Except for character string literals, if the argument for this form of EXTRACT is a value of the Teradata RDBMS type DATE, the value contained is warranted to be a valid date, for which EXTRACT cannot return an error.

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

Externally created values of time can be appropriately encoded and stored in a REAL column, to any desired precision as long as the encoding created a value representable by REAL without precision loss.

You should not store DateTime values as REAL in any new applications. Instead, use the more rigorously defined ANSI SQL-99 DateTime data types.

Example 6

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

```
EXTRACT (HOUR FROM CURRENT_TIME)
```

Example 7

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.

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

Example 8

The following example relates to non-ANSI DateTime definitions and returns the seconds as DECIMAL(8,2). This choice of precision is based on the fact that system time on the Teradata RDBMS is measured in centiseconds.

```
EXTRACT (SECOND FROM CURRENT_TIME)
```

Chapter 6:

Aggregate Functions

This chapter describes the following SQL aggregate functions:

- AVG
- CORR
- COUNT
- COVAR_POP
- COVAR_SAMP
- KURTOSIS
- MAX
- MIN
- REGR_AVGX
- REGR_AVGY
- REGR_COUNT
- REGR_INTERCEPT
- REGR_R2
- REGR_SLOPE
- REGR_SXX
- REGR_SXY
- REGR_SYY
- SKEW
- STDDEV_POP
- STDDEV_SAMP
- SUM
- VAR_POP
- VAR_SAMP

For aggregate window functions and their Teradata-specific equivalents, see Chapter 7: “Ordered Analytical Functions.”

Aggregate Functions

Introduction

Aggregate functions are typically used in arithmetic expressions. The result of an aggregate operation is a single numeric value in the result table.

You can use GROUP BY clauses to produce more complex, finer grained results in multiple result 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 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.

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.

This is so because aggregates return only one value, while a non-GROUP BY column reference can return any number of values.

Nesting Aggregates

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

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

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

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

For more information on aggregate window functions, see Chapter 7: “Ordered Analytical Functions”.

Aggregate Results When an Expression Evaluates to Zero

Aggregation on an expression that represents zero elements behaves as indicated by the following table:

This form of aggregate function …	Returns this result when *expression* evaluates to 0 …
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 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 6-12 and "Example 3" on page 6-12. Note that the aggregates are behaving exactly as they should—the results are *not* mathematically anomalous.

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

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

Aggregates Used in WHERE and HAVING Clauses

Aggregates can be used in the following types of clauses:

- The WHERE clause of an ABORT statement to specify an abort condition.
  But an aggregate function *cannot* be used 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 ...
```

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 ...
```

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

```
SELECT SUM(DISTINCT x), SUM(DISTINCT y) FROM ...
```

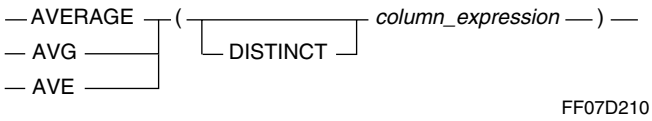
DISTINCT aggregations can be done only at the first level of aggregation.

AVG

Purpose

Returns a column value that is the arithmetic average of a specified column in a result table.

Syntax



where:

Syntax element ...	Specifies ...
DISTINCT	not to include duplicate values in *column_expression* when computing the average value for the column.
column_expression	the column or column expression for which an average is to be computed.

ANSI Compliance

AVG is ANSI SQL-99-compliant.

AVERAGE and AVE are Teradata extensions to the ANSI standard.

For the AVG window function that computes a group, cumulative, or moving average, see "AVG" on page 7-24.

Result Type and Attributes

The following table lists the default attributes for the result of AVG(x):

Data Type	Format	Title
REAL	**IF the operand is ...** / **THEN the format is ...** • numeric, • date, • interval → the same format as x. character → the default format for FLOAT.	Average(x)

For an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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 *column_expression* reference must not be 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 "Manipulating Nulls" in the chapter "SQL Data Handling Fundamentals" in *Teradata RDBMS SQL Reference, Volume 1* and "Aggregates and Nulls" on page 6-3.

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;

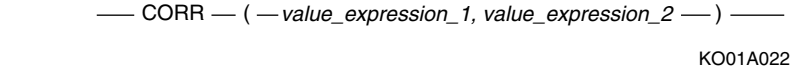
Region and Avg (sales)
----------------------
North         21840.17
East          55061.32
Midwest       15535.73
```

CORR

Purpose

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

Syntax



where:

Syntax element …	Specifies …
value_expression_2	a numeric column expression to be correlated with a second numeric column expression. The expressions cannot contain any ordered analytical or aggregate functions.
value_expression_1	

ANSI Compliance

CORR is ANSI SQL-99-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.
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:

$$\text{CORR} = \frac{\text{COVAR\_SAMP}(x,y)}{\text{STDDEV\_SAMP}(x)\text{STDDEV\_SAMP}(y)}$$

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 Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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 7: "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
```

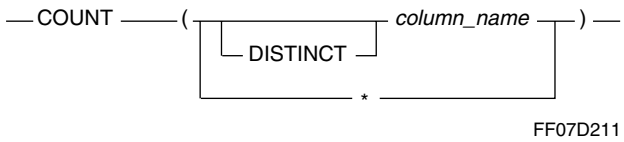
The result -.9543 suggest an inverse relationship between the variables. That is, for the area and sales price range specified in the query, the value for NbrSold increases as sales price decreases and decreases as sales price increases.

COUNT

Purpose

Returns a column value that is the number of qualified rows in a result table.

Syntax



where:

Syntax element ...	Specifies ...
DISTINCT	not to include duplicate values in *column_name* when counting the number of non-null occurrences (rows) for the column.
column_name	the name of the column for which a row count of non-null entries is to be computed.
*	count all occurrences, including nulls.

THIS syntax ...	Counts the total number of ...
COUNT(*column_name*)	non-null occurrences of *column_name*.
COUNT (DISTINCT *column_name*)	non-null unique row values of *column_name*.
COUNT(*)	rows in each group of a GROUP BY clause, including groups with nulls for *column_name*.

For COUNT functions that return the group, cumulative, or moving count, see "COUNT" on page 7-30.

Usage Notes

COUNT is valid for any data type.

With the exception of COUNT(*), the computation does not include nulls. For more information, see "Manipulating Nulls" in the chapter "SQL Data Handling Fundamentals" in *Teradata RDBMS SQL Reference, Volume 1* and "Aggregates and Nulls" on page 6-3.

For an example that uses COUNT(*) and nulls, see "Example 2" on page 6-12.

Result Type and Attributes

The following table lists the default attributes for the result of COUNT:

Operation	Mode	Data Type		Format	Title
COUNT(x)	ANSI	DECIMAL(*n*,0) where		Default format for resulting data type	Count(x)
		***n* is ...**	**if MaxDecimal in DBSControl is ...**		
		15	0 or 15		
		18	18		
	Teradata	INTEGER			
COUNT(*)	ANSI	DECIMAL(*n*,0) where		Default format for resulting data type	Count(*)
		***n* is ...**	**if MaxDecimal in DBSControl is ...**		
		15	0 or 15		
		18	18		
	Teradata	INTEGER			

ANSI mode uses DECIMAL because Teradata 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 change the script to use CAST, as illustrated by the following example.

```
SELECT CAST(COUNT(*) AS DECIMAL(15,0) FORMAT 'Z(14)9')
FROM BIGTABLE;
```

For information on data type default formats, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

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.

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.

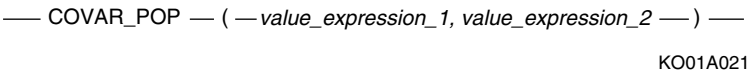
```
Count(Distinct(DeptNo))
-----------------------
                      5
```

COVAR_POP

Purpose

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

Syntax



KO01A021

where:

Syntax element ...	Specifies ...
value_expression_2	a numeric column expression to be paired with a second numeric column 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-99-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 7: "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:

$$\text{COVAR\_POP} = \frac{\text{SUM}((x - \text{AVG}(x))(y - \text{AVG}(y)))}{\text{COUNT}(x)}$$

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:

Data Type	Format	Title
REAL	**IF the operand is ...** / **THEN the format is ...** character → the default format for FLOAT. • numeric • date • interval → the same format as x.	COVAR_POP(y,x)

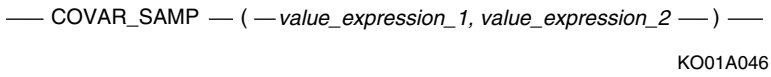
For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

COVAR_SAMP

Purpose

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

Syntax



where:

Syntax element ...	Specifies ...
value_expression_2	a numeric column expression to be paired with a second numeric column 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-99-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 7: "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:

$$\text{COVAR\_SAMP} = \frac{\text{SUM}((x - \text{AVG}(x))(y - \text{AVG}(y)))}{\text{COUNT}(x) - 1}$$

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:

Data Type	Format	Title
REAL	IF the operand is ... / THEN the format is ... character / the default format for FLOAT. • numeric • date • interval / the same format as x.	COVAR_SAMP(y,x)

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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

c1	height	weight
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;

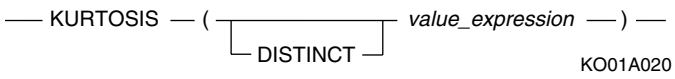
Covar_Samp(weight,height)
-------------------------
                      150
```

KURTOSIS

Purpose

Returns the kurtosis of the distribution of *value_expression*.

Syntax



where:

Syntax element ...	Specifies ...
value_expression	a scalar numeric 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.
DISTINCT	to exclude duplicates of *value_expression* from the computation.

ANSI Compliance

KURTOSIS is ANSI SQL-99-compliant.

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 Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Computation

The equation for computing KURTOSIS is defined as follows:

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

where:

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

IF ...	THEN KURTOSIS returns ...
there are fewer than four non-null data points in the data used for the computation	NULL
`STDDEV_SAMP(x) = 0`	

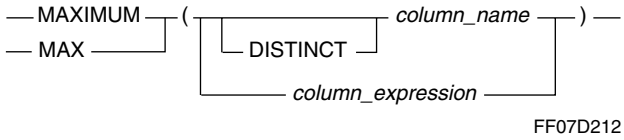
Division by zero results in NULL rather than an error.

MAX

Purpose

Returns a column value that is the maximum value in *column_name* in a result table.

Syntax



where:

Syntax element ...	Specifies ...
DISTINCT	not to include duplicate values in *column_name* when computing the maximum value for the column.
column_name	the name of the column for which the maximum is to be computed.
column_expression	a value derived from an arithmetic computation on the values in *column_name*. The result table maximum is computed from the expression rather than from the raw column values.

ANSI Compliance

MAX is ANSI SQL-99-compliant.

MAXIMUM is a Teradata extension to the ANSI SQL-99 standard.

For the MAX window function that computes a group, cumulative, or moving maximum value, see "MAX" on page 7-43.

Result Type and Attributes

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

Data Type	Format	Title
Same data type as operand x.	Same format as operand x.	Maximum(x)

Usage Notes

MAX is valid for character data as well as numeric data.

When used with a character expression, MAX returns the highest sort order.

Nulls are not included in the result computation. For more information, see "Manipulating Nulls" in the chapter "SQL Data Handling Fundamentals" in *Teradata RDBMS SQL Reference, Volume 1* and "Aggregates and Nulls" on page 6-3.

If you use the *column_expression* syntax, the *column_expression* must refer to at least one column in the table from which data is selected.

The *column_name* reference must not be to a view column that is derived from a function.

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 * ProdID)
FROM Inventory;

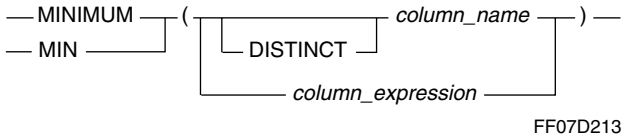
Maximum(ProdID)
---------------
3815
```

MIN

Purpose

Returns a column value that is the minimum value in *column_name* in a result table.

Syntax



where:

Syntax element . . .	Specifies . . .
DISTINCT	not to include duplicate values in *column_name* when computing the minimum value for the column.
column_name	the name of the column for which a minimum is to be computed.
column_expression	a value derived from an arithmetic computation on the values in *column_name*. The result table minimum is computed from the expression rather than from the raw column values.

ANSI Compliance

MIN is ANSI SQL-99-compliant.

MINIMUM is a Teradata extension to the ANSI SQL-99 standard.

For the MIN window function that computes a group, cumulative, or moving minimum value, see "MIN" on page 7-51.

Result Type and Attributes

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

Data Type	Format	Title
Same data type as operand x.	Same format as operand x.	Minimum(x)

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 the chapter "SQL Data Handling Fundamentals" in *Teradata RDBMS SQL Reference, Volume 1* and "Aggregates and Nulls" on page 6-3.

If you use the *column_expression* syntax, the *column_expression* must refer to at least one column in the table from which data is selected.

The *column_name* reference must not be to a view column that is derived from a function.

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

The report is as follows.

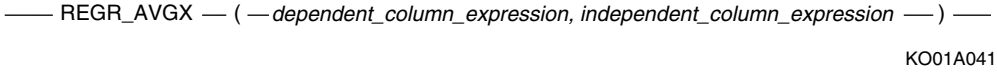
ProdID	Quantity
1124	24
1355	21
3215	25
4391	22

REGR_AVGX

Purpose

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

Syntax



KO01A041

where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_AVGX is ANSI SQL-99-compliant.

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.

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 7: “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:

$$\text{REGR\_AVGX} = \frac{\text{SUM}(x)}{n}$$

where:

This variable ...	Represents ...
x	*independent_column_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:

Data Type	Format	Title
REAL	**IF the operand is ...** / **THEN the format is ...** character / the default format for FLOAT. • numeric • date • interval / the same format as x.	REGR_AVGX(y,x)

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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;

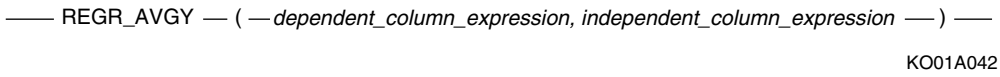
Regr_Avgx(weight,height)
------------------------
                      68
```

REGR_AVGY

Purpose

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

Syntax



where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_AVGY is ANSI SQL-99-compliant.

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.

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 7: “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:

$$\text{REGR\_AVGY} = \frac{\text{SUM(y)}}{\text{n}}$$

where:

This variable ...	Represents ...
y	*dependent_column_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:

Data Type	Format	Title
REAL	IF the operand is ... / THEN the format is ... character: the default format for FLOAT. • numeric • date • interval: the same format as x.	REGR_AVGY(y,x)

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

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;

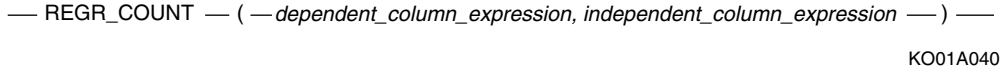
Regr_Avgy(weight,height)
------------------------
                     140
```

REGR_COUNT

Purpose

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

Syntax



KO01A040

where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_COUNT is ANSI SQL-99-compliant.

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.

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 7: “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 data type, format, and title for REGR_COUNT(y, x) are as follows:

Data Type	Format	Title
REAL	IF the operand is … / THEN the format is … character / the default format for FLOAT. numeric / the same format as x.	REGR_COUNT(y,x)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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;

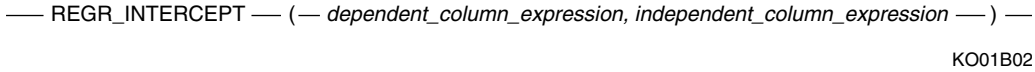
Regr_Count(weight,height)
-------------------------
                        9
```

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



KO01B02

where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_INTERCEPT is ANSI SQL-99-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 7: "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:

$$\text{REGR\_INTERCEPT} = \text{AVG}(y) - \text{REGR\_SLOPE}(y,x)\text{AVG}(x)$$

where:

This variable ...	Represents ...
x	*independent_column_expression*
y	*dependent_column_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 Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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;

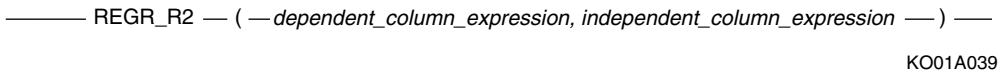
REGR_INTERCEPT(NbrSold,SalesPrice)
----------------------------------
                             249.9
```

REGR_R2

Purpose

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

Syntax



KO01A039

where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_R2 is ANSI SQL-99-compliant.

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.

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 7: “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:

$$\text{REGR\_R2} = \frac{\text{POWER}(\text{COUNT}(xy) \bullet \text{SUM}(xy) - \text{SUM}(x) \bullet \text{SUM}(y)\ ,\ 2)}{((\text{COUNT}(xy) \bullet \text{SUM}(x^{**}2) - \text{SUM}(x) \bullet \text{SUM}(x)) \bullet (\text{COUNT}(xy) \bullet \text{SUM}(y^{**}2) - \text{SUM}(y) \bullet \text{SUM}(y)))}$$

where:

This variable ...	Represents ...
x	*independent_column_expression* x is the independent, or predictor, variable expression.
y	*dependent_column_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:

Data Type	Format	Title
REAL	**IF the operand is ...** / **THEN the format is ...** character / the default format for FLOAT. numeric / the same format as x.	REGR_R2(y,x)

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

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;

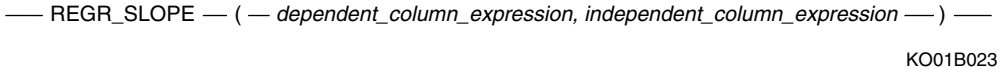
REGR_R2(weight,height)
----------------------
                   .58
```

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



KO01B023

where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_SLOPE is ANSI SQL-99-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 7: “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:

$$\text{REGR\_SLOPE} = \frac{(\text{COUNT(x)SUM(x*y)}) - (\text{SUM(x)SUM(y)})}{(\text{COUNT(x)SUM(x**2)}) - (\text{SUM(x)**2})}$$

where:

This variable . . .	Represents . . .
x	*independent_column_expression*
y	*dependent_column_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 an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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;

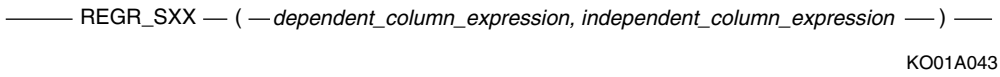

REGR_SLOPE(NbrSold,SalesPrice)
------------------------------
        -7.92857142857143E-004
```

REGR_SXX

Purpose

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

Syntax



KO01A043

where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_SXX is ANSI SQL-99-compliant.

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.

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 7: “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_column_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:

Data Type	Format	Title
REAL	**IF the operand is ...** / **THEN the format is ...** character / the default format for FLOAT. • numeric • date • interval / the same format as x.	REGR_SXX(y,x)

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

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;

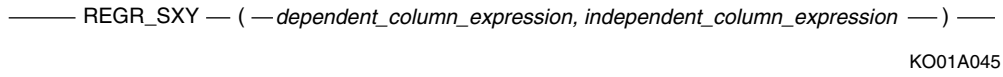
Regr_Sxx(weight,height)
-----------------------
                    240
```

REGR_SXY

Purpose

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

Syntax



where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_SXY is ANSI SQL-99-compliant.

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.

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 7: “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_column_expression* x is the independent, or predictor, variable expression.
y	*dependent_column_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:

Data Type	Format	Title
REAL	**IF the operand is ...** / **THEN the format is ...** character / the default format for FLOAT. • numeric • date • interval / the same format as x.	REGR_SXY(y,x)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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;

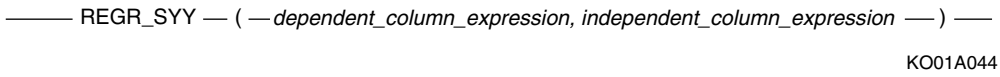
Regr_Sxy(weight,height)
-----------------------
                   1200
```

REGR_SYY

Purpose

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

Syntax



where:

Syntax element . . .	Specifies . . .
dependent_column_expression	the dependent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.
independent_column_expression	the independent variable for the regression. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

REGR_SYY is ANSI SQL-99-compliant.

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.

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 7: “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_column_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:

Data Type	Format	Title
REAL	**IF the operand is ...** / **THEN the format is ...** character / the default format for FLOAT. • numeric • date • interval / the same format as x.	REGR_SYY(y,x)

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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 ...
value_expression	a scalar numeric 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.
DISTINCT	to exclude duplicates of *value_expression* from the computation.

ANSI Compliance

SKEW is ANSI SQL-99-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 Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Computation

The equation for computing SKEW is defined as follows:

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

where:

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

IF ...	THEN SKEW returns ...
there are fewer than three non-null data points in the data used for the computation	NULL
STDDEV_SAMP(x) = 0	

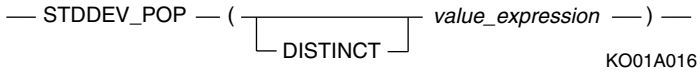
Division by zero results in NULL rather than an error.

STDDEV_POP

Purpose

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

Syntax



where:

Syntax element . . .	Specifies . . .
value_expression	a scalar numeric column expression whose population standard deviation is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
DISTINCT	to exclude duplicates of *value_expression* from the computation.

ANSI Compliance

STDDEV_POP is ANSI SQL-99-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 7: “Ordered Analytical Functions”.

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

Report Breaks

STDDEV_POP operates differently depending on whether or not 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 6-57.

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:

$$\text{STDDEV\_POP} = \text{SQRT}(\text{COUNT(x)SUM(x**2)} - \frac{(\text{SUM(x)**2})}{(\text{COUNT(x)**2})})$$

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>
</table>
</td><td>STDDEV_POP(x)</td></tr>
</table>

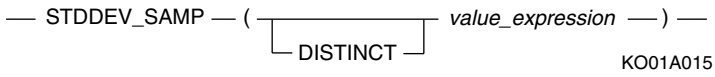
For information on the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

STDDEV_SAMP

Purpose

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

Syntax



where:

Syntax element ...	Specifies ...
value_expression	a scalar numeric column expression whose sample standard deviation is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
DISTINCT	to exclude duplicates of *value_expression* from the computation.

ANSI Compliance

STDDEV_SAMP is ANSI SQL-99-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 than that for the population standard deviation to minimize the effect of outliers on the computed value.

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 7: “Ordered Analytical Functions”.

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

Report Breaks

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

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 6-54.

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_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*

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

The data type, format, and title for STDDEV_SAMP(x) are as follows:

Data Type	Format	Title
REAL	**IF the operand is …** / **THEN the format is …** character / the default format for FLOAT. • numeric • date • interval / the same format as x.	STDDEV_SAMP(x)

For information on the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

SUM

Purpose

Returns a column value that is the arithmetic sum of a specified column in a result table.

Syntax



where:

Syntax element ...	Specifies ...
DISTINCT	not to include duplicate values in *column_expression* when computing the sum for the column.
column_expression	the column or column expression for which a sum is to be computed.

ANSI Compliance

SUM is ANSI SQL-99-compliant.

For the SUM function that returns the cumulative, group, or moving sum, see "SUM" on page 7-74.

Result Type and Attributes

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

Data Type	Format	Title
Same as the operand	**IF the operand is ...** / **THEN the format is ...** BYTEINT or SMALLINT → the default format of the INTEGER data type. character → the default format for FLOAT. not SMALLINT, BYTEINT, or character → the default format for the data type of the operand.	Sum(x)

For an explanation of the formatting characters in the format, and information on data type default formats, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Usage Notes

The *column_expression* reference 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 Chapter 1: "SQL Data Handling Fundamentals" of *Teradata RDBMS SQL Reference, Volume 1* and "Aggregates and Nulls" on page 6-3.

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))
```

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;

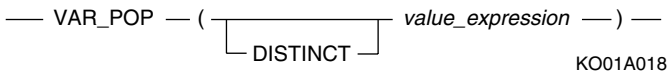
Sum
38,525,151.91
```

VAR_POP

Purpose

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

Syntax



where:

Syntax element ...	Specifies ...
value_expression	a scalar numeric column expression whose population variance is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
DISTINCT	to exclude duplicates of *value_expression* from the computation.

ANSI Compliance

VAR_POP is ANSI SQL-99-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:

$$\text{VAR\_POP} = \text{COUNT(x)SUM(x**2)} - \frac{\text{(SUM(x)**2)}}{\text{(COUNT(x)**2)}}$$

where:

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

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

Division by zero results in NULL rather than an error.

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 7: "Ordered Analytical Functions".

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

Report Breaks

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

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 6-64.

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.

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>
</table>
</td><td>VAR_POP(x)</td></tr>
</table>

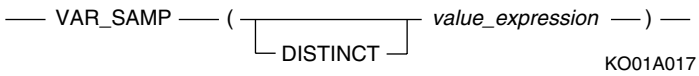
For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

VAR_SAMP

Purpose

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

Syntax



where:

Syntax element ...	Specifies ...
value_expression	a scalar numeric column expression whose sample variance is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
DISTINCT	to exclude duplicates of *value_expression* from the computation.

ANSI Compliance

VAR_SAMP is ANSI SQL-99-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:

$$\text{VAR\_SAMP} = \text{COUNT(x)SUM(x**2)} - \frac{\text{(SUM(x)**2)}}{\text{(COUNT(x))(COUNT(x)} - 1\text{)}}$$

where:

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

When there are fewer than two non-null data points in the sample used for the computation, then 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 7: "Ordered Analytical Functions".

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

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 6-62.

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>
</table>
</td><td>VAR_SAMP(x)</td></tr>
</table>

For information on the default format of data types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Chapter 7:

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-99-compliant window functions, as well as Teradata-specific functions. Topics include:

- Window aggregate functions
  - AVG
  - COUNT
  - MAX
  - MIN
  - SUM
- Rank function
  - RANK
- Distribution function
  - PERCENT_RANK
- Row number function
  - ROW_NUMBER
- Teradata-specific functions
  - CSUM
  - MAVG
  - MDIFF
  - MLINREG
  - MSUM
  - QUANTILE
  - RANK

Ordered Analytical Functions

Introduction

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.

How Ordered Analytical Functions Extend Teradata Queries

Introduction

Ordered analytical functions extend the Teradata 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 Functions and Hash or Join Indexes

When an ordered analytical function is specified on columns that are also defined for a compressed hash or join index, the Optimizer does not select the hash or join index to process the query.

Why Perform 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 and many third-party query management and analytical tools have full access to the Teradata 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-specific functions. For example, the Teradata Warehouse Miner FREQ function uses the Teradata-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 = 50
ORDER BY xcnt DESC, gender, marital_status
```

The result for this query looks like the following table.

Gender	Marital Status	Count	Pct	CumulativeCnt	CumulativePct	Rank</B<>
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

Introduction

Teradata supports two syntax alternatives for ordered analytical functions:

- ANSI SQL-99-compliant
- Teradata

Window aggregate, rank, distribution, and row number functions are ANSI SQL-99-compliant, while Teradata-specific functions are not.

The use of Teradata-specific functions is strongly discouraged. These functions are retained only for backward compatibility with existing applications.

Relationship Between Teradata-Specific Functions and Window Functions

The following table identifies equivalent ANSI SQL-99 window functions for Teradata-specific functions:

Teradata-Specific Functions	Equivalent ANSI SQL-99 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

Ordered Analytical Computations

The following table lists ordered analytical computations and the functions that enable you to perform them:

IF you want to compute the . . .	THEN use this function . . .
cumulative sum	• SUM window function • CSUM
cumulative, group, or moving count	COUNT window function
group sum	SUM window function
moving average	• AVG window function • MAVG
moving difference between the current row-column value and the preceding n^{th} row-column value	MDIFF
moving linear regression	MLINREG
moving sum	• SUM window function • MSUM
quantile scores for the values in a column	QUANTILE
ordered rank of all rows in a group	• RANK window function • RANK
relative rank of a row in a group	PERCENT_RANK window function
sequential row number of the row within its window partition according to the window ordering of the window	ROW_NUMBER
cumulative, group, or moving maximum value	MAX window function
cumulative, group, or moving minimum value	MIN window function

Window Feature

Introduction

The window feature is ANSI SQL-99 compliant and 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
- ROWS

The window feature can be applied to the ANSI SQL-99-compliant window functions:

- AVG
- COUNT
- MAX
- MIN
- PERCENT_RANK
- RANK
- ROW_NUMBER
- SUM

PARTITION BY Phrase

PARTITION BY determines the group, or groups, over which the ordered analytical function executes.

If there is no PARTITION BY phrase, then the entire result set, delivered by the FROM clause, constitutes a single group, or 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
```

ROWS Phrase

ROWS can be specified with the ANSI SQL-99-compliant window aggregate functions:

- AVG
- COUNT
- MAX
- MIN
- SUM

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 Month, 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-99-compliant aggregate functions:

- AVG
- COUNT
- MAX
- MIN
- SUM

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 month 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.

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 SUM(sale),
SUM(profit) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING)
FROM facts
GROUP BY sale, profit
HAVING SUM(sale) > 15
QUALIFY SUM(profit) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING) > 2;
```

Use of DISTINCT Clause

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

Not Permitted in Subqueries

Ordered analytical functions are not permitted in subqueries.

Not Permitted in WHERE Clauses

Ordered analytical functions are not permitted in WHERE clauses.

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.

Computation Sort Order Does Not Define 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                     ?
```

Nesting Aggregates in Ordered Analytical Functions

Introduction

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

Even though only window functions allow aggregates specified together in the same SELECT list, both 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, 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:

```
SELECT StoreID, COUNT(StoreID) OVER ()
FROM sales_tbl
GROUP BY StoreID;
```

The results look like this:

```
StoreID_1  Group Count(StoreID_2)
---------  ----------------------
     1001                       2
     1002                       2
```

Without the GROUP BY, the results look like this:

```
StoreID  Group Count(StoreID)
-------  --------------------
   1001                     8
   1001                     8
   1001                     8
   1001                     8
   1002                     8
   1002                     8
   1002                     8
   1002                     8
```

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.

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   2000.00              2000.00
   1001   1000.00              3000.00
   1001   3000.00              6000.00
   1001   1000.00              7000.00
   1002   1000.00              1000.00
   1002   3000.00              4000.00
   1002   3000.00              7000.00
   1002   2000.00              9000.00
```

Combining Window Functions, Teradata-Specific Functions, and GROUP BY

The following table provides the semantics of the allowable combinations of window functions, Teradata-specific functions, aggregate functions, and the GROUP BY clause.

Combination				Semantics
Window Function	Teradata-Specific Function	Aggregate Function	GROUP BY Clause	
X				A value is computed for each row.
	X			A value is computed for each row. The entire table constitutes a single group, or partition, over which the Teradata-specific function executes.
		X		One aggregate value is computed for the entire table.
X			X	GROUP BY collapses all rows with the same value for the group-by columns into a single row, and a value is computed for each resulting row.
	X		X	GROUP BY determines the partitions over which the Teradata-specific function executes. The clause does *not* collapse all rows with the same value for the group-by columns into a single row.
		X	X	An aggregation is performed for each group.
X	X			Teradata-specific functions do not have partitions. The whole table is one partition.
X	X		X	GROUP BY determines partitions for Teradata-specific functions. GROUP BY does *not* collapse all rows with the same value for 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: Some 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
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
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).

Ordered Analytical Functions

The following sections describe the ordered analytical functions in alphabetical order.

AVG

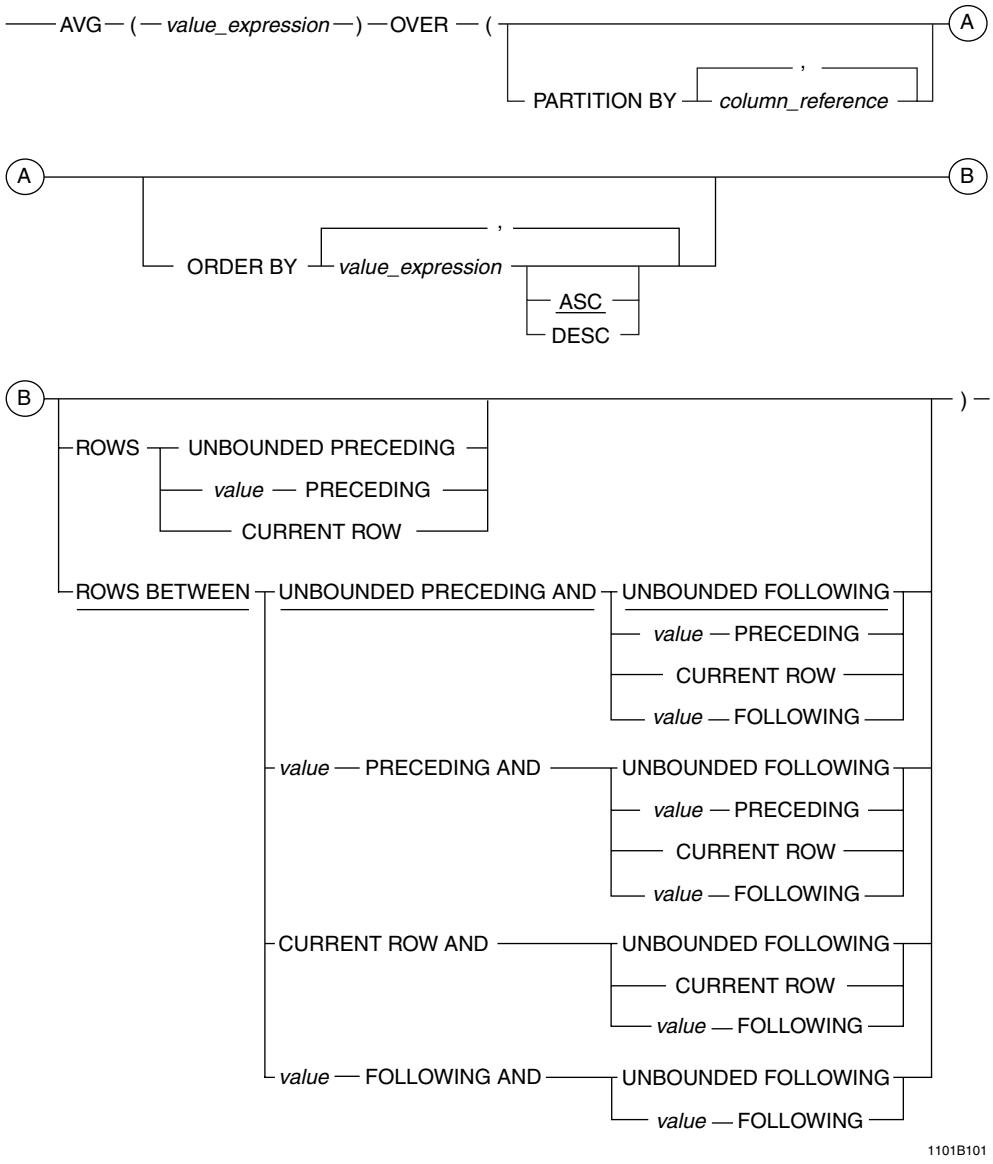
Purpose

Computes the cumulative, group, or moving average of a *value_expression*.

Type

SQL-99 window aggregate function.

Syntax



where:

Syntax element …	Specifies …
value_expression	the column for which a cumulative, group, or moving average is computed.
OVER	the arguments that define how values are grouped, ordered, and considered when computing the cumulative, group, or moving average for the *value_expression*. Values are grouped according to the optional PARTITION BY clause, 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 AVG function operates. PARTITION BY is optional. If there is no PARTITION BY clause, 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 v*alue_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.
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.

Syntax element . . .	Specifies . . .
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

AVG is ANSI SQL-99-compliant.

Using AVG Instead of MAVG

The form of the AVG window function that specifies an aggregation group of ROWS *value* PRECEDING is the ANSI equivalent of the MAVG Teradata-specific function.

Note that the ROWS *value* PRECEDING phrase specifies the number of rows preceding the current row that are used, together with the current row, to compute the moving average. The total number of rows in the aggregation group is *value* + 1. For the MAVG function, the total number of rows in the aggregation group is the value of *width*.

For AVG window function, an aggregation group of ROWS 5 PRECEDING, for example, means that the 5 rows preceding the current row, plus the current row, are used to compute the moving average. Thus the moving average for the 6th row of a partition would have considered row 6, plus rows 5, 4, 3, 2, and 1 (that is, 6 rows in all).

For the MAVG function, a *width* of 5 means that the current row, plus 4 preceding rows, are used to compute the moving average. The moving average for the 6th row would have considered row 6, plus rows 4, 5, 3, and 2 (that is, 5 rows in all).

Relationship to the AVG Aggregate Function

For the description of the AVG aggregate function, see "AVG" on page 6-5.

value_expression

AVG can take a single sum column as an argument (sales, for example) or an expression (sale * profit).

The following is *not* valid:

```
AVG(*)
```

Problems With Missing Data

Ensure that data you analyze using AVG 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.

Type of Computation

To compute this type of average ...	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

Result Type and Attributes

The data type, format, and titles for AVG(x) are as follows:

<table>
<tr><th>Data Type</th><th>Format</th><th>Titles</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>
</table>
</td><td>• Moving Avg(x)
• Cumulative Avg(x)
• Group Avg(x)
• Remaining Avg(x)</td></tr>
</table>

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Example: 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, month, sales,
AVG(sales) OVER (PARTITION BY territory
                 ORDER BY month
                 ROWS 2 PRECEDING)
FROM sales_history;
```

Territory	Month	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.

COUNT

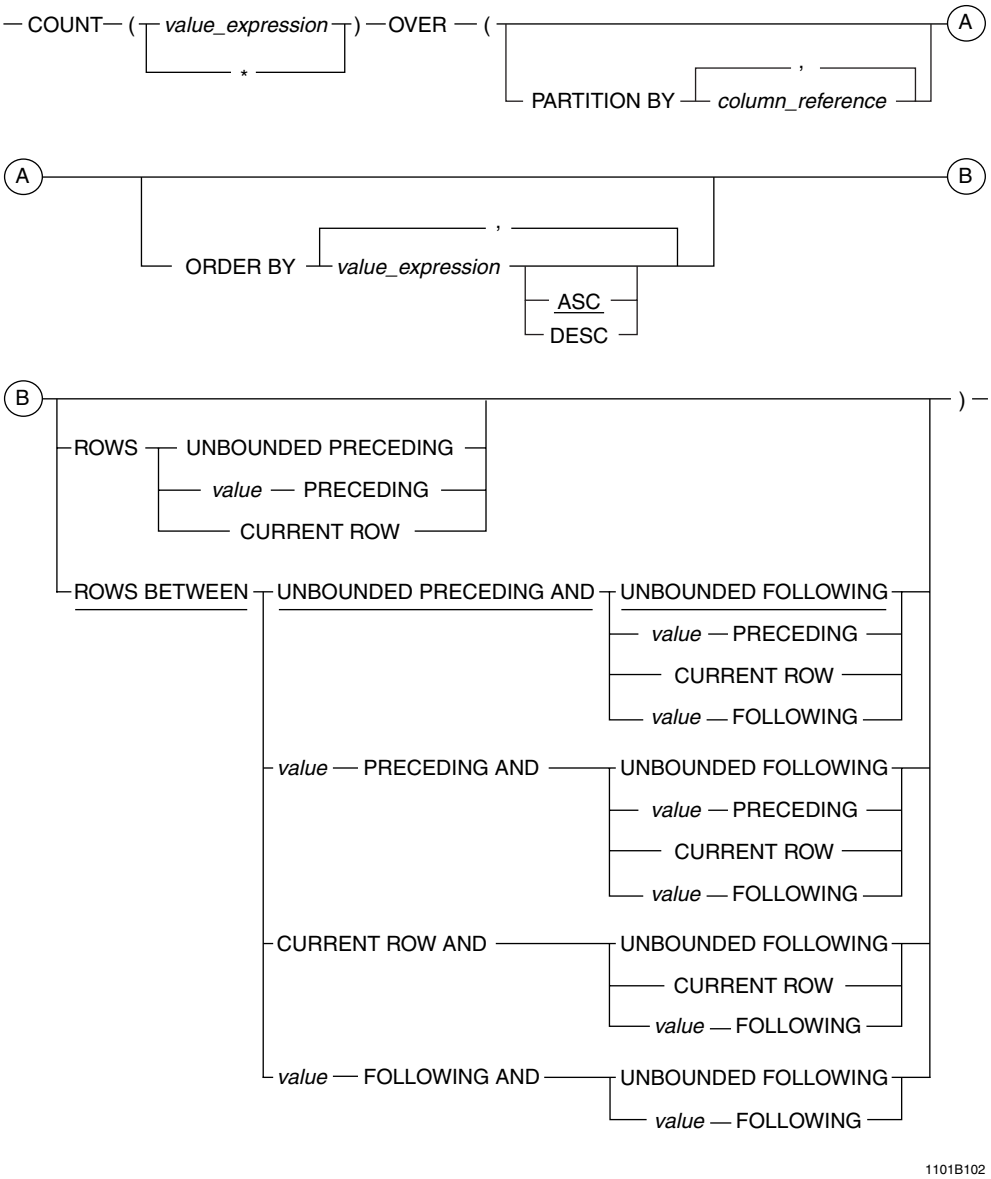
Purpose

Returns the cumulative, group, or moving count for a *value_expression*.

Type

SQL-99 window aggregate function.

Syntax



where:

Syntax element . . .	Specifies . . .
value_expression	the column expression for which a group, cumulative, or moving count is returned.
OVER	the arguments that define how values are grouped, ordered, and considered when computing the cumulative, group, or moving count for the *value_expression*. Values are grouped according to the optional PARTITION BY clause, 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 COUNT function operates. PARTITION BY is optional. If there is no PARTITION BY clause, 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 v*alue_expression* the order in which to sort the values in a group, or partition.
ASC	ascending sort order. The default is ASC.
DESC	descending sort order.
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.

Syntax element . . .	Specifies . . .
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

COUNT is ANSI SQL-99-compliant.

Relationship to the COUNT Aggregate Function

For the description of the COUNT aggregate function, see "COUNT" on page 6-10.

value_expression

COUNT can take a single column name as an argument (sales, for example) or an expression (sales + profit).

It can also take "*" 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;
```

The preceding SELECT might yield the following results, where the group count for all rows is returned for each of the four partitions defined by city and kind:

City	Kind	Sales	Profit	Group Count (*)
LA	canvas	45	320	4
LA	canvas	125	190	4
LA	canvas	125	400	4
LA	canvas	20	120	4
LA	leather	20	40	2
LA	leather	?	?	2
Seattle	canvas	15	30	3
Seattle	canvas	20	30	3
Seattle	canvas	20	100	3
Seattle	leather	35	50	2
Seattle	leather	?	?	2

Type of Computation

To compute this type of count ...	Use any of the following aggregation groups ...
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

To compute this type of count ...	Use any of the following aggregation groups ...
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

Result Type and Attributes

The data type, format, and titles for COUNT(x) are as follows:

Operation	Mode	Data Type	Format	Titles
COUNT(x)	ANSI	DECIMAL(18,0)	Default format for resulting data type	• Moving Count(x) • Cumulative Count(x) • Group Count(x) • Remaining Count(x)
	Teradata	INTEGER		
COUNT(*)	ANSI	DECIMAL(18,0)	Default format for resulting data type	• Moving Count(*) • Cumulative Count(*) • Group Count(*) • Remaining Count(*)
	Teradata	INTEGER		

ANSI mode uses DECIMAL because Teradata tables frequently have a cardinality exceeding the range of INTEGER.

Teradata mode uses INTEGER to avoid regression problems.

For information on data type default formats, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Example 1: 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 2: 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	?	?	?

CSUM

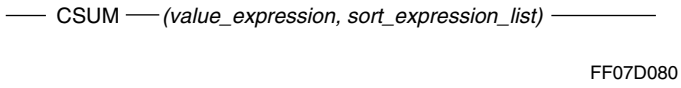
Purpose

Returns the cumulative (or running) sum of a column expression, assuming the rows are sorted by *sort_expression_list*.

Type

Teradata-specific function.

Syntax



FF07D080

where:

Syntax element ...	Specifies ...
value_expression	a scalar numeric column expression for which a running sum is to be computed.
sort_expression_list	column references to be used to sort the values. A *sort_expression_list* is a list of expressions (with optional sort direction specification) separated by commas. 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 default sort direction is ASC.

ANSI Compliance

CSUM is a Teradata extension to the ANSI SQL-99 standard.

Using SUM Instead of CSUM

The use of CSUM is strongly discouraged. It is a Teradata extension to the ANSI SQL-99 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 "SUM" on page 7-74.

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 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>
</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 Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

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

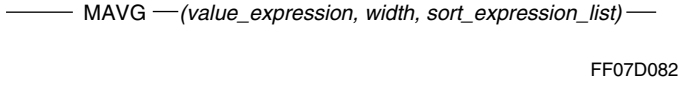
Purpose

Computes the moving average of a column using the current row and the preceding *width*-1 rows.

Type

Teradata-specific function.

Syntax



FF07D082

where:

Syntax element . . .	Specifies . . .
value_expression	a scalar numeric 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_list	a column reference to be used to sort the values. A *sort_expression_list* is a list of expressions (with optional sort direction specification) separated by commas. 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 default sort direction is ASC.

ANSI Compliance

MAVG is a Teradata extension to the ANSI SQL-99 standard.

Using AVG Instead of MAVG

The use of MAVG is strongly discouraged. It is a Teradata extension to the ANSI SQL-99 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 "AVG" on page 7-24.

Result Type and Attributes

The data type, format, and title for MAVG(*x*, *w*, *y direction*) are as follows:

Data Type	Format	Title
Same as operand x	**IF the operand is ...** / **THEN the format is ...** character / the default format for FLOAT. • numeric • date • interval / the same format as x.	MAvg(x, w, y direction)

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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 *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

MAX

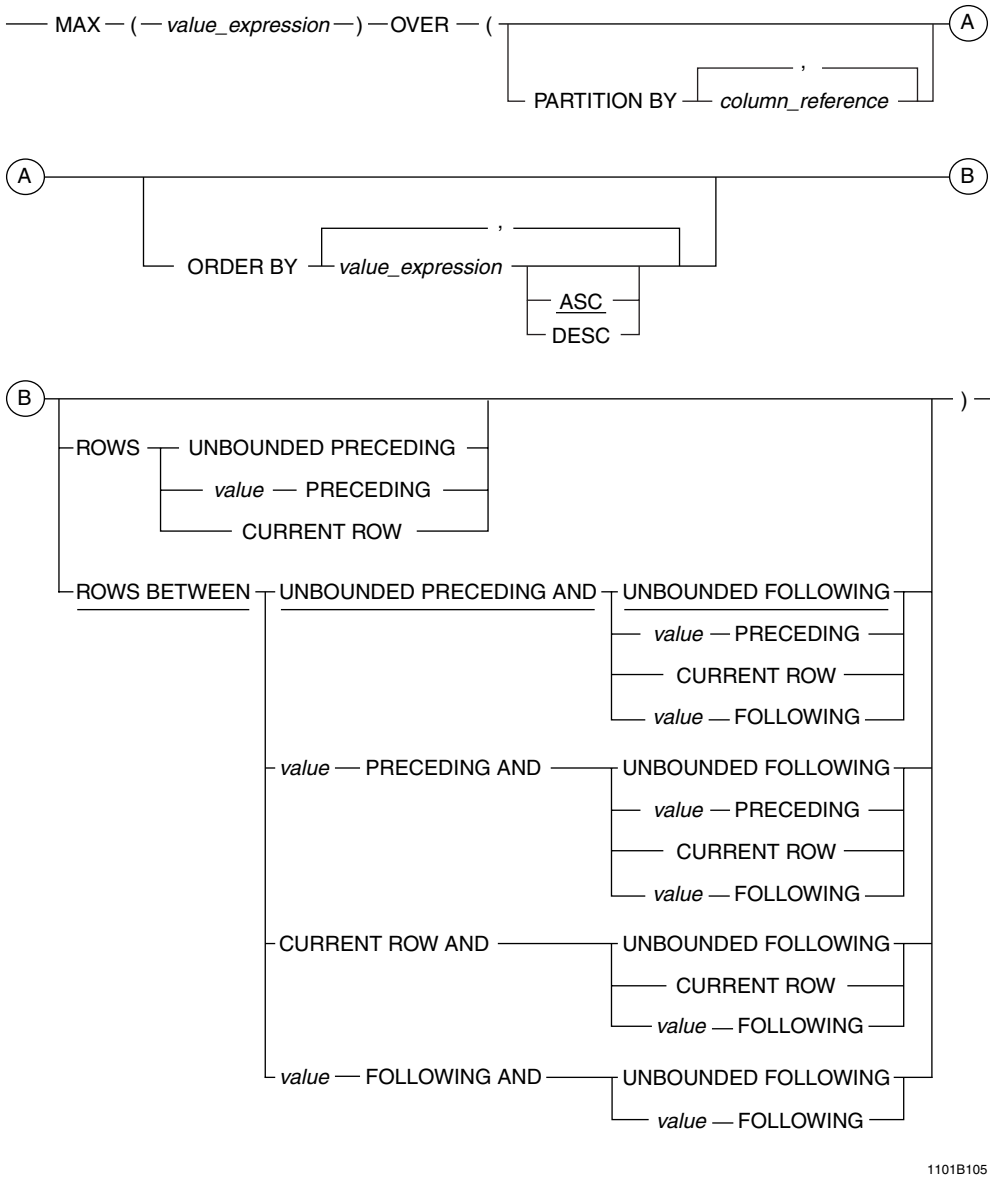
Purpose

Returns the cumulative, group, or moving maximum value for an expression.

Type

SQL-99 window aggregate function.

Syntax



where:

Syntax element …	Specifies …
value_expression	the column expression for which a group, cumulative, or moving maximum value is returned.
OVER	the arguments that define how values are grouped, ordered, and considered when computing the cumulative, group, or moving maximum value for the *value_expression*. Values are grouped according to the optional PARTITION BY clause, 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 MAX function operates. PARTITION BY is optional. If there is no PARTITION BY clause, 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 v*alue_expression* the order in which to sort the values in a group, or partition.
ASC	ascending sort order. The default is ASC.
DESC	descending sort order.
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.

Syntax element . . .	Specifies . . .
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

MAX is ANSI SQL-99-compliant.

Relationship to the MAX Aggregate Function

For the description of the MAX aggregate function, see "MAX" on page 6-22.

value_expression

MAX can take a single column name as an argument (sales, for example) or an expression (sales + profit).

Type of Computation

To compute this type of maximum value…	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

Result Type and Attributes

The data type, format, and titles for the result of MAX(x) are as follows:

Data Type	Format	Titles
Same data type as operand x.	Same format as operand x.	• Group Max(x) • Cumulative Max(x) • Moving Max(x) • Remaining Max(x)

For information on the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Example

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
```

MDIFF

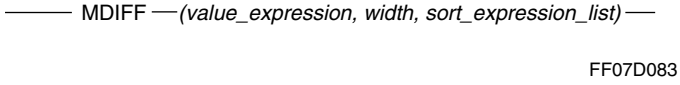
Purpose

Returns the moving difference between the current row-column value and the preceding n^{th} row-column value.

Type

Teradata-specific function.

Syntax



where:

Syntax element . . .	Specifies . . .
value_expression	a scalar numeric column 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_list	a column reference to be used to sort the values. A *sort_expression_list* is a list of expressions (with optional sort direction specification) separated by commas. 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 default sort direction is ASC.

ANSI Compliance

MDIFF is a Teradata extension to the ANSI SQL-99 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_list* 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-99 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 "SUM" on page 7-74.

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 the operand 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 and an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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

MIN

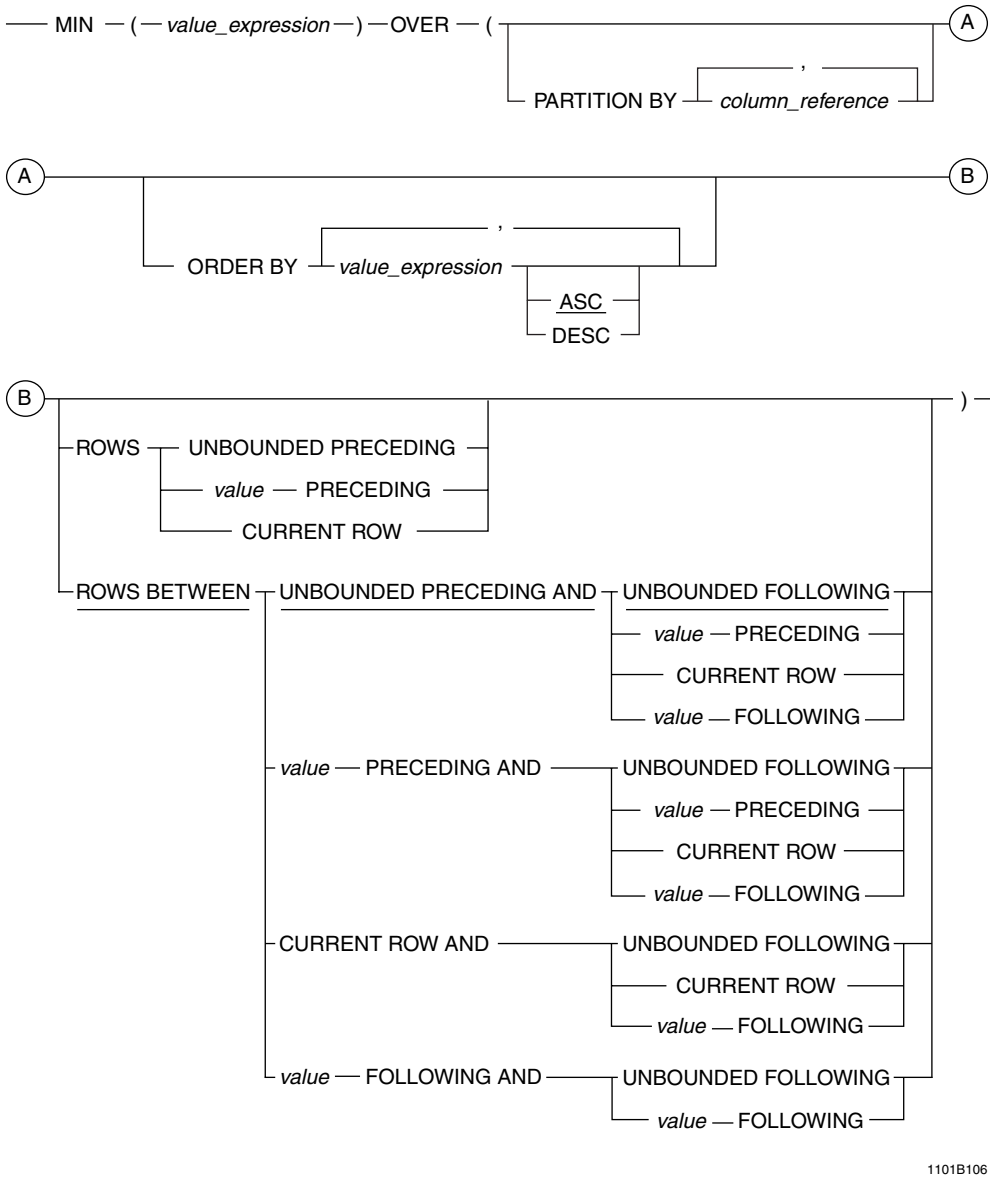
Purpose

Returns the cumulative, group, or moving minimum value for an expression.

Type

SQL-99 window aggregate function.

Syntax



where:

Syntax element . . .	Specifies . . .
value_expression	the column expression for which a group, cumulative, or moving minimum is returned.
OVER	the arguments that define how values are grouped, ordered, and considered when computing the cumulative, group, or moving minimum value for the *value_expression*. Values are grouped according to the optional PARTITION BY clause, 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 MIN function operates. PARTITION BY is optional. If there is no PARTITION BY clause, 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 v*alue_expression* the order in which to sort the values in a group, or partition.
ASC	ascending sort order. The default is ASC.
DESC	descending sort order.
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.

Syntax element . . .	Specifies . . .
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

MIN is ANSI SQL-99-compliant.

Relationship to the MIN Aggregate Function

For the description of the MIN aggregate function, see "MIN" on page 6-24.

value_expression

MIN can take a single column name as an argument (sales, for example) or an expression (sales + profit).

Type of Computation

To compute this type of minimum value ...	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

Result Type and Attributes

The data type, format, and titles for the result of MIN(x) are as follows:

Data Type	Format	Titles
Same data type as operand x.	Same format as operand x.	• Group Min(x) • Cumulative Min(x) • Moving Min(x) • Remaining Min(x)

For information on the default format of data types, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Example

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
```

MLINREG

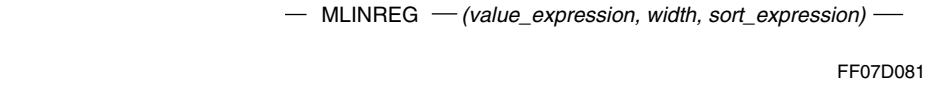
Purpose

Returns a predicted value for a column 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 scalar numeric column expression for which a predicted value is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
width	the number of rows to be used 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 1. The maximum is 4096.
sort_expression	a column reference to be used to sort the values and to define the dependent variable for calculating the linear regression. A *sort_expression* is an expression (with optional sort direction specification). For example, MLINREG(Sale, 6, Region DESC), where Sale is the *value_expression*, 6 is the *width*, and Region DESC is the *sort_expression*. The default sort direction is ASC. Only one *sort_expression* is allowed.

ANSI Compliance

MLINREG is Teradata extension to the ANSI SQL-99 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-99 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:

Data Type	Format	Title
Same as operand x	**IF the operand is ...** / **THEN the format is ...** character / the default format for FLOAT. • numeric • date • interval / the same format as x.	MLinReg(x, w, y direction)

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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

Display the expected sales using past trends for each month for each product using the sales data for the previous six months.

```
SELECT itemID, month, sales, MLINREG(sales,7,month)
FROM sales_table;
GROUP BY itemID;
```

Assume that the relevant columns of sales_table look like the first three columns in the results table, below. Then the moving linear regression computation adds the Mlinreg column (assuming that the row A/1997-01/Null already exists) and returns the following results table:

itemID	month	sales	Mlinreg
A	1996-01	100	?
A	1996-02	110	?
A	1996-03	120	120
A	1996-04	130	130
A	1996-05	140	140
A	1996-06	150	150
A	1996-07	170	160
A	1996-08	190	176.7
A	1996-09	210	198
A	1996-10	230	222
A	1996-11	250	246.7
A	1996-12	270	270
A	1997-01	?	290
B	1996-01	20	?
B	1996-02	30	?
...	...	...	...

MSUM

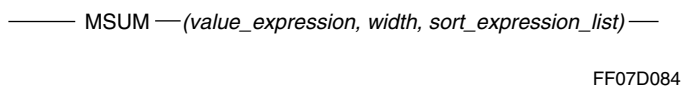
Purpose

Computes the moving sum of a column using the current row and the preceding *n*-1 rows. This function is very similar to the MAVG function.

Type

Teradata-specific function.

Syntax



FF07D084

where:

Syntax element …	Specifies …
value_expression	a scalar numeric 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_list	a column reference to be used to sort the values. A *sort_expression_list* is a list of expressions (with optional sort direction specification) separated by commas. 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*. The default sort direction is ASC.

ANSI Compliance

MSUM is a Teradata extension to the ANSI SQL-99 standard.

Using SUM Instead of MSUM

The use of MSUM is strongly discouraged. It is a Teradata extension to the ANSI SQL-99 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 "SUM" on page 7-74.

Result Type and Attributes

The data type, format, and title for MSUM(*x*, *w*, *y direction*) are as follows:

Data Type	Format	Title
Same as operand x	**IF the operand is ...** / **THEN the format is ...** character / the default format for FLOAT. numeric / the same format as x.	MSum(x, w, y direction)

For information on the default format of data types and an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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

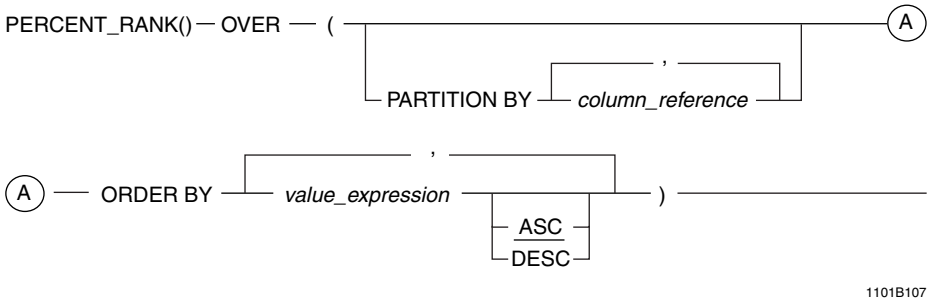
Purpose

Returns the relative rank of rows for a *value_expression*.

Type

SQL-99 window function.

Syntax



where:

Syntax element …	Specifies …
OVER	how the values, grouped according to *column_reference* in the PARTITION BY clause and named by *value_expression* in the ORDER BY clause, are ranked.
PARTITION BY	in its *column_reference* the column, or comma-separated list of columns, according to which ranking resets. PARTITION BY is optional. If there is no PARTITION BY clause, 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.

ANSI Compliance

The PERCENT_RANK window function, which uses ANSI-specific syntax, is ANSI SQL-99-compliant.

Computation

The formula for PERCENT_RANK is:

$$\text{PERCENT\_RANK} = \frac{(RK - 1)}{(NR - 1)}$$

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.
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 Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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

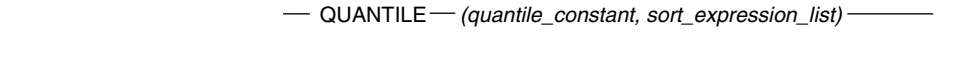
Purpose

Computes the quantile scores for the values in a column.

Type

Teradata-specific function.

Syntax



FF07D085

where:

Syntax element . . .	Specifies . . .
quantile_constant	a positive integer constant used to define the number of quantile partitions to be used.
sort_expression_list	a column reference to be used to sort the values. A *sort_expression_list* is a list of expressions (with optional sort direction specification) separated by commas. For example, QUANTILE(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 default sort direction is DESC.

ANSI Compliance

QUANTILE is a Teradata extension to the ANSI SQL-99 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_list* 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_list* 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-99 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

QUANTILE returns an integer value that represents the quantile of the *sort_expression_list* value of the row relative to the value of 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 and an explanation of the formatting characters in the format, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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

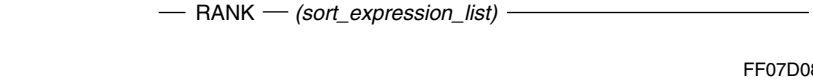
Purpose

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_list	a column reference to be used to sort the values. A *sort_expression_list* is a list of expressions (with optional sort direction specification) separated by commas. For example, RANK(Region ASC, Store DESC), where Region ASC, Store DESC is the *sort_expression_list*. The default sort direction is DESC.

ANSI Compliance

RANK is a Teradata extension to the ANSI SQL-99 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-99 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 7-70.

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 and an explanation of the formatting characters in the format, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

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

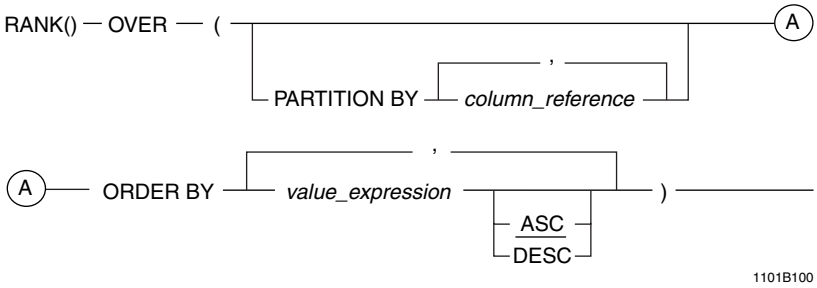
Purpose

Returns an ordered ranking of rows based on the *value_expression* in the ORDER BY clause.

Type

SQL-99 window function.

Syntax



where:

Syntax element . . .	Specifies . . .
OVER	how the values, grouped according to the optional PARTITION BY clause 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 clause, 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.

ANSI Compliance

The RANK window function is ANSI SQL-99-compliant.

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 Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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

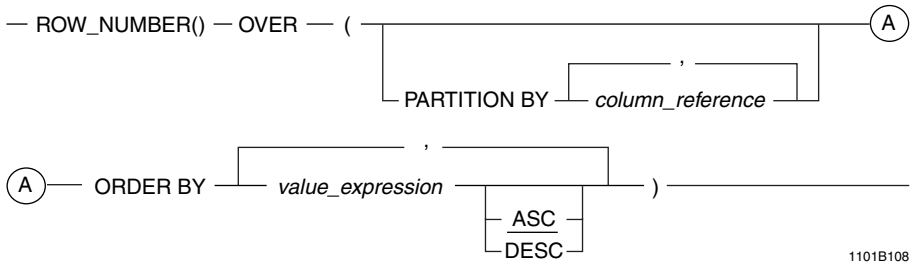
Purpose

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

SQL-99 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 clause, 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 v*alue_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.

ANSI Compliance

The ROW_NUMBER window function is ANSI SQL-99-compliant.

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 "COUNT" on page 7-30.

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
```

SUM

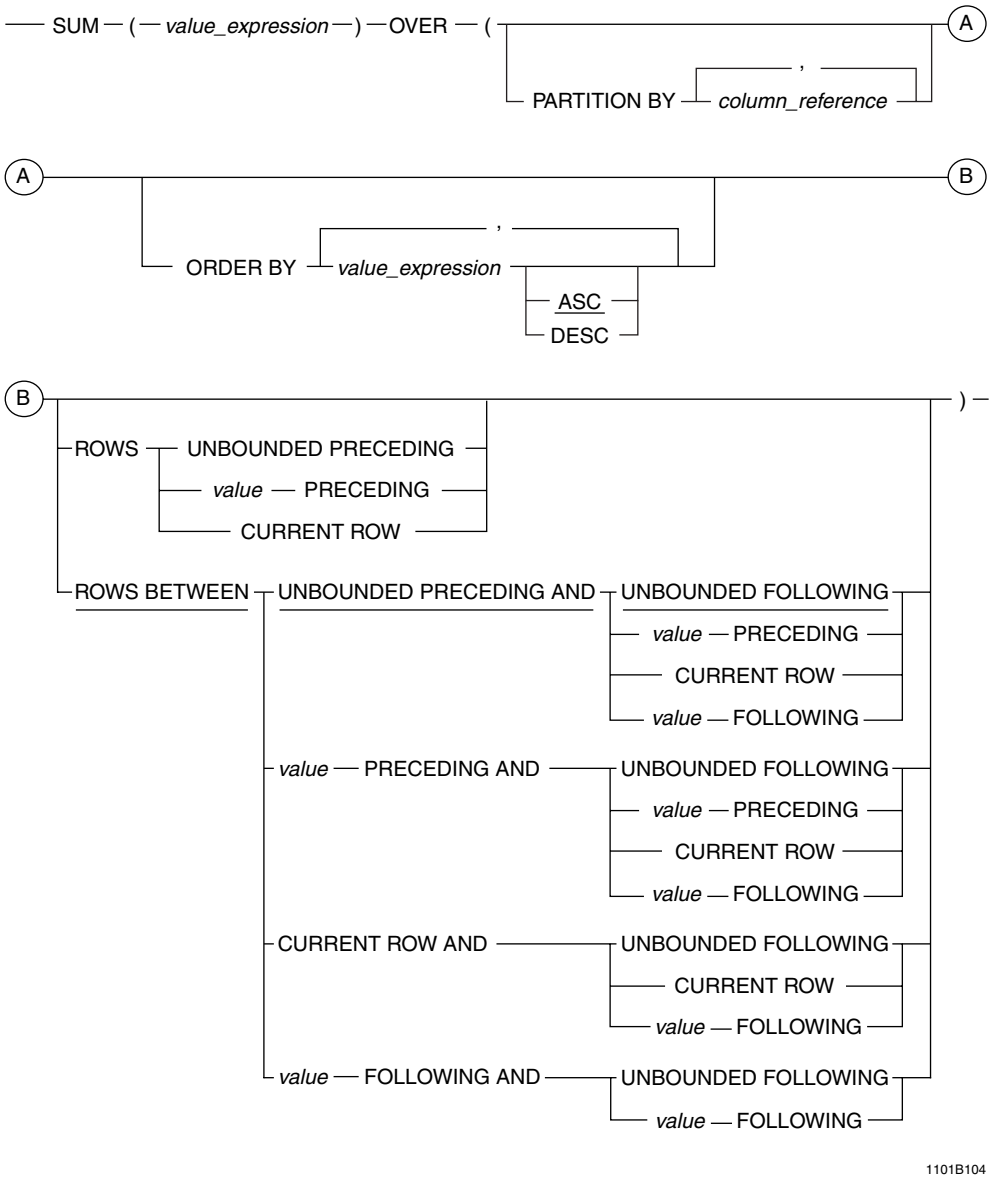
Purpose

Returns the cumulative, group, or moving sum of a *value_expression*, depending on how the aggregation group in the SUM function is specified.

Type

SQL-99 window aggregate function.

Syntax



where:

Syntax element ...	Specifies ...
value_expression	the sum column for which a sum is computed.
OVER	the arguments that define how values are grouped, ordered, and considered when computing the sum for the *value_expression*. Values are grouped according to the optional PARTITION BY clause, ordered according to the ORDER BY clause, and considered according to the aggregation group within the partition.
PARTITION BY	in its *column_reference* the group, or groups, over which the SUM function operates. PARTITION BY is optional. If there is no PARTITION BY clause, 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 order is ASC.
DESC	descending sort order.
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.

Syntax element . . .	Specifies . . .
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

SUM is ANSI SQL-99-compliant.

Using SUM Instead of CSUM and MSUM

Be sure to use the ANSI-compliant SUM window function for any new applications you develop. Avoid using CSUM and MSUM for applications intended to be ANSI-compliant and portable.

The following defines the relationship between the SUM window function and the CSUM and MSUM Teradata-specific functions, respectively:

- The SUM window function that uses the ORDER BY clause and specifies ROWS UNBOUNDED PRECEDING is the ANSI equivalent of CSUM.
- The SUM window function that uses the ORDER BY clause and specifies ROWS *value* PRECEDING is the ANSI equivalent of MSUM.

  Note that the ROWS *value* PRECEDING phrase specifies the number of rows preceding the current row that are used, together with the current row, to compute the moving average. The total number of rows in the aggregation group is *value* + 1. For the MSUM function, the total number of rows in the aggregation group is the value of *width*.

  Thus for the SUM window function that computes a moving sum, an aggregation group of ROWS 5 PRECEDING means that the 5 rows preceding the current row, plus the current row, are used to compute the moving sum. The moving sum for the 6th row of a partition, for example, 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*.

Problems With Missing Data

Ensure that data you analyze using SUM 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.

value_expression

SUM can take a single sum column as an argument (sales, for example) or an expression (sale * profit).

The following is not valid:

```
SUM(*)
```

Type of Computation

To compute this type of sum ...	Use any of these aggregation groups ...
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
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

To compute this type of sum ...	Use any of these aggregation groups ...
Group	• ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
Remaining	• ROWS BETWEEN *value* PRECEDING AND UNBOUNDED FOLLOWING • ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING • ROWS BETWEEN *value* FOLLOWING AND UNBOUNDED FOLLOWING

Result Type and Attributes

The data type, format, and title for SUM(x) are as follows:

Data Type	Format	Title
Same as the operand	IF the operand is ... / THEN the format is ... character / the default format for FLOAT. numeric / the default format for the data type of the operand.	• Group Sum(x) • Cumulative Sum(x) • Moving Sum(x) • Remaining Sum(x)

For an explanation of the formatting characters in the format, and information on data type default formats, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Example 1: 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;
```

The possible results of the above SELECT are shown below:

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 2: 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, kind ROWS BETWEEN UNBOUNDED
PRECEDING AND UNBOUNDED FOLLOWING) FROM monthly;
```

The possible results of the above 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 3: 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 above 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 4: 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 5: 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 above 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 above 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	?

Chapter 8:

String Functions

This chapter describes functions that operate on character, byte, and numeric strings, including:

- Concatenation Operator
- CHAR2HEXINT
- INDEX
- LOWER
- POSITION
- SOUNDEX
- SUBSTRING/SUBSTR
- TRANSLATE
- TRANSLATE_CHK
- TRIM
- UPPER
- VARGRAPHIC

String Functions

Introduction

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

What is a String?

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	• BYTE • VARBYTE	• BYTEINT • DECIMAL • FLOAT • INTEGER • NUMERIC • SMALLINT

ANSI Equivalence of Teradata String Functions

Several of the Teradata string functions are extensions to the ANSI SQL-99 standard.

To maintain ANSI compatibility, use the ANSI equivalent functions instead of Teradata 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 10: "Attribute Functions."
comparison operators	Chapter 3: "Comparison Operators."
the LIKE predicate	Chapter 9: "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.

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.

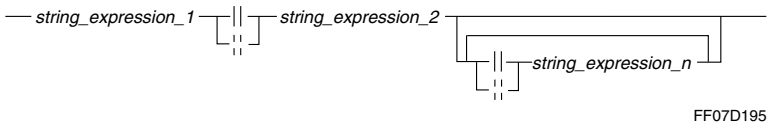
For details, see "Implicit Character-to-Character Translation" on page 13-32.

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-99 standard. Do not use them as concatenation operators.

Solid and broken VERTICAL LINE character pairs (| |) are ANSI SQL-99-compliant forms of the concatenation operator.

Argument Types and Rules

Use the concatenation operator on strings and string expressions of type:

- Byte
- Numeric
- Character

The following rules apply to the arguments:

- 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 Type Conversions" on page 13-3.
- If any argument is a byte type, all other arguments must also be byte types.
- 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 13-32.

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*:

Data Type		Heading
IF the arguments are ...	**THEN the result is a ...**	(*arg1* || *arg2*)
byte strings	byte string.	
numeric or character strings	character string.	

If either argument is null, the result is null.

The data types and attributes of the arguments determine whether the result type of a concatenation operation is a fixed length or varying length string. Result types appear in the following table, where *n* is the sum of the lengths of all arguments:

IF this argument ...	Is this data type or attribute ...	THEN the result is this data type or attribute ...
either	VARBYTE	VARBYTE(*n*)
	VARCHAR	VARCHAR(*n*)
	numeric	
both	BYTE	BYTE(*n*)
	CHARACTER (with same server character set)	CHARACTER(*n*)
	CHARACTER (with different server character sets)	VARCHAR(*n*)
	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.

Table tsttbla defines column_1 as BYTE(2) and column_2 as VARBYTE(10).

The hexadecimal value '4142'XB exists in column_1 and the hexadecimal value '7A7B7C'XB exists in column_2.

```
SELECT column_2 || column_1
FROM tsttbla ;
```

The result is:

```
(column_2||column_1)
------------------------
7A7B7C4142
```

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 13-32.

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 whether or not 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.

These examples assume that the string expressions follow the rules defined in the chapter "SQL Data Definition" in *Teradata RDBMS SQL Reference, Volume 3.*

For an explanation of symbols and other notation in the examples, see "Character Shorthand Notation Used In This Book" on page A-6.

Example 1: KanjiEBCDIC

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
<**ABC**>	<**DEF**>G	<**ABCDEF**>G
<**ABC**>	<>	<**ABC**>
<**ABC**>a	<**DEF**>	<**ABC**>a<**DEF**>

Example 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
```

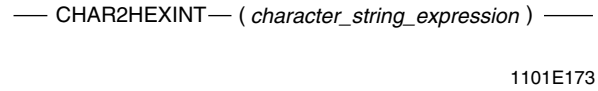
string_expression_1	string_expression_2	Result
mn**ABCX**	**B**	mn**ABCXB**
mn**ABCX**	g	mn**ABCX**g

CHAR2HEXINT

Purposes

Returns the hexadecimal representation for a character string.

Syntax



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-99 standard.

Argument Types

Use CHAR2HEXINT on character strings or character string expressions.

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:

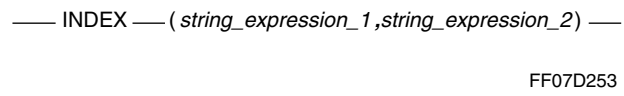
```
Internal Hex Representation of TableName   TableName
----------------------------------------   ----------------
41636365737352696768473                    AccessRights
4163634C6F6752756C6554626C                 AccLogRuleTbl
4163634C6F6754626C                         AccLogTbl
4163636F756E7473                           Accounts
4163637467                                 Acctg
416C6C                                     All
436F70496E666F54626C                       CopInfoTbl
```

INDEX

Purpose

Returns the position in *string_expression_1* where *string_expression_2* starts.

Syntax



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-99 standard.

Use POSITION instead of INDEX for ANSI SQL-99 compliance.

Argument Types and Rules

INDEX operates on the following types of arguments:

- Character
- Byte
- Numeric

The following table describes rules for the data types of the arguments:

IF ...	THEN ...
one string expression is of type BYTE	both string expressions must be of type BYTE.
any string expression is numeric	it is converted implicitly to CHARACTER type. For more information, see "Implicit Type Conversions" on page 13-3.

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 8-17.

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 13-32.

The CASESPECIFIC attribute affects whether characters are considered to be a match.

<table>
<tr><th>IF CASESPECIFIC is ...</th><th>THEN ...</th></tr>
<tr><td>not specified for either expression</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>
<tr><td>specified</td><td>simple Latin letters are considered to be matching only if they are the same letters and the same case.
When ANSI transaction semantics are in effect for the session, CASESPECIFIC is in effect by default.</td></tr>
</table>

Rules for KANJI1 Server Character Set

When the server character set is KANJI1 and the client character set is KanjiEBCDIC, the Shift-Out/Shift-In characters are included in the offset count, but they are not matched. Instead, they are treated only as an indication of a transition between a single byte character and an multibyte character.

The non-zero position of the result is reported as follows:

IF the client character set is ...	THEN the result is ...
KanjiEBCDIC	the byte position corresponding to the first byte of the logical character offset (including Shift-Out/Shift-In in the offset count) within *string_expression_1.*
other than KanjiEBCDIC	the 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 8-24.

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* is different from the server character set for *string_2*. 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
M**X**N<**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 SUBSTRING, INDEX, and concatenation 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.

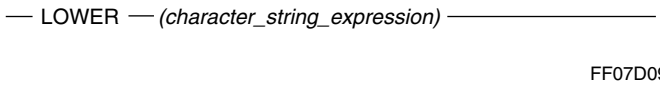
```
empno      Emp Name
---------- --------------
10021      T. Smith
10007      J. Aguilar
10018      S. Russell
10011      M. Chin
10019      P. Newman
```

LOWER

Purpose

Returns a character string identical to *character_string_expression*, except that all uppercase letters are replaced by their lowercase equivalents.

Syntax



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-99-compliant.

Argument Types

Use LOWER on character strings or character string expressions.

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.
- LOWER is supported only for the LATIN server character set. If the type of argument for LOWER is anything other than LATIN, 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.

You can also replace characters with uppercase equivalents. For more information, see "UPPER" on page 8-51.

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.

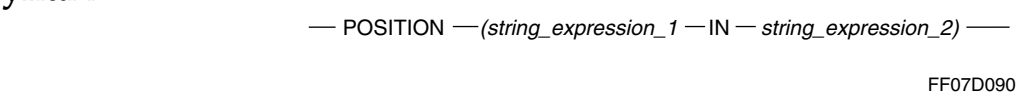
```
USING (last_name CHAR(20),first_name CHAR(20))
INSERT INTO names (LOWER(:last_name), LOWER(:first_name));
```

POSITION

Purpose

Returns the position in *string_expression_2* where *string_expression_1* starts.

Syntax



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-99-compliant.

Use POSITION instead of INDEX for ANSI SQL-99 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
- Byte
- Numeric

The following table describes rules for the data types of the arguments:

IF ...	THEN ...
one string expression is of type BYTE	both string expressions must be of type BYTE.
any string expression is numeric	it is converted implicitly to CHARACTER type. For a description of this process, see "Implicit Type Conversions" on page 13-3.

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.

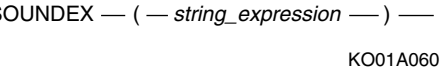
- INDEX counts Shift-Out and Shift-In characters; POSITION does not.
- INDEX counts bytes; POSITION counts logical characters. As a result, an **A**, for example, counts as two bytes (two *physical* characters) for INDEX, but only one *logical* character for POSITION.

SOUNDEX

Purpose

Returns a character string that represents the Soundex code for *string_expression*.

Syntax



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-99 standard.

Argument Types

Use SOUNDEX on character strings or character string expressions that use the LATIN or UNICODE server character set.

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 table below outlines the Soundex coding guide:

Stage	Process
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 occurence 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 8-29 use the following table data.

family
family_name
John
Joan
Joey
joanne
michael
Bob

```
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 Non-Valid Usage

The following SOUNDEX examples are not valid for the reasons given in the table.

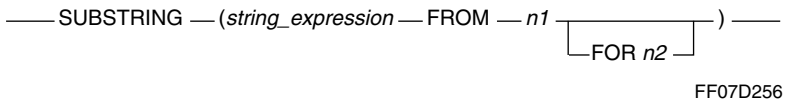
Statement	Why the Statement is Not Valid
`SELECT SOUNDEX(12345);`	12345 is a numeric string, not a character string.
`SELECT SOUNDEX('ábç');`	The characters á and ç are not simple Latin characters.

SUBSTRING/SUBSTR

Purpose

Extracts a substring from a named string based on position.

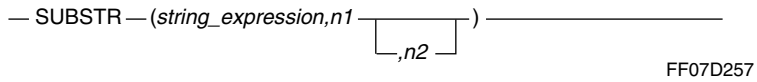
ANSI Syntax



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 be extracted 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*.
n2	the length of the substring to be extracted from *string_expression*.

Teradata Syntax



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 be extracted from *string_expression*.
n2	the length of the substring to be extracted from *string_expression*.

ANSI Compliance

SUBSTRING is ANSI SQL-99-compliant.

SUBSTR is a Teradata extension to the ANSI SQL-99 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. For more information, see "Implicit Type Conversions" on page 13-3.

Result Type and Attributes

Here are the default result type and attributes for SUBSTR(*string*, *n1*, *n2*) and SUBSTRING(*string* FROM *n1* FOR *n2*):

Data Type		Heading
IF the *string* argument is a ...	**THEN the result type is ...**	Substring(*string* From *n1* For *n2*) Substr(*string*, *n1*, *n2*)
byte string	VARBYTE.	
numeric or character string	VARCHAR.	

Result Value and Length for Character Strings

For a character string, SUBSTRING/SUBSTR extracts *n2* characters from *string_expression* starting at position *n1.*

In ANSI mode, the length of the result is always set to the length of the original string.

In Teradata mode, the length is governed by the rules described in the following table.

IF n2 is ...	THEN ...	
specified	**IF ...**	**THEN ...**
	n2 < 0	SUBSTRING/SUBSTR returns an error.
	0 ≤ *n2* and *n1* > *string_length*	SUBSTRING/SUBSTR returns a zero length string.

<table>
<tr><th>IF n2 is ...</th><th>THEN ...</th></tr>
<tr><td>specified
(cont'd)</td><td>
<table>
<tr><th>IF ...</th><th>THEN ...</th></tr>
<tr><td>0 ≤ n2 and n1 < 1</td><td>SUBSTRING/SUBSTR sets n2 = n2 + n1 - 1 and sets n1 = 1.
<table>
<tr><th>IF ...</th><th>THEN SUBSTRING/SUBSTR returns ...</th></tr>
<tr><td>n2 <= 0</td><td>a zero length string.</td></tr>
<tr><td>n2 > string_length</td><td>the source string.[1]</td></tr>
<tr><td>0 < n2 <= string_length</td><td>a string that starts at n1 and extends for n2 characters.[2]</td></tr>
</table></td></tr>
<tr><td>0 < n2 and
1 <= n1 <= string_length</td><td>
<table>
<tr><th>IF ...</th><th>THEN SUBSTRING/SUBSTR returns ...</th></tr>
<tr><td>(n1+n2 - 1) > string_length</td><td>a string that starts at n1 and ends with the last character of the source string.[3]</td></tr>
<tr><td>0 < (n1+n2 - 1) <= string_length</td><td>a string that starts at n1 and extends for n2 characters.[2]</td></tr>
</table></td></tr>
</table></td></tr>
<tr><td>not specified</td><td>
<table>
<tr><th>IF ...</th><th>THEN SUBSTRING/SUBSTR returns ...</th></tr>
<tr><td>n1 < 1</td><td>the source string. [1]</td></tr>
<tr><td>n1 > string_length</td><td>a zero length string.</td></tr>
<tr><td>1 <= n1 <= string_length</td><td>a string that starts at n1 and ends with the last character of the source string.[3]</td></tr>
</table></td></tr>
</table>

Note	Length of Result in Teradata Mode
1	*string_length*
2	*n2*
3	*string_length* - *n1* + 1

Usage Rules for SUBSTRING and SUBSTR

SUBSTRING is the ANSI SQL-99 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
	`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.

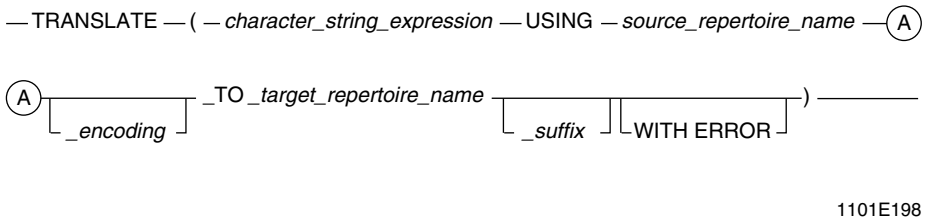
IF c1 contains ...	THEN this query ...	Returns ...
mn**ABCX**	`SELECT SUBSTR(c1, 6, 1) FROM ctable1;`	**X**
	`SELECT SUBSTR(c1,4) FROM ctable1;`	**BCX**

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 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 Types" on page 8-38.
_encoding	an optional literal for translating from KANJI1 to UNICODE that indicates a specific encoding of KANJI1.
target_repertoire_name	the target character set of the string to be translated. For supported values, see "Supported Translations Between Types" on page 8-38.

IF the translation is from this character set ...	THEN use this value for *_encoding* ...
• KatakanaEBCDIC • KanjiEBCDIC5026_0I • KanjiEBCDIC5038_0I	_KanjiEBCDIC
KanjiEUC_0U	_KanjiEUC
KanjiShiftJIS_0S	_KANJISJIS
ASCII or EBCDIC	_SBC

Syntax element . . .	Specifies . . .
_suffix	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
WITH ERROR	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 8-39).

ANSI Compliance

TRANSLATE is ANSI SQL-99-compliant.

Argument Types

Use TRANSLATE on character strings or character string expressions.

Result Type and Attributes

The default attributes for TRANSLATE (*string* USING *source*_TO_*target*) are:

Data Type	Heading
VARCHAR(*n*) CHARACTER SET *target* where *source*_TO_*target* determines the character set or type of *target*, according to the supported translations in "Supported Translations Between Types" on page 8-38.	Translate(*string* USING *source*_to_*target*)

Supported Translations Between Types

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_repertoire_name*_TO_*target_repertoire_name*	Source Character Set	Target Character Set
LATIN_TO_GRAPHIC	LATIN	GRAPHIC
LATIN_TO_KANJISJIS	LATIN	KANJISJIS
LATIN_TO_UNICODE	LATIN	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_LATIN	KANJISJIS	LATIN
KANJISJIS_TO_GRAPHIC	KANJISJIS	GRAPHIC
KANJISJIS_TO_UNICODE	KANJISJIS	UNICODE
GRAPHIC_TO_LATIN	GRAPHIC	LATIN
GRAPHIC_TO_KANJISJIS	GRAPHIC	KANJISJIS
GRAPHIC_TO_UNICODE	GRAPHIC	UNICODE
GRAPHIC_TO_UNICODE_PadSpace	GRAPHIC	UNICODE
UNICODE_TO_LATIN	UNICODE	LATIN
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_GRAPHIC	UNICODE	GRAPHIC
UNICODE_TO_KANJISJIS	UNICODE	KANJISJIS
UNICODE_TO_UNICODE_Fullwidth	UNICODE	UNICODE
UNICODE_TO_UNICODE_Halfwidth	UNICODE	UNICODE
UNICODE_TO_UNICODE_FoldSpace	UNICODE	UNICODE
UNICODE_TO_GRAPHIC_PadGraphic	UNICODE	GRAPHIC
UNICODE_TO_GRAPHIC_VarGraphic	UNICODE	GRAPHIC

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	non-KANJISJIS
• KANJI1_KanjiEBCDIC_TO_UNICODE • UNICODE_TO_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	non-KanjiEUC
• KANJISJIS_TO_LATIN • GRAPHIC_TO_LATIN • UNICODE_TO_LATIN • UNICODE_TO_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 8-36.

This form of translation is also useful for migrating object names. For information, see "Migration" on page 8-41.

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_repertoire_name_TO_target_repertoire_name	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.
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.

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.

Pad Character Translation

The following translations do not translate the pad character:

source_repertoire_name_TO_target_repertoire_name	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, then you can use one of the following translations:

source_repertoire_name_TO_target_repertoire_name	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, GRAPHIC data must be translated from the old form 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 treats constants as UNICODE, and then translates the string from LATIN to KANJISJIS. However, the translation generates an error because the last character of the string 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

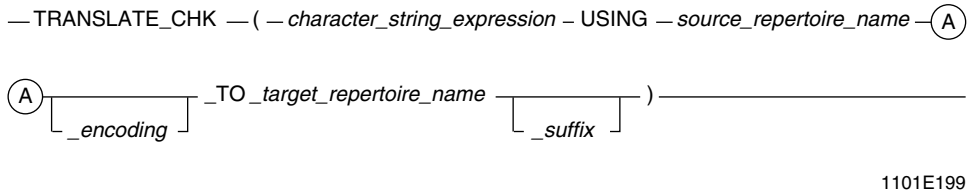
Function	Result	Type of the Result
TRANSLATE('abc' USING UNICODE_TO_LATIN)	'abc'	VARCHAR(3) CHARACTER SET LATIN
TRANSLATE('abc' USING UNICODE_TO_UNICODE_Fullwidth)	'abc'	VARCHAR(3) CHARACTER SET UNICODE
TRANSLATE('abc年' USING UNICODE_TO_LATIN WITH ERROR) where ε represents the designated error character for LATIN (0x1A).	'abcε'	VARCHAR(4) CHARACTER SET LATIN

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 Types" on page 8-38.
_encoding	an optional literal for translating from KANJI1 to UNICODE that indicates a specific encoding of KANJI1.

IF the translation is from this character set ...	THEN use this value for *_encoding* ...
• KatakanaEBCDIC • KanjiEBCDIC5026_0I • KanjiEBCDIC5038_0I	_KanjiEBCDIC
KanjiEUC_0U	_KanjiEUC
KanjiShiftJIS_0S	_KANJISJIS
ASCII or EBCDIC	_SBC

Syntax element . . .	Specifies . . .
target_repertoire_name	the target character set of the string to be translated. For supported values, see "Supported Translations Between Types" on page 8-38.
_suffix	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

ANSI Compliance

TRANSLATE_CHK is a Teradata extension to the ANSI SQL-99 standard.

Argument Types

Use TRANSLATE_CHK on character strings and character string expressions.

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.

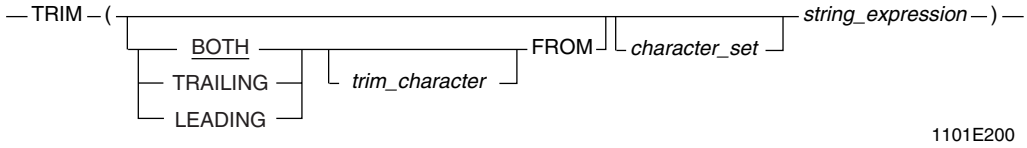
```
SELECT ckanji1
FROM table_1
WHERE TRANSLATE_CHK(ckanji1 USING Kanji1_KanjiEUC_TO_Unicode) <> 0;
```

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:

Syntax Element . . .	Specifies . . .
BOTH TRAILING LEADING	how the specified trim character or byte is to be trimmed from *string_expression*. The keywords and their meanings are described in the following table. **Keyword** / **Meaning** BOTH — Both trailing and leading characters or bytes are to be trimmed. TRAILING — Only trailing characters or bytes are to be trimmed. LEADING — Only leading characters or bytes are to be trimmed. If you omit this option, the default is BOTH, and the default trim character is a null byte for byte types and a pad character for character types.
trim_character	the character or byte to be trimmed from the head, tail, or both, of *string_expression*. When you specify BOTH, TRAILING, or LEADING, you must specify *trim_character.* You cannot specify a *trim_character* of type KANJI1, nor can you apply a *trim_character* to a *string_expression* of type KANJI1.
FROM	a keyword required when BOTH, TRAILING, or LEADING are specified.
string_expression	a byte or character string or string expression to be trimmed.

ANSI Compliance

TRIM is ANSI SQL-99-compliant.

Argument Types and Rules

TRIM operates on the following types of *string_expression* arguments:

- Character
- Byte
- Numeric

If a numeric expression is used as the *string_expression* argument, it is converted implicitly to CHARACTER type. For more information, see "Implicit Type Conversions" on page 13-3.

The *trim_character* argument can only be a byte or character type.

Result Type and Attributes

Here are the default result type and attributes for TRIM(*string*):

Data Type		Heading
IF the *string* argument is ...	**THEN the result type is ...**	Trim(BOTH FROM *string*)
a byte string	VARBYTE.	
a numeric or character string	VARCHAR.	

It is possible for the length of the result to be zero.

The server character set of the result is the same as the argument.

If the *string* 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'
```

Example 2

If the TRIM function is removed, the statement:

```
SELECT last_name || ', ' || first_name
FROM names;
```

returns trailing blanks in the string:

```
'Jones       , Mary        '
```

Example 3

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 4

The following statement trims trailing SEMICOLON characters from the specified string.

```
SELECT TRIM( TRAILING ';' FROM textfield) FROM texttable;
```

Example 5

The following table illustrates several more complicated TRIM functions*:*

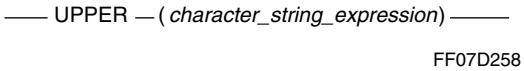
Function	Result
SELECT TRIM(LEADING 'a' FROM 'aaabcd');	'bcd'
SELECT TRIM(_Unicode 'ΔΔabc年ΔΔΔ');	'abc年'
SELECT TRIM(_Unicode 'ΔΔabc年ΔΔΔ');	'abc年ΔΔ' Δ (GRAPHIC pad) is not removed.
CREATE TABLE t1 (c1 CHARACTER(6) CHARACTER SET GRAPHIC); INSERT t1 (_Graphic 'abc年ΔΔ'); SELECT TRIM(c1) from t1;	'abc年' Δ (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



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-99-compliant.

Argument Types

UPPER is valid only for character strings and character string expressions.

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 8-21.

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 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');
```

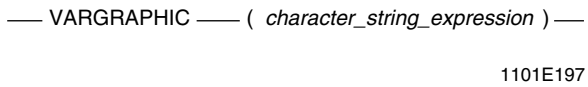
The result is 'ABCD年'.

VARGRAPHIC

Purpose

Returns the VARGRAPHIC representation of the character data in *character_string_expression*.

Syntax



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-99 standard.

Argument Types

VARGRAPHIC operates on the following types of arguments:

- Character
- Numeric

If the argument is numeric, it is implicitly converted to a character type. For more information, see "Implicit Type Conversions" on page 13-3.

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 8-56.

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.

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

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
32	2	42F2	42F2
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

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	**JIS X 0201 Printable Character**	**KanjiEBCDIC 5026/5035**	**Katakana EBCDIC**
4E	N	42D5	42D5
4F	O	42D6	42D6
50	P	42D7	42D7
51	Q	42D8	42D8
52	R	42D9	42D9
53	S	42E2	42E2
54	T	42E3	42E3
55	U	42E4	42E4
56	V	42E5	42E5
57	W	42E6	42E6
58	X	42E7	42E7
59	Y	42E8	42E8
5A	Z	42E9	42E9
5B	[	4444	FEFE
5C	\	425B	425B
5D	]	4445	FEFE
5E	^	4470	425F
5F	_	426D	426D
60	‘	4279	FEFE
61	a	4281	FEFE
62	b	4282	FEFE
63	c	4283	FEFE
64	d	4284	FEFE
65	e	4285	FEFE
66	f	4286	FEFE
67	g	4287	FEFE
68	h	4288	FEFE
69	i	4289	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	**JIS X 0201 Printable Character**	**KanjiEBCDIC 5026/5035**	**Katakana EBCDIC**
6A	j	4291	FEFE
6B	k	4292	FEFE
6C	l	4293	FEFE
6D	m	4294	FEFE
6E	n	4295	FEFE
6F	o	4296	FEFE
70	p	4297	FEFE
71	q	4298	FEFE
72	r	4299	FEFE
73	s	42A2	FEFE
74	t	42A3	FEFE
75	u	42A4	FEFE
76	v	42A5	FEFE
77	w	42A6	FEFE
78	x	42A7	FEFE
79	y	42A8	FEFE
7A	z	42A9	FEFE
7B	{	42C0	FEFE
7C	|	424F	424F
7D	}	42D0	FEFE
7E	-[e]	42A1	42A1
7F		FEFE	FEFE
80		FEFE	FEFE
81		FEFE	FEFE
82		FEFE	FEFE
83		FEFE	FEFE
84		FEFE	FEFE
85		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	**JIS X 0201 Printable Character**	**KanjiEBCDIC 5026/5035**	**Katakana EBCDIC**
86		FEFE	FEFE
87		FEFE	FEFE
88		FEFE	FEFE
89		FEFE	FEFE
8A		FEFE	FEFE
8B		FEFE	FEFE
8C		FEFE	FEFE
8D		FEFE	FEFE
8E		FEFE	FEFE
8F		FEFE	FEFE
90		FEFE	FEFE
91		FEFE	FEFE
92		FEFE	FEFE
93		FEFE	FEFE
94		FEFE	FEFE
95		FEFE	FEFE
96		FEFE	FEFE
97		FEFE	FEFE
98		FEFE	FEFE
99		FEFE	FEFE
9A		FEFE	FEFE
9B		FEFE	FEFE
9C		FEFE	FEFE
9D		FEFE	FEFE
9E		FEFE	FEFE
9F		FEFE	FEFE
A0		FEFE	FEFE
A1	[f]	4341	4341

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
A2	g	4342	4342
A3	h	4343	4343
A4	i	4344	4344
A5	j	4345	4345
A6	k	4346	4346
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

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
BE	SEE	438F	438F
BF	SO	4390	4390
C0	TAI	4391	4391
C1	CHI	4392	4392
C2	TSU	4393	4393
C3	TE	4394	4394
C4	TO	4395	4395
C5	NA	4396	4396
C6	NI	4397	4397
C7	NU	4398	4398
C8	NE	4399	4399
C9	NO	439A	439A
CA	HA	439D	439D
CB	HI	439E	439E
CC	FU	439F	439F
CD	HE	43A2	43A2
CE	HO	43A3	43A3
CF	MA	43A4	43A4
D0	MI	43A5	43A5
D1	MU	43A6	43A6
D2	ME	43A7	43A7
D3	MO	43A8	43A8
D4	YA	43A9	43A9
D5	YU	43AA	43AA
D6	YO	43AC	43AC
D7	RA	43AD	43AD
D8	RI	43AE	43AE
D9	RU	43AF	43AF

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
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
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

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/5035	Katakana EBCDIC
F6		FEFE	FEFE
F7		FEFE	FEFE
F8		FEFE	FEFE
F9		FEFE	FEFE
FA		FEFE	FEFE
FB		FEFE	FEFE
BC		FEFE	FEFE
FD		FEFE	FEFE
FE		FEFE	FEFE
FF		FEFE	FEFE

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 9:

Logical Predicates

This chapter describes SQL logical predicates, including:

- BETWEEN/NOT BETWEEN
- ANY/ALL
- EXISTS/NOT EXISTS
- IN/NOT IN
- IS NULL/IS NOT NULL
- LIKE
- OVERLAPS

Logical Predicates

Definition

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 a column name, a constant, or an arithmetic expression.

Logical predicates are also referred to as conditional expressions or search conditions. 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 whose 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 "Comparison Operators" under "Supported Comparison Operators" on page 3-3.	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.

Case Blind Comparisons

Introduction

Case blind comparisons can be accomplished using the UPPER function, applied to any character string value that is not assured to contain no lower case 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 8-51.

Case Sensitivity in Comparisons

All character data accessed in the execution of a SQL statement has an attribute of CASESPECIFIC or NOT CASESPECIFIC, either as a default or by explicit designation.

NOT CASESPECIFIC is not an ANSI-compatible attribute. ANSI SQL does *all* character comparisons as the equivalent of CASESPECIFIC.

As a result, all string comparisons done in ANSI mode are case specific even if the columns being compared are defined as NOT CASESPECIFIC. The only way to ensure that string comparisons are not casespecific is to explicitly declare the relevant columns in your requests to be not casespecific using the (NOT CASESPECIFIC) attribute.

Provision is made at session logon to declare a session as operating in ANSI or Teradata mode.

IF a session is in this mode . . .	THEN the default for character data in a CREATE or ALTER statement is . . .
ANSI	CASESPECIFIC
Teradata (GRAPHIC characters only)	
Teradata (all other characters)	NOT CASESPECIFIC

If a session is in ANSI mode, then the default for character data in a CREATE or ALTER statement is CASESPECIFIC. If Teradata mode, the default is NOT CASESPECIFIC.

This attribute persists unless the column definition is modified by a later ALTER statement.

Character literal strings in queries, view and macro definitions, CHECK constraints, and data accessed via the USING phrase have defaults that are determined at session logon, unless explicitly modified. The default attribute exists only for the life of the session.

IF a session is in this mode . . .	THEN all character string literals are . . .
ANSI	CASESPECIFIC
Teradata (GRAPHIC characters only)	
Teradata (all other characters)	NOT CASESPECIFIC

Comparison Rule for Two Character Arguments

The comparison rule for two character arguments is as follows.

IF this argument . . .	IS defined in this way . . .	THEN the comparison is . . .	FOR . . .
either	CASESPECIFIC	case specific	all cases.
both	NOT CASESPECIFIC	case blind	<simple Latin Letters>

Because of the long persistence of the CASESPECIFIC attribute for columns, and the shorter persistence for attributes carried in the source text of queries, the same statement can have different results for a user logged on in ANSI or Teradata mode.

Example

The following example illustrates the possible results for different case comparisons.

```
SELECT *
FROM STUDENTS
WHERE FIRSTNAME = 'George';
```

IF column FIRSTNAME is . . .	AND the session is in this mode . . .	THEN the match succeeds for rows with FIRSTNAME containing . . .
CASESPECIFIC	either	'George'
NOT CASESPECIFIC	ANSI	'George'
NOT CASESPECIFIC	Teradata	'george', 'George', 'GEORGE', and anything else that matches.

To assure case blind comparisons and have an ANSI compliant application, define FIRSTNAME with CASESPECIFIC, the ANSI default.

DATE Operand

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 5-27.

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';
```

Comparisons Using Subqueries

Definition

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.

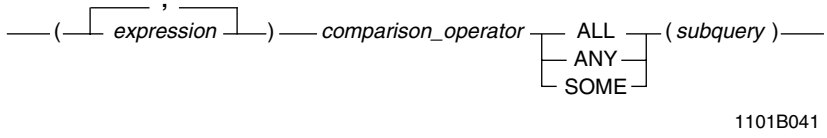
Comparison Operators and Subqueries

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.

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:

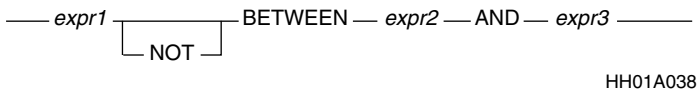
```
SELECT EmpNo, Name, DeptNo, JobTitle, Salary, YrsExp
FROM Employee
WHERE (Salary,YrsExp) >= ALL
 (SELECT Salary,YrsExp FROM Employee) ;
```

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-99-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.
```

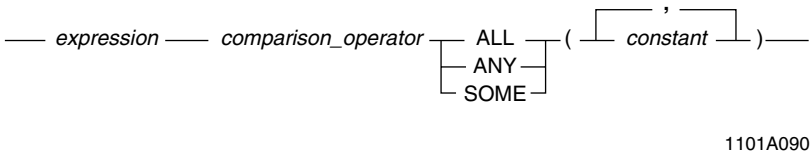
This expression . . .	Is equivalent to . . .
x BETWEEN y AND z	((x >= y) AND (x <=z))

ANY/ALL/SOME

Definition

Allows one or more expressions to be compared with one or more constants or with one or more values returned by a subquery.

Syntax



ANSI Compliance

ANY, SOME, and ALL are ANSI SQL-99-compliant quantifier predicates.

Usage

IF this quantifier is specified . . .	THEN the result is true when . . .
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	

The following table explains the consequences of these results.

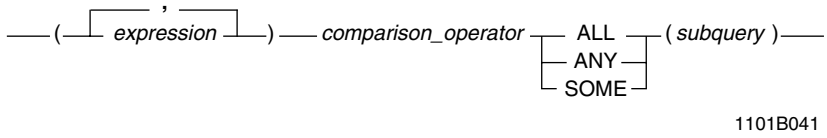
This expression . . .	Is equivalent to . . .
`x < ALL (:a, :b, :c)`	`(x < :a) AND (x < :b) AND (x < :c)`
`x > ANY (:a,:b,:c)`	`(x > :a) OR (x > :b) OR (x > :c)`
`x > SOME (:a,:b,:c)`	

Implicit conversion rules are the same as for the comparison operators.

If *expression* evaluates to NULL, the result is considered to be unknown.

Subquery Syntax

When the following syntax is used, the subquery must select the same number of expressions as are specified in the expression list.



Quantifier and Subqueries

If a quantifier is used with a subquery, the subquery may return none, one, or several values.

The results are determined as follows:

The following expression ...	Is ...	WHEN ...
expression > ALL *subquery*	TRUE	*expression* is greater than every value in the set of values returned by subquery. It is also TRUE if subquery returns no values.
	TRUE	*subquery* returns no values.
expression > ANY *subquery*	TRUE	*expression* is greater than at least one value of the set of values returned by subquery.
	FALSE	*subquery* returns no values.

For example.

```
SELECT EmpNo,Name,JobTitle,Salary,YrsExp
FROM Employee
WHERE (Salary,YrsExp) >= ALL
 (SELECT Salary,YrsExp FROM Employee) ;
```

Example

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) ;`

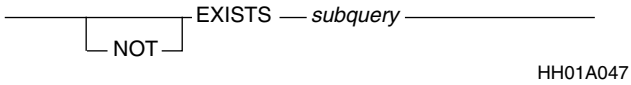
EXISTS/NOT EXISTS

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



HH01A047

ANSI Compliance

EXISTS and NOT EXISTS are ANSI SQL-99-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.

Relationship Between EXISTS/NOT EXISTS and IN/NOT IN

EXISTS predicate is used to test 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. There are problems that can be solved only by using EXISTS and/or NOT EXISTS predicate. For an example, see "For ALL" on page 9-16.

For information on IN and NOT IN, see "IN/NOT IN" on page 9-20.

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 which 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.

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 Chapter 1: ”The SELECT Statement” in *Teradata RDBMS SQL Reference, Volume 6* .

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 9-17, the following rows are returned:

```
SName          SNo
-----          ---
Helen Chu      1
```

IN/NOT IN

Purpose

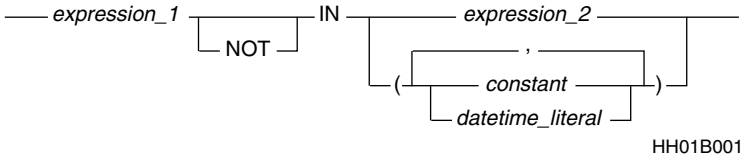
Tests the existence of the value of an expression or expression list in a comparable set in one of two ways:

- 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-99-compliant.

Syntax 1: *expression* IN and NOT IN *expression* or constants



HH01B001

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*.
IN NOT IN	whether the test is inclusive or exclusive. **You can substitute …** / **FOR …** • IN ANY, • = ANY / IN • <> ALL, • ALL / NOT IN
expression_2	the value in which the existence of *expression_1* is to be tested.

Syntax element ...	Specifies ...
constant	• constant • macro parameter • built-in value such as TIME or DATE You can specify a maximum of 512 constants for an IN clause.
datetime_literal	an ANSI DateTime literal.

Usage

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_n* data types and the *constant* values must be compatible. Implicit conversion rules are the same as for the comparison operators.

If the value of *expression_n* is null, then the result is considered to be unknown.

If the set of constants consists of 70 or fewer members, the system uses hashing to retrieve the rows. If the set consists of more than 70 members, the system does a full-table scan. Note that full-table scans are much more time consuming than row hashed accesses.

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 the expression is not equal to any values in the list of constants.

THIS form ...	Is valid in this mode ...	AND means ...
expression_1 IN *expression_2*	Teradata	*expression_1* = *expression_2*
expression_1 NOT IN *expression_2*	• Teradata • ANSI	*expression_1* <> *expression_2*

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 9-14.

Example

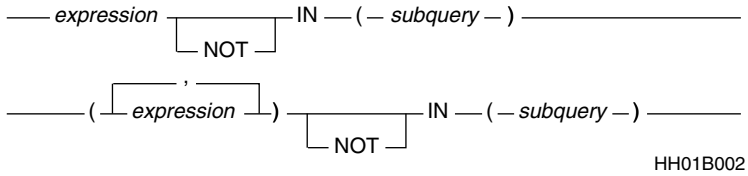
This statement . . .	Is equivalent to this statement . . .

`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);`

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.*
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.

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.

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') ;
```

[NOT] IN Clauses and Stored Procedures

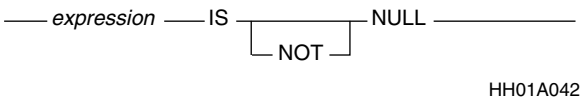
You cannot specify a [NOT] IN clause in a stored procedure conditional expression if that expression also references an alias for a local variable, parameter, or cursor.

IS NULL/IS NOT NULL

Purpose

Searches for or excludes nulls in an expression.

Syntax



ANSI Compliance

IS NULL and IS NOT NULL are ANSI SQL-99-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 be listed 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) ;
```

Logical Operators and Search Conditions

Purpose

Specify the criteria for logically producing the result of a search condition.

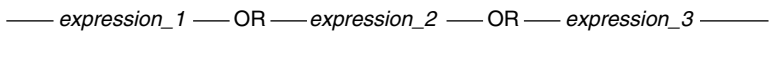
Definition: Logical Operator

An operator applied to the result of a predicate to determine the result of a search condition.

The logical operators are:

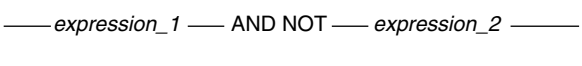
- AND
- NOT
- OR

For example:



FF07D220

Use NOT to negate an expression, for example:



FF07D221

Definition: Search Condition

A search condition, or conditional expression, consists of one or more conditional terms connected by one or more logical predicates like any the following.

- 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 as the operand in a WHERE or HAVING clause.

When used in a HAVING clause, a logical expression can be used with an aggregate operator.

For example:

```
WHERE a=column_1
GROUP BY a
HAVING AVG(b) BETWEEN column_2 AND column_3
```

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

As final Boolean results, the following expressions evaluate as explained in the table.

Result	Evaluation
NOT TRUE	FALSE
NOT FALSE	TRUE
UNKNOWN	FALSE
NOT UNKNOWN	FALSE

AND Truth Table

The following table illustrates the AND logic used in evaluating search conditions.

Keep in mind that a logical result of UNKNOWN resolves to FALSE.

	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.

Keep in mind that a logical result of UNKNOWN resolves to FALSE.

	x FALSE	x UNKNOWN	x TRUE
y FALSE	FALSE	UNKNOWN	TRUE
y UNKNOWN	UNKNOWN	UNKNOWN	TRUE
y TRUE	TRUE	TRUE	TRUE

Examples of Search Conditions

The following examples illustrate the use of 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

Example 3

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:

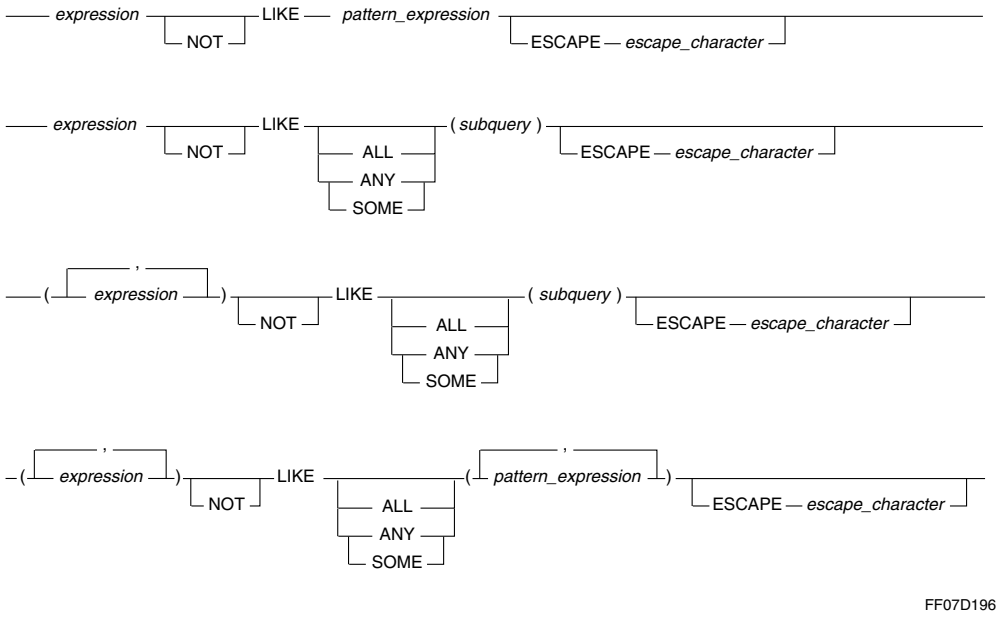
```
SELECT AVG(Salary)
FROM Employee
WHERE DeptNo IN (100,300,500,600)
GROUP BY DeptNo
HAVING AVG(Salary) BETWEEN 35000 AND 55000 ;
```

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.
quantifier	the quantifier argument is one of the keywords ANY (or SOME) or ALL. SOME is a synonym for ANY.
subquery	a SELECT statement argument.
ESCAPE *escape_character*	keyword/variable combination specifying a single escape character (single or multibyte).

ANSI Compliance

LIKE is ANSI SQL-99-compliant.

Optimized Performance Using a NUSI

Under certain conditions, the Optimizer can elect to evaluate a LIKE expression by scanning a NUSI instead of the base table. For this optimization to occur, the following conditions must be in place.

- The *expression* string must be fewer than 16 bytes.
- The NUSI must contain only one column.
- There must be statistics collected for both the base table primary index and for the NUSI column against which the *expression* string is evaluated.
- The *expression* string must be either the mode or max value in at least one interval in the base table statistics histogram.

If these conditions are not met, then the Optimizer specifies an all-rows scan to evaluate the expression.

expression

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.

If neither *pattern_expression* nor *expression* has been designated CASESPECIFIC, any lowercase letters are converted to uppercase before the comparison operation occurs.

If either *expression* or *pattern_expression* has been designated CASESPECIFIC, letters are considered matched only if they are the same letters and the same case.

pattern_expression

The pattern_expression, which can be any character string, must always be placed to the right of the LIKE operator.

The % and _ characters may be used in any combination in *pattern_expression*.

The % (PERCENT SIGN) character represents any string of zero or more arbitrary characters.

The _ (LOW LINE) character represents exactly one arbitrary character.

Any single character is acceptable in the position in which the underscore character appears, and any string of characters is acceptable as a replacement for the percent. However, 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*.

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 extends the permissible metacharacter set for the LIKE predicate to include the fullwidth underscore and the fullwidth percent sign.

FOR this server character set ...	THIS metacharacter ...	MATCHES 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 used in the pattern string it must be immediately followed by either 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 occurs immediately followed by another instance of the 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 ESCAPE character occurs immediately followed by an underscore metacharacter, the sequence is treated as a single underscore character (not a wildcard character).
- When an ESCAPE character occurs immediately followed by a percent metacharacter, 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 the Teradata RDBMS 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.

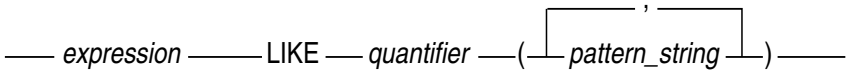
Both leading and trailing pad characters in a pattern are significant to the matching rules.

Example 4:

LIKE 'ΔΔZ%' locates only those fields that start with two pad characters followed by Z.

quantifier

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%'`

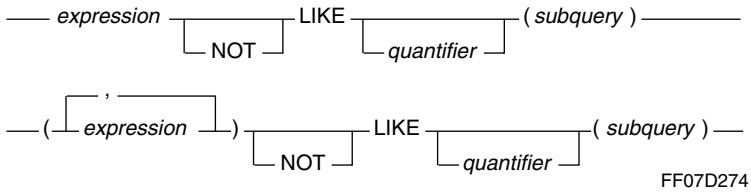
The following statement selects from the employee table the row of any employee whose job title includes the characters “Pres” or begins with the characters “Man”:

```
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.

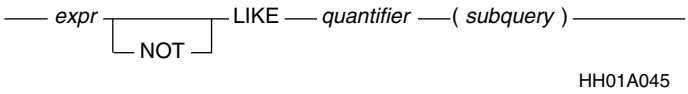
Example

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 meta character or by the appropriate character data type or another *escape_character*.

The resultant two-character sequence matches a single character if *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 data type, 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 "Character Data Type Assignment Rules" on page 13-31.

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
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.

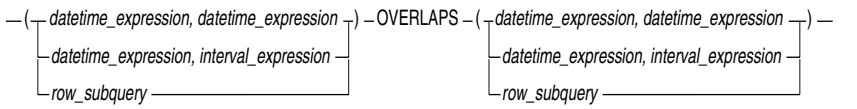
_ represents a SPACING UNDERSCORE.

OVERLAPS

Purpose

Tests whether two time periods overlap one another.

Syntax



FF07D164

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.

ANSI Compliance

OVERLAPS is ANSI SQL-99-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

Each time period represents a start and end DateTime, either using explicit 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*.

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))
```

Rules

The following rules apply to the OVERLAPS comparison.

- When you specify two DateTime types, they must be comparable.
- Both first columns of each left and right element must have the same data type: both DATE, both TIME, or both TIMESTAMP.
- Each second column of each left and right element must either be the same DateTime data type as its corresponding first column (in other words, the two types must be compatible) or it must be an Interval data type that involves only DateTime fields whose precision is such that its value can be added to that of the corresponding DateTime data 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.

Chapter 10:

Attribute Functions

This chapter describes SQL attribute functions, including:

- BYTES
- CHARACTER_LENGTH
- CHARACTERS
- FORMAT
- OCTET_LENGTH
- TYPE

Attribute Functions

Introduction

Attribute functions return descriptive information about their operand. 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 "Core Data Type Attributes" under "Data Type Phrases" in the chapter "SQL Data Definition" in *Teradata RDBMS SQL Reference, Volume 3*.

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-99 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:

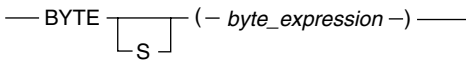
- BYTES
- FORMAT
- TYPE

BYTES

Purpose

Returns the number of bytes contained in the specified byte string.

Syntax



FF07D174

where:

Syntax element ...	Specifies ...
byte_expression	the byte string for which the number of bytes is to be returned. The data types for *byte_expression* are restricted to BYTE or VARBYTE. BYTES does not process strings defined with other data types.

ANSI Compliance

BYTES is a Teradata extension to the ANSI SQL-99 standard.

Usage Notes

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 to suppress the trailing double zero bytes in a value.

The syntax for using TRIM with the BYTES function is as follows.



FF07D215

For more information on TRIM, see "TRIM" on page 8-47.

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:

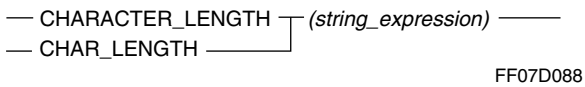
```
BadgePic         Bytes(BadgePic)
--------------   ---------------
20003BA0                       4
9A3243F805                     5
EEFF08C3441900                 7
```

CHARACTER_LENGTH

Purpose

Returns the length of a string either in logical characters or in bytes.

Syntax



where:

Syntax element …	Specifies …
string_expression	the string expression for which the length is to be returned.

ANSI Compliance

CHARACTER_LENGTH is ANSI SQL-99-compliant.

Usage Notes

CHARACTER_LENGTH is the ANSI form of the Teradata CHARACTERS function. Use CHARACTER_LENGTH instead of CHARACTERS for ANSI SQL-99 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.)

CHARACTER_LENGTH returns results as follows.

IF the input string has this type …	THEN the result is returned as the string length in …
CHARACTER	logical characters
anything else	bytes

For all character data types except KANJI1, CHARACTER_LENGTH returns the length of *string_expression* in characters. For Kanji1 character data type, the following results are obtained.

FOR this 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 character set of the current session. A mix of single and multibyte characters is expected.
ASCII EBCDIC	the length of *string_expr* as the number of bytes.

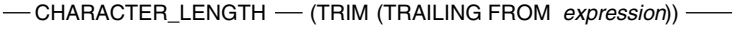
The type of *string_expression* must be CHARACTER. For non-CHARACTER data types, the function returns an error.

Because trailing pad characters are considered characters, 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 characters, including any trailing pad characters, contained in that value.

If you do not want trailing pad characters included in the character count for a data value, use the TRIM function to suppress the trailing pad characters in a string.

The syntax for using TRIM with CHARACTER_LENGTH is as follows.



FF07D216

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 character data type . . .	AND example . . .	CHARACTER_LENGTH returns . . .
GRAPHIC	ABC	3
KANJI1	De<**MNP**>	10
	<><>	4

Example Set 2: KanjiShift-JIS

FOR this character data type . . .	AND example . . .	CHARACTER_LENGTH returns . . .
KANJI1	<><>	10
	De**F**	3
UNICODE	ABC	3
GRAPHIC	ABC	3

Example Set 3: KanjiEUC

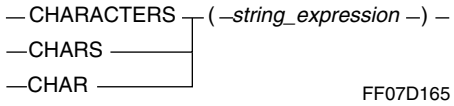
FOR this character data type . . .	AND example . . .	CHARACTER_LENGTH returns . . .
KANJI1	ss_3**C**ss_3**D**	2
GRAPHIC		2
UNICODE	<><>	0
	d**A** ss_2B ss_3**E**	4
LATIN	ABC	3

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.

ANSI Compliance

CHARACTERS is a Teradata extension to the ANSI SQL-99standard.

Value Returned by CHARACTERS Depends on Character Type

Because CHARACTERS returns the number of logical characters or bytes in *string_expression*, the value can differ depending on the character data type of *string_expression*. The following table illustrates the differences among the various character data types when a column is defined as CHARACTER(12).

FOR this character data type ...	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 types.

Use CHARACTER_LENGTH Instead of CHARACTERS

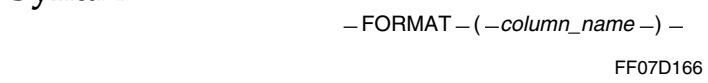
Use of the CHARACTERS function is deprecated. Instead, use the ANSI-equivalent “CHARACTER_LENGTH.”

FORMAT

Purpose

Returns the declared format for the named expression.

Syntax



FF07D166

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-99 standard.

Expected Data Type Returned by FORMAT

The data type returned by a FORMAT phrase is a variable character string of up to 30 characters.

FORMAT clause date separator characters need not match the separator characters used on input.

Example

The following statement requests the format of the Salary column in the Employee table.

```
SELECT FORMAT(Salary) FROM Employee;
```

The result is the following.

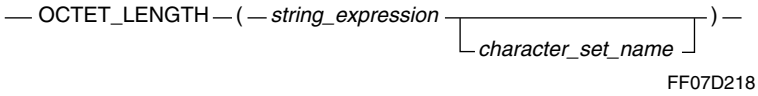
```
FORMAT(Salary)
ZZZ,ZZ9.99 ...
```

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. See the list of Teradata-provided character sets in the table on page 10-11. If *character_set_name* is not provided, the session character set is assumed.

ANSI Compliance

OCTET_LENGTH is ANSI SQL-99-compliant.

Usage Notes

Any Shift-Out/Shift-In and trailing GRAPHIC pad characters are included in the result count.

OCTET_LENGTH operates in the same manner in both Teradata and ANSI modes.

IF *string_expression* is . . .	THEN . . .
of type KANJI1	the result is independent of *character_set_name*.
not CHARACTER data	an error is generated.

The following table lists the character sets shipped with the Teradata system. While these character sets are shipped with the system, they must be installed individually by your system administrator to become available for use.

Your site might also have user-defined sets. Check with your system administrator for a complete list of character sets available at your site.

Character Set	Where Found
ASCII	Built in
EBCDIC	
LATIN1_0A	DBC.CharTranslations
LATIN9_0A	
EBCDIC037_0E	
KATAKANAEBCDIC	kanjiinst.bteq
KANJIEBCDIC5026_0I	
KANJIEBCDIC5035_0I	
KANJIEUC_0U	
KANJISJIS_0S	

Example

Examples of output from the OCTET_LENGTH function are given in the following table.

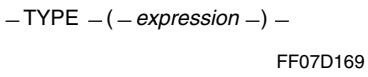
Character Set	Character Data Type	string_expression	Result
EBCDIC	LATIN	abcdefgh	8
ASCII	KANJI1	abcdefgh	8
KanjiEBCDIC	KANJI1	AB<**CDE**>P	11
KanjiEBCDIC	GRAPHIC	MNOP	8 (record mode)
			10 (field mode)
KanjiEUC	KANJISJIS	d**A** ss_2B ss_3**E**	8
KanjiShift-JIS	KANJISJIS	**D**e**F**	5
KanjiShift-JIS	UNICODE	ABC	6

TYPE

Purpose

Returns the data type defined for an expression.

Syntax



FF07D169

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-99 standard.

Usage Notes

TYPE returns a character string that contains the name of the data type of the expression. For a list of the supported data types, see the summary of data type declarations in "Data Type Phrases" in the chapter "SQL Data Definition" in *Teradata RDBMS SQL Reference, Volume 3.*

TYPE does not return information on range constraints.

If expression is of type CHARACTER, then the result of function includes the server character set (if different from the user default character set). For examples, see "Example 1" and "Example 2" on page 10-13.

IF the character data type has this relation to the user default character type ...	THEN the type is returned as follows ...
the same	an unqualified varying string of type LATIN.
different	a fully qualified varying string of the type specified for the column.

If *expression* is a SUBSTR expression, and you are in ANSI mode, then TYPE returns the data type with the maximum length of the first operand of the SUBSTR expression. For an example, see "Example 5" on page 10-13.

The rules for Teradata mode are more complicated. For details, see "Result Type and Attributes" on page 8-31.

Example 1

If the user default character set is LATIN, then an expression defined as CHARACTER(3) CHARACTER SET LATIN returns the result CHARACTER(3).

Example 2

If the user default character set is LATIN, but the expression being analyzed is a UNICODE expression of varying length 11, then the value VARCHAR(11) CHARACTER SET UNICODE is returned.

Example 3

If the user-defined character set is LATIN and the following TYPE function is used to request the data type defined for the Name and Salary columns of the Employee table, then the data returned is shown in the immediately following table.

```
SELECT TYPE(Name), TYPE(Salary)
FROM Employee;

TYPE(Name)   TYPE(Salary)
-----------  ------------
VARCHAR(12)  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(n) CHARACTER SET GRAPHIC  VARCHAR(n) 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(Name,3,2))
FROM EMPLOYEE;
```

The returned result is as follows:

```
TYPE(SUBSTR(Name,3,2))
----------------------
VARCHAR(12)
```

Chapter 11:

Hash-Related Functions

This chapter describes the following hash-related functions:

- HASHAMP
- HASHBAKAMP
- HASHBUCKET
- HASHROW

What are Hash-Related Functions?

Definition

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

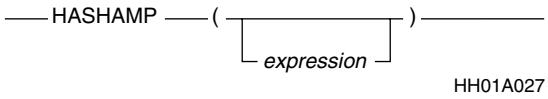
Use the hash-related functions to identify the statistical properties of the current primary index, or to evaluate these properties for other columns to determine their suitability as a future primary index. The statistics can help you to minimize hash synonyms and enhance the uniformity of data distribution.

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. The expression must result in an INTEGER value greater than or equal to 0 and less than or equal to 65535. If *expression* cannot be implicitly converted to an INTEGER, or if *expression* is a character string that does not represent a number, an error is reported. For information on obtaining a hash bucket number that you can use for *expression*, see "HASHBUCKET" on page 11-7.

ANSI Compliance

HASHAMP is a Teradata extension to the ANSI SQL-99 standard.

Result

IF *expression* ...	THEN ...
evaluates to a valid hash bucket number	HASHAMP determines the primary 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.
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 *Teradata RDBMS Utilities*.

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.

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.

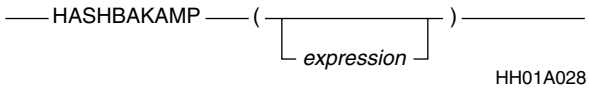
```
SELECT HASHAMP (HASHBUCKET (HASHROW (column_1,column_2))),
COUNT (*)
FROM T
GROUP BY 1
ORDER BY 1;
```

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. The expression must result in an INTEGER value greater than or equal to 0 and less than or equal to 65535. If *expression* cannot be implicitly converted to an INTEGER, or if *expression* is a character string that does not represent a number, an error is reported.

ANSI Compliance

HASHBAKAMP is a Teradata extension to the ANSI SQL-99 standard.

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 *Teradata RDBMS Utilities*.

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.

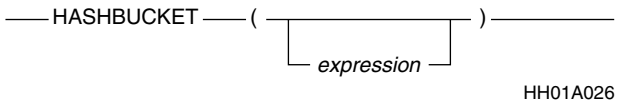
```
SELECT B, HASHBAKAMP (B)
FROM T
ORDER BY 1;
```

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. For information on obtaining a row hash value that you can use for *expression*, see "HASHROW" on page 11-9.

ANSI Compliance

HASHBUCKET is a Teradata extension to the ANSI SQL-99 standard.

Result

HASHBUCKET returns an INTEGER data type.

IF *expression* ...	THEN ...
does not appear in the argument list	HASHBUCKET returns an INTEGER value that is the highest hash bucket number.
evaluates to NULL	HASHBUCKET returns NULL.
evaluates to a valid BYTE(4) row hash value	HASHBUCKET returns the hash bucket number corresponding to the row hash value. Hash bucket numbers can have a value from 0 to 65535.

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, which is 65535. Hash buckets range from 0 to 65535.

```
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.

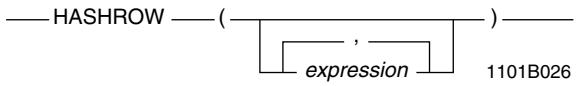
```
SELECT HASHAMP (HASHBUCKET (HASHROW (C1, C2))),
   HASHBUCKET (HASHROW (C1,C2))
   FROM T
   GROUP BY 1,2
   ORDER BY 1,2 ;
```

HASHROW

Purpose

Returns the hexadecimal row hash value for an expression or sequence of expressions. If no expression is specified, HASHROW returns the maximum value that can be generated by applying the hash function on an unsigned integer.

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.

ANSI Compliance

HASHROW is a Teradata extension to the ANSI SQL-99 standard.

Result

The resulting row hash value is typed BYTE(4).

IF *expression* or a list of *expressions* ...	THEN ...
does not appear in the argument list	HASHROW returns the maximum hash code value.
evaluates to NULL	HASHROW returns '00000000'XB.
is a valid, non-NULL expression or list of expressions that can appear in the expression list of the select clause of a SELECT statement	HASHROW evaluates *expression* or the list of *expressions* and applies the hash function on the result. HASHROW returns the resulting row hash value.

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 '0'XB to 'FFFFFFFF'XB.

You can embed a HASHROW call within a HASHBUCKET call. For information on HASHBUCKET, see "HASHBUCKET" on page 11-7.

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 datefld and timefld 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 12:

Built-In Functions

This chapter describes built-in functions, including:

- ACCOUNT
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DATABASE
- DATE
- PROFILE
- ROLE
- SESSION
- TIME
- USER

What are Built-In Functions?

Definition

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.

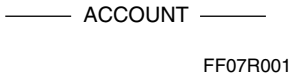
If a SELECT statement that contains a built-in function references a table name, then the result of the query contains one row for every row of the table that satisfies the search condition.

ACCOUNT

Purpose

Returns the account string for the current user.

Syntax



FF07R001

ANSI Compliance

ACCOUNT is a Teradata extension to the ANSI SQL-99standard.

Properties

The value for ACCOUNT has the following characteristics:

- Data type: VARCHAR
- Length: 30
- Not nullable

Usage Note

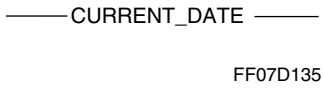
If a SET SESSION ACCOUNT statement has changed the current account string, then the ACCOUNT function returns the new account string based on the request level: whether for an entire session or for an individual request.

CURRENT_DATE

Purpose

Returns the current system date.

Syntax



ANSI Compliance

CURRENT_DATE is ANSI SQL-99-compliant.

Properties

The value for CURRENT_DATE has the following characteristics:

- Data type: DATE
- Length: 4
- Not nullable

CURRENT_DATE Is Similar to 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 12-10.

Output Format

The default output format of CURRENT_DATE is the same as the default output format of the DATE type when the Dateform mode is set to IntegerDate.

To change the output format of the CURRENT_DATE result, use the FORMAT output phrase. For example, the following statement requests the current date and specifies a format that is different from the default:

```
SELECT CURRENT_DATE (FORMAT 'E4,BM4BDD,BYYYY');
```

The result looks like this:

```
                        Date
--------------------------
Friday, September 06, 2002
```

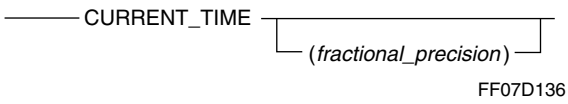
For more information on default data type output formats and the FORMAT output phrase, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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-99-compliant.

CURRENT_TIME Is Similar to 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 12-15.

Properties

The value for CURRENT_TIME has the following characteristics:

- Data type: TIME WITH TIME ZONE
- Length: 8
- Not nullable

CURRENT_TIME Fields

The fields in CURRENT_TIME are:

- HOUR
- MINUTE
- SECOND
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

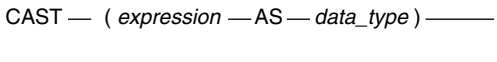
Precision

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.

Casting CURRENT_TIME

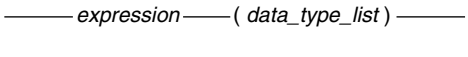
You can convert CURRENT_TIME using Teradata explicit conversion syntax or ANSI CAST syntax:

- ANSI SQL-99-compliant CAST



HH01A021

- Teradata explicit conversion



HH01A019

For more information on the syntax, see "CAST in Explicit Data Type Conversions" on page 13-8 and "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 13-11.

Using FORMAT to Change the Default Output Format

To change the default output format of the CURRENT_TIME result, use the FORMAT output phrase. For example, the following statement requests the current time and specifies a format that is different from the default:

```
SELECT CURRENT_TIME (FORMAT 'hhh:mim:sss');
```

The result looks like this:

```
Current Time(0)
---------------
    07h:32m:25s
```

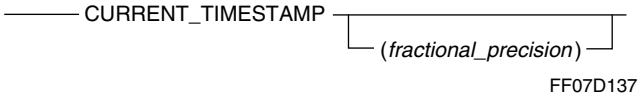
For more information on default data type output formats and the FORMAT output phrase, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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-99-compliant.

Properties

The value for CURRENT_TIMESTAMP has the following characteristics:

- Data type: TIMESTAMP WITH TIME ZONE
- Length: 12
- Not nullable

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

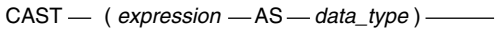
The fields in CURRENT_TIMESTAMP are:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

Casting CURRENT_TIMESTAMP

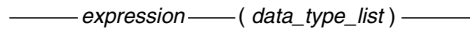
You can convert CURRENT_TIMESTAMP using Teradata explicit conversion syntax or ANSI CAST syntax:

- ANSI SQL-99-compliant CAST



HH01A021

- Teradata explicit conversion



HH01A019

For more information on the syntax, see "CAST in Explicit Data Type Conversions" on page 13-8 and "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 13-11.

Using FORMAT to Change the Default Output Format

To change the default output format of the CURRENT_TIMESTAMP result, use the FORMAT output 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
```

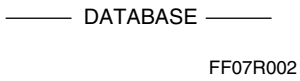
For more information on default data type output formats and the FORMAT output phrase, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

DATABASE

Purpose

Returns the name of the default database for the current user.

Syntax



FF07R002

ANSI Compliance

DATABASE is a Teradata extension to the ANSI SQL-99 standard.

Properties

The value for DATABASE has the following characteristics:

- Data type: VARCHAR
- Length: 30
- Not nullable

Usage Note

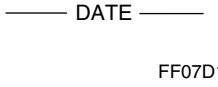
If a DATABASE request has changed the current default database, then the DATABASE function returns the new name of the default.

DATE

Purpose

Returns the current date.

Syntax



FF07D134

The default format of DATE depends on the value of the Dateform mode.

IF the value of the Dateform mode is ...	THEN the default format of the DATE function is ...
INTEGERDATE	the default format for DATE data types as specified in the SDF.
ANSIDATE	'YYYY-MM-DD'

ANSI Compliance

DATE is a Teradata extension to the ANSI SQL-99 standard.

For the ANSI-compliant syntax and behavior for the equivalent function, see "CURRENT_DATE" on page 12-4.

Properties

The value for DATE has the following characteristics:

- Data type: DATE
- Length: 4
- Not nullable

Usage Notes

Use of DATE is deprecated. Use the ANSI SQL-99-compliant CURRENT_DATE function instead.

Caution: The value of DATE as stored internally allows for dates from 0001-01-01 to 9999-12-31.

To avoid problems concerning years from 2000-01-01 on, use the ANSI format 'YYYY-MM-DD' when you set the system date.

You can also force the output format using a FORMAT clause. For more information on default data type output formats and the FORMAT output phrase, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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') ;

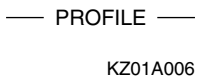
       Date
Mar 30, 1996
```

PROFILE

Purpose

Returns the current profile for the session or NULL if none.

Syntax



ANSI Compliance

PROFILE is a Teradata extension to the ANSI SQL-99 standard.

Properties

The value for PROFILE has the following characteristics:

- Data type: VARCHAR
- Character Data Type: Unicode
- Length: 30
- Nullable

Example

You can identify the current profile for the session with the following statement:

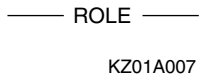
```
SELECT PROFILE ;
```

ROLE

Purpose

Returns the current role for the session or NULL if none.

Syntax



ANSI Compliance

ROLE is a Teradata extension to the ANSI SQL-99 standard.

Properties

The value for ROLE has the following characteristics:

- Data type: VARCHAR
- Character Data Type: Unicode
- Length: 30
- Nullable

Example

You can identify the current role for the session with the following statement:

```
SELECT ROLE ;
```

SESSION

Purpose

Returns the number of the session for the current user.

Syntax



FF07R003

ANSI Compliance

SESSION is a Teradata extension to the ANSI SQL-99 standard.

Properties

The value for SESSION has the following characteristics:

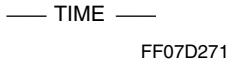
- Data type: INTEGER
- Length: 4
- Not nullable

TIME

Purpose

Provides the current time based on a 24-hour day.

Syntax



ANSI Compliance

TIME is a Teradata extension to the ANSI SQL-99 standard.

For the ANSI SQL-99-compliant syntax and behavior for the equivalent function, see "CURRENT_TIME" on page 12-5.

Properties

The value for TIME has the following characteristics:

- Data type: FLOAT
- Length: 8
- Format: HHMMSS.CC (hours, minutes, seconds, hundredths of a second)

Usage Note

Use of TIME is deprecated. You should use the ANSI SQL-99-compliant CURRENT_TIME function instead.

Example 1

The following example requests the current time:

```
SELECT TIME (FORMAT '99:99:99.99');
```

The hundredths of a second are not displayed by the default format, but they can be displayed by overriding the format.

Example 2

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:

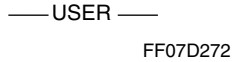
```
INSERT INTO HypoTable (ColumnA, ColumnB, InsertTime)
VALUES ('Abcde', 12345, TIME);
```

USER

Purpose

Provides the user name of the current user.

Syntax



ANSI Compliance

USER is ANSI SQL-99-compliant.

Properties

The value for USER has the following characteristics:

- Data type: VARCHAR
- Character Data Type: Unicode
- Length: 30
- Not nullable

Example 1

You can identify the current user with the following statement:

```
SELECT USER ;
```

The system responds with something like the following.

```
User
SQL01
```

Example 2

The following example selects the job title for the current user.

```
SELECT JobTitle FROM Employee WHERE Name = USER;
```

Chapter 13:

Data Type Conversions

This chapter describes the SQL CAST function, in addition to the rules for converting data from one type to another, both explicitly and implicitly.

Topics include:

- Date type conversion
- Implicit type conversion rules
- CAST function
- Teradata conversion syntax
- Data conversion in field mode
- Numeric conversion
- Character conversion
- BYTE conversion
- Teradata DATE conversion
- ANSI DateTime and Interval conversion
- Signed zone DECIMAL conversion

Data Type Conversions

Introduction

A data type conversion modifies the data type of an expression and can be either implicit or explicit.

Explicit conversions can be made using the CAST function or Teradata conversion syntax.

Forms of Data Type Conversions

The Teradata RDBMS supports the following forms of data conversion:

- Implicit
  For information, see "Implicit Type Conversions" on page 13-3.
- Explicit using the CAST function.
  For information, see "CAST in Explicit Data Type Conversions" on page 13-8.
- Explicit using Teradata conversion syntax.
  For information, see "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 13-11.

Implicit Type Conversions

Introduction

Teradata permits the assignment and comparison of some types without requiring the types to be explicitly converted.

ANSI Compliance

Implicit conversions are Teradata extensions to the ANSI standard.

Example

Assuming the following declarations, the following statement is not valid in ANSI SQL because T1.Yrs is a CHARACTER(2) value, which is added to a numeric value, and assigned to a numeric value.

- T2.Age is `SMALLINT`
- T1.Yrs is `CHARACTER(2)`

```
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.

Supported Data Types

Teradata performs implicit conversion on the following types:

FROM ...	TO ...
BYTE	BYTE
numeric	numeric
	DATE If the numeric value does not represent a valid date, an error is reported.
	character Cannot convert numeric to GRAPHIC characters.
DATE	numeric
	DATE
	character Cannot convert DATE to GRAPHIC characters.

FROM ...	TO ...
character	numeric Valid result only if the character string represents a numeric value.
	DATE 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.
	character If a character is not in the repertoire of the target character set, an error is reported.
	TIME/TIMESTAMP In field mode, the string must conform to the format of the target TIME or TIMESTAMP type. In record or indicator mode, the string must use the ANSI TIME or TIMESTAMP format.

Implicit Conversion Rules

SQL performs implicit type conversions on expressions before any operation is performed. The rules for implicit conversion are as follows:

Type of Conversion	Rules
Numeric to numeric	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 numeric expressions, see "Binary Arithmetic Result Data Types" on page 2-3.
Numeric to character	If a numeric argument in an SQL string function is implicitly converted to a 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

<table>
<tr><th>Type of Conversion</th><th>Rules</th></tr>
<tr><td>Numeric to character
(cont'd.)</td><td>For all other formats, the server character set is LATIN.
For information on data type formats, formatting characters, and the FORMAT phrase, see Chapter 8: "Output Format Phrases" in Teradata RDBMS SQL Reference, Volume 3.</td></tr>
<tr><td>Character to character</td><td>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.
<table><tr><th>THIS expression ...</th><th>IS converted to ...</th><th>AND the result is ...</th></tr><tr><td>'x'='x '</td><td>'x~~b~~'='x '</td><td>TRUE</td></tr><tr><td>'x'='xx'</td><td>'x~~b~~'='xx'</td><td>FALSE</td></tr></table>where ~~b~~ is a pad character.
For rules on the effect of server character sets on character conversion, see "Implicit Character-to-Character Translation" on page 13-32.</td></tr>
<tr><td>Character to numeric</td><td>If a 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 Chapter 8: "Output Format Phrases" in Teradata RDBMS SQL Reference, Volume 3.
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:
<table><tr><th>THIS expression ...</th><th>IS converted to ...</th><th>AND the result is ...</th></tr><tr><td>1.1*'$20.00'</td><td>1.10E 00*2.00E1</td><td>2.20E 01</td></tr><tr><td>'2'+'2'</td><td>2.00E 00+2.00E 00</td><td>4.00E 00</td></tr><tr><td>'A' + 2</td><td>----------</td><td>error</td></tr></table>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:
<pre>INSERT INTO Employee (EmpNo, Name, Salary)
VALUES (10022, 'Clements D', '$38,000.00');</pre></td></tr>
</table>

Type of Conversion	Rules

Character to numeric (*cont'd.*)	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`

Example 1: 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');
```

Example 2: 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 Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Example 3: Implicit Conversion of Character to DATE

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');
```

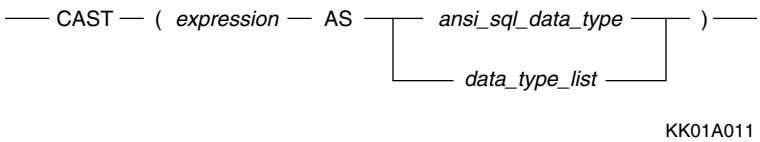
CAST in Explicit Data Type Conversions

Purpose

Converts an expression of a given data type to a different data type or the same data type with a different format.

Teradata supports two different syntaxes for CAST functionality, only one of which is ANSI SQL-99-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_type_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-99-compliant.

The form of CAST syntax that specifies *data_type_list* is a Teradata extension to the ANSI SQL-99 standard. Note than when *data_type_list* is an ANSI data type declaration, then this form of the syntax is also ANSI-compliant.

Usage Notes

The ANSI SQL-99-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 in that it allows a type declaration *or* data attributes *or* both. For more information on data types and attributes, see Chapter 1: "SQL Data Definition" in *Teradata RDBMS SQL Reference, Volume 3*.

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 8-36).
- BYTE to any data type other than BYTE.
- Any data type other than BYTE to BYTE.

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_type_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-99-compatible. When results for a field value that is a result of a numeric overflow of a numeric or decimal data type are returned in FIELD MODE, what is returned is asterisks ('***') rather than an error message.

RECORD and INDICATOR modes do not behave in this manner and return an error message.

Examples

The following examples illustrate how to perform data type conversions using CAST.

Example 1

Using ANSI CAST syntax:

```
WHERE INTEGER_FIELD = CAST(SUBSTRING(CHAR_FIELD FROM 3 FOR 3)
AS INTEGER);
```

Example 2

Using ANSI CAST syntax:

```
SELECT CAST(SUBSTRING(CHAR_FIELD FROM 1 FOR 2) AS
INTEGER),CAST(SUBSTRING (CHAR_FIELD FROM 3 FOR 3) AS INTEGER)
FROM T1;
```

Example 3

Using Teradata CAST syntax:

```
CREATE TABLE t2 (f1 TIME(0) FORMAT 'HHhMIm');

INSERT t2 (CAST('15h33m' AS TIME(0) FORMAT 'HHhMIm'));

SELECT f1 FROM t2;
```

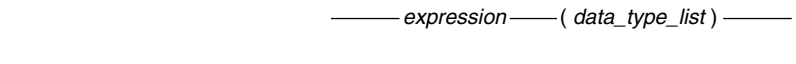
The result from the SELECT statement is:

```
    f1
------
15h33m
```

Teradata Conversion Syntax in Explicit Data Type Conversions

Introduction

Teradata conversion syntax is defined as follows:



HH01A019

where:

Syntax element ...	Specifies ...
expression	the data expression to be converted to the new definition defined by *data_type_list*
data_type_list	a data type declaration or data attributes or both. List elements must be separated by commas.

ANSI Compliance

This syntax is a Teradata extension to the ANSI SQL-99 standard.

Using CAST Instead of Teradata Conversion Syntax

Using Teradata conversion syntax is strongly discouraged. It is an extension to the ANSI SQL-99 standard and is retained only for backward compatibility with existing applications. Instead, use CAST to explicitly convert data types.

Usage Notes

When *data definition* includes a data type declaration, 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 the 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.

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 13-8.

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 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 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 maximum value of a SMALLINT is 32768:

```
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.

Numeric Conversion

Introduction

Numeric conversion is performed on numeric data, producing either a different numeric, character, date, or interval data type result.

Supported numeric conversions are as follows:

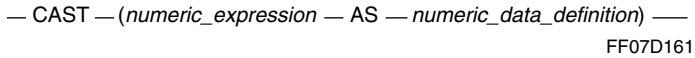
- "Numeric-to-Numeric Conversion" on page 13-15
- "Numeric-to-Character Conversion" on page 13-17
- "Numeric-to-DATE Conversion" on page 13-21
- "Exact Numeric-to-INTERVAL Conversion" on page 13-65

Numeric-to-Numeric Conversion

Purpose

Converts a numeric expression defined with one data type to a different numeric data type.

CAST Syntax



FF07D161

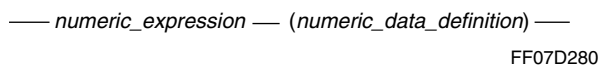
where:

Syntax element …	Specifies …
numeric_expression	an expression or existing field having a numeric data type.
numeric_data_definition	the new numeric data type to which *numeric_expression* is to be converted. A *numeric_data_definition* that specifies a FORMAT clause enables an alternative output format for *numeric_expression*. FORMAT is a non-ANSI Teradata extension.

ANSI Compliance

CAST is ANSI SQL-99-compliant.

Teradata Conversion Syntax



FF07D280

where:

Syntax element …	Specifies …
numeric_expression	an expression or existing field having a numeric data type.
numeric_data_definition	the new numeric data type to which *numeric_expression* is to be converted. A *numeric_data_definition* that specifies a FORMAT clause enables an alternative output format for *numeric_expression*.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL-99 standard.

Example 1

This example casts the numeric integer expression named IntegerField to DECIMAL(7,2).

```
CAST (IntegerField AS DECIMAL (7,2))
```

Conversion of numeric to 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 Chapter 3: “Decimal/Numeric Data Types" in *Teradata RDBMS SQL Reference, Volume 3*.

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).

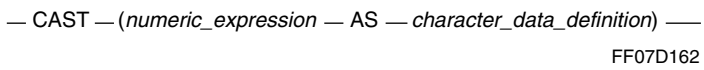
The use of FORMAT in CAST is a Teradata extension to the ANSI standard.

Numeric-to-Character Conversion

Purpose

Converts a numeric data type to a character data type.

CAST Syntax



FF07D162

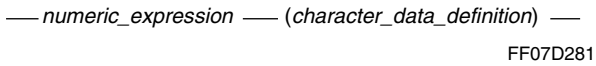
where:

Syntax element …	Specifies …
numeric_expression	the numeric data expression to be cast to a character type.
character_data_definition	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.

ANSI Compliance

CAST is ANSI SQL-99-compliant.

Teradata Conversion Syntax



FF07D281

where:

Syntax element …	Specifies …
numeric_expression	the numeric data expression to be cast to a character type.
character_data_definition	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.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL-99 standard.

How CAST Differs from Teradata Conversion Syntax

The process followed for the CAST function appears in the following table:

Stage	Process
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 following table describes the process:

Stage	Process
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 truncated.
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 13-19.

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 (b = 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))	'b$9,000.00'
20000.00	Salary (FORMAT'9(5)') (CHAR (5))	'20000'
9000.00	CAST (Salary AS CHAR(10))	'$9,000.00b'

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'.

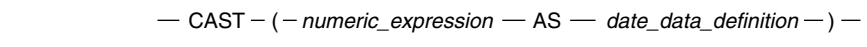
```
SELECT CAST (EmpNo AS CHAR(5)) FROM Employee;
```

Numeric-to-DATE Conversion

Purpose

Converts a numeric expression to a DATE data type.

CAST Syntax



1101A077

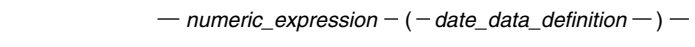
where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.
date_data_definition	the new DATE data type to which *numeric_expression* is to be converted. A *date_data_definition* that specifies a FORMAT clause enables an alternative output format. FORMAT is a non-ANSI Teradata extension.

ANSI Compliance

CAST is ANSI SQL-99-compliant; however, converting a numeric type to a date type is a Teradata extension to the ANSI SQL-99 standard.

Teradata Conversion Syntax



1101A076

where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.
date_data_definition	the new DATE data type to which *numeric_expression* is to be converted. A *date_data_definition* that specifies a FORMAT clause enables an alternative output format.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL-99 standard.

Translation of Numbers to Dates

Although not recommended, you can convert numbers to dates.

Teradata RDBMS stores each DATE value as a four-byte integer 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.

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.

Example 1

This example casts the numeric integer expression to a date format.

```
SELECT CAST (921201 AS DATE);
```

The result looks like this when the DateForm mode of the session is set to ANSIDate:

```
    921201
----------
1992-12-01
```

For More Information

For more information and more examples on conversions that involve date types, see “Teradata DATE Conversion” on page 13-36 and “ANSI DateTime and Interval Conversion” on page 13-39.

Character Conversion

Introduction

Character expression conversion is performed on character type data producing either a numeric type data result or a character type data result.

Supported character conversions are as follows:

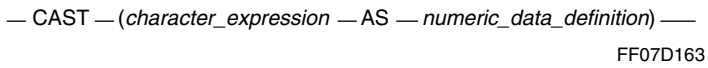
- "Character-to-Numeric Conversion" on page 13-24
- "Character-to-Character Conversion" on page 13-28

Character-to-Numeric Conversion

Purpose

Converts a character data string to a numeric value.

CAST Syntax



FF07D163

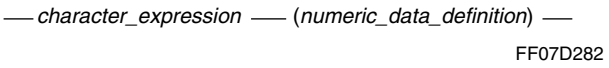
where:

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.

ANSI Compliance

CAST is ANSI SQL-99-compliant.

Teradata Conversion Syntax



FF07D282

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.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL-99 standard.

Usage Notes

Before any 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.

Stage	Process
1	Leading pad characters are ignored. Trailing pad characters are ignored, except for signed zoned decimal input. Note that embedded spaces are allowed as separators and as part of a 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.
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 is not treated as an error, but asterisks are displayed in lieu of an non-representable number.

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	Result
'$20,000.00'	DECIMAL(10,2)	20000.00
'$$$.50'	DECIMAL(10,2)	error
'$.50'	DECIMAL(8,2)	.50
'.345'	DECIMAL(8,3)	.345
'-1.234E-02'	FLOAT	-.01234
'-1E.-2'	FLOAT	error
'- 55'	INTEGER	-55
'E67'	FLOAT	error
'9876'	DECIMAL(4,2) FORMAT '99V99'	98.76
'-123'	INTEGER	-123
'9876'	DECIMAL(4,2) FORMAT '9(2)V9(2)'	98.76
'1-2-3'	INTEGER	123
'123-'	INTEGER	-123
'123- '	INTEGER	-123
'-1.234E 02'	FLOAT	-123.4

Character String	Converted To	Result
'111,222,333'	INTEGER FORMAT 'G9(I)'	111222333
'2.49 US Dollars'	DECIMAL(10,2) FORMAT 'GZ(I)D9(F)BN'	2.49
'25000 USD'	INTEGER FORMAT '9(I)C'	25000

A conversion that does not specify a FORMAT phrase uses the corresponding data type default format as defined in the SDF.

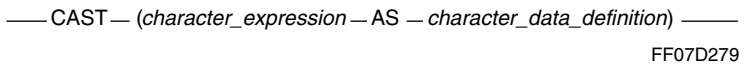
For more information on default data type formats, the SDF file, and the meaning of formatting characters in a FORMAT phrase, see Chapter 8: “Output Format Phrases” in *Teradata RDBMS SQL Reference, Volume 3*.

Character-to-Character Conversion

Purpose

Shortens or expands output character strings.

CAST Syntax



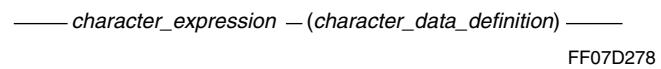
where:

Syntax element …	Specifies …
character_expression	a character expression to be cast to a different character data definition. FORMAT is a non-ANSI Teradata extension.
character_data_definition	the new character data format to which *character_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL-99-compliant.

Teradata Conversion Syntax



where:

Syntax element …	Specifies …
character_expression	a character expression to be cast to a different character data definition.
character_data_definition	the new character data format to which *character_expression* is to be converted.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL-99 standard.

CAST Syntax Usage Notes

The server character set of *character_expression* must have the same server character set as *character_data_definition*.

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 *character_data_definition*.

This is not the recommended way to perform this translation. Instead, use the TRANSLATE function. For information, see "TRANSLATE" on page 8-36.

General Usage Notes

If the source string (either CHAR or VARCHAR) is longer than the target data type (CHAR or VARCHAR), 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.

IF the source string data type is . . .	AND it is . . .	AND the target data type is . . .	THEN . . .
CHAR	longer than the target	VARCHAR	any trailing pad characters are trimmed.
CHAR or VARCHAR	shorter than the target	CHAR	any trailing pad characters are appended to the target.
CHAR	all pad characters	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
'HELLO '	7	VARCHAR(3)	'HEL', if session is in Teradata mode	3
			Error, if session is in ANSI mode	

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 field of the above types. 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 type of the field.

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 Chapter 3: ”SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6*.
- 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 Chapter 3: ”SQL Data Manipulation Language Statement Syntax” in *Teradata RDBMS SQL Reference, Volume 6*.
- You cannot assign or export GRAPHIC data from a single byte character set like ASCII or EBCDIC.

Implicit Character-to-Character Translation

Introduction

Implicit string translation occurs when two character strings are incompatible within a given operation. For example, 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.
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>)
```

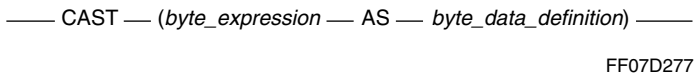
The result is Latin.

BYTE Conversion

Purpose

Converts a byte expression to a different byte data definition.

CAST Syntax



FF07D277

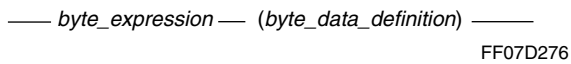
where:

Syntax element . . .	Specifies . . .
byte_expression	an expression in BYTE format to be cast to a different BYTE data definition.
byte_data_definition	the new BYTE data format to which *byte_expression* is to be converted.

ANSI Compliance

CAST is ANSI SQL-99-compliant.

Teradata Conversion Syntax



FF07D276

where:

Syntax element . . .	Specifies . . .
byte_expression	an expression in BYTE format to be cast to a different BYTE data definition.
byte_data_definition	the new BYTE data format to which *byte_expression* is to be converted.

ANSI Compliance

Teradata conversion syntax is a Teradata extension to the ANSI SQL-99 standard.

Conversions Where Source and Target Types Differ in Length

If the length specified by *byte_data_definition* is less than the length of *byte_expression*, bytes beyond the specified length are truncated. No error is reported.

If *byte_data_definition* is fixed-length and the length is greater than that of *byte_expression*, bytes of value binary zero are appended as required.

BYTE Conversion Supports BYTE Types Only

For byte data, the Teradata RDBMS allows conversions only to an identical type (byte to byte). Byte data cannot be converted to any data type other than byte, including numeric, graphic, or character.

Teradata DATE Conversion

Introduction

This topic describes the conversion of character and numeric data to and from non-ANSI DATE formats. Non-ANSI DATE formats are defined with a DateForm of IntegerDate or a FORMAT phrase.

For information on how to convert ANSI DateTime data, see "ANSI DateTime and Interval Conversion" on page 13-39 and "DATE-to-Character Conversion" on page 13-41.

For more information on implicit Teradata DATE conversion rules, see "Implicit Type Conversions" on page 13-3.

ANSI Compliance

Teradata DATE conversions are Teradata extensions to the ANSI SQL-99 standard.

Character to Teradata DATE Conversion

To convert a character string to a non-ANSI DATE value, you can use a FORMAT phrase. The character string must represent a date in the format of the target DATE type.

For example, conversion works if a character string represents a date in the form YY/MM/DD or YYYY/MM/DD and the target DATE column format is YY/MM/DD. Conversion fails, however, if the column format is M4BDD,BY4.

You can also use a FORMAT phrase to convert a DATE value to a string. The date is returned as specified in the FORMAT phrase.

You cannot convert DATE to a character data type if the server character set is GRAPHIC.

For details on using the FORMAT phrase with DATE types, see Chapter 8: "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Teradata DATE to Numeric Conversion

DATE data may be converted to SMALLINT, BYTEINT, INTEGER, DECIMAL(n,m), or FLOAT.

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.

A FORMAT phrase in DATE to numeric conversion may only contain the 9 formatting character. For example:

```
SELECT CAST (DATE '1985-12-31' AS INTEGER FORMAT '9999999');
```

Numeric to Teradata DATE Conversion

Although not recommended, you can convert numbers to dates. Numbers are converted to dates using the formula described in the previous discussion on Teradata DATE to numeric conversion.

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 raw date value . . .	Translates to this 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.

Example 1

To list employees who were born between January 30, 1938, and March 30, 1943, you could specify the date information as follows, where dob is a DATE column and the format is M3BDDBY4:

```
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
```

Example 2

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
```

Example 3

The following example treats DATE as a numeric format. This example is included only for the sake of completeness: you should *not* define date data as a numeric type. Instead, use the ANSI Date 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
```

ANSI DateTime and Interval Conversion

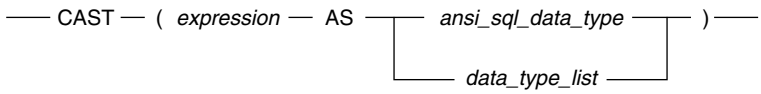
Introduction

ANSI DateTime values are exported directly to a client system as character fields with fixed format (CharFix). Client applications may manipulate this data or display it in other formats.

You cannot convert DateTime data to a character data type of GRAPHIC.

The following pages discuss DateTime and Interval conversions to (or from) other data types using CAST syntax and Teradata conversion syntax.

CAST Syntax



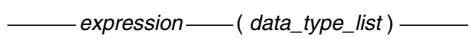
KK01A011

ANSI Compliance

CAST is ANSI SQL-99-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



HH01A019

ANSI Compliance

This is a Teradata extension to the ANSI SQL-99 standard.

DATE to TIMESTAMP Conversion

Usage Notes

The result of the conversion is taken directly from the YEAR, MONTH, and DAY fields of the source DATE record.

HOUR, MINUTE, and SECONDS are set to zeros.

If the target type is specified WITH TIME ZONE, then the Time Zone displacement of the current session is used.

Example

Assuming that the current date is July 10, 1999 and the local time zone is -8, the following query returns ‘1999-10-07 00:00:00.000-08:00’.

```
SELECT CAST(CURRENT_DATE AS TIMESTAMP(3) WITH TIME ZONE);
```

DATE-to-Character Conversion

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

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

Field 1 in the table INTDAT is a DATE value with the explicit format 'YYYY,MMM,DD'. Assume that you want to convert this to a value of CHAR(12), and an explicit output format of 'MMMBDD,BYYYY'.

```
SELECT F1 FROM INTDAT ;
```

The result (without a type change) is the following report:

```
F1
----------
1900,Dec,31
```

Now 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
```

TIME-to-TIME Conversion

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);
```

TIME to TIMESTAMP Conversion

Usage Notes

On conversion, YEAR, MONTH, and DAY are set to the values for CURRENT_DATE. HOUR, MINUTE, and SECOND are taken directly from the YEAR, MONTH, and DAY fields of the source TIME record.

If the target value is defined as WITH TIME ZONE, then its TIME and ZONE fields are set from the explicit or implicit values taken from the source value.

Example

Assuming that 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));
```

TIME-to-Character Conversion

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

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.

TIMESTAMP-to-TIMESTAMP Conversion

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 13-48.

TIMESTAMP-to-DATE Conversion

Usage Notes

The result is the YEAR, MONTH, and DAY values from the TIMESTAMP value of *expression* after any necessary adjustments for Time Zone displacement have been made.

This means that fields are not just copied. If the Time Zone adjustment changes a value in the range of DATE, then that changed value is used.

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.

TIMESTAMP-to-TIME Conversion

Usage Notes

The result of the conversion is taken directly from the HOUR, MINUTE, and SECOND fields of the source TIMESTAMP record.

If the target type is specified WITH TIME ZONE, then either the explicit or the implicit Time Zone displacement of the value is included.

Otherwise, the assigned value is adjusted to the current Time Zone displacement of the session.

You cannot convert a character data type of GRAPHIC to TIMESTAMP.

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);
```

TIMESTAMP-to-Character Conversion

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.
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.

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.

INTERVAL-to-INTERVAL Conversion

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.

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);
```

INTERVAL-to-Exact Numeric Conversion

When *expression* is an INTERVAL with only one field, then you can use CAST syntax or Teradata explicit conversion syntax to convert the INTERVAL data to an exact numeric data type.

This numeric value is the value of the single numeric field in the INTERVAL record.

INTERVAL-to-Character Conversion

You can use CAST syntax or Teradata explicit conversion syntax to convert an INTERVAL type to its canonical character string representation.

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.

Character-to-DATE Conversion

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.

If the string can be converted to a valid DATE, then it is. Otherwise, an error is returned.

IF the current session is in this mode when the conversion is made . . .	THEN the default output format of the target DATE data type is . . .
INTEGERDATE	'YY/MM/DD', or the current setting of the default date format in the specification for data formatting (SDF) file
ANSIDATE	'YYYY-MM-DD'

Forcing a FORMAT on CAST for Converting Character to DATE

You can use a FORMAT phrase to specify an explicit format for the DATE target data type. A conversion that does not specify a FORMAT phrase uses the default output format for the DATE data type.

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 output formats and the FORMAT phrase, see "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Converting CHARACTER to DATE in an INSERT Statement

To convert character data to date values that can be inserted into a DATE table column, use CAST with a FORMAT phrase that matches the format of the DATE column.

For example, table DATCHAR12 contains a field defined as F1 CHAR(12) that is loaded with date values expressed as CHAR(12) in the form 'MMMBDD,BYYYY'.

Table INTDATE contains a field defined as F1 DATE, and was created by a session in INTEGERDATE mode.

You perform a SHOW TABLE for INTDATE and find that the F1 DATE field is described as F1 DATE FORMAT 'YY/MM/DD'.

Use the following INSERT statement to insert the date values stored in DATCHAR12 as CHAR(12) values into INTDATE as DATE values.

```
INSERT INTO INTDATE
SELECT CAST(F1 AS DATE FORMAT 'MMMBDD,BYYYY')
FROM DATCHAR12;
```

Note that values in DATCHAR12 must be valid dates. For any instance of F1 in DATCHAR12 that contains data that does not represent a valid date, an error is returned to the application.

Character-to-TIME Conversion

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.

You cannot convert a character data type of 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 output format for the TIME data type.

IF the character string is converted to ...	THEN the default output 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 output formats and the FORMAT phrase, see "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

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, 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 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 regardless of whether the target data type specifies a 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)');
  ```

FORMAT Phrase Restrictions

In character-to-TIME conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- Z
- T

Example 1

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

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

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

This query returns the value '15h33m'.

```
SELECT CAST('15h33m'
AS TIME(0) FORMAT 'HHhMIm');
```

Character-to-TIMESTAMP Conversion

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.

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 output format for the TIMESTAMP data type.

IF the character string is converted to …	THEN the default output 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 output formats and the FORMAT phrase, see "Output Format Phrases" in *Teradata RDBMS SQL Reference, Volume 3*.

Example

The following query returns '1997-12-31 23:59:59.999999-08:00'.

```
SELECT CAST('1997-12-31 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 ( '1990-09-19 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 ( '1999-01-19 +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('12-02-99 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('12-02-99 12h:15.12s:30m'
  AS TIMESTAMP FORMAT 'DD-MM-YYBHHh:SSDS(6)s:MIm');
  ```

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

Character-to-INTERVAL Conversion

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.

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);
```

Exact Numeric-to-INTERVAL Conversion

Usage Notes

Numeric data is converted to an INTERVAL value with a single DateTime field.

If the numeric value is in the defined value range for the INTERVAL, then it is used as the single field of the INTERVAL; otherwise, an error is returned.

Example

The following query returns ' -5' (with three leading pad characters).

```
SELECT CAST(-5 AS INTERVAL YEAR(4));
```

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 the Teradata RDBMS.

The rightmost character of the input data string is assumed to contain the zone (overpunch) bit.

The characters representing the various zone-numeric combinations are listed in the following table.

Last Character (Input String)	Numeric Conversion	Last Character (Input String)	Numeric Conversion
{	n … 0	}	-n … 0
A	n … 1	J	-n … 1
B	n … 2	K	-n … 2
C	n … 3	L	-n … 3
D	n … 4	M	-n … 4
E	n … 5	N	-n … 5
F	n … 6	O	-n … 6
G	n … 7	P	-n … 7
H	n … 8	Q	-n … 8
I	n … 9	R	-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.

If field values are normally retrieved for manipulation as numeric data, the sign FORMAT phrase should not be used at column creation time.

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 an 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’.

Appendix A:

Notation Conventions

This appendix describes the notation conventions used in this book.

Introduction

This book uses three conventions to describe the SQL syntax and code:

Convention	Description
Syntax diagrams	Describes SQL syntax form, including options. For details, see "Syntax Diagram Conventions" on page A-2.
Square braces in the text	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.
Japanese character code shorthand notation	Represent unprintable Japanese characters. For details, see "Character Shorthand Notation Used In This Book" on page A-6.

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 A-8.

Syntax Diagram Conventions

Notation Conventions

The following table defines the notation used in this section:

<table>
<tr><th>Item</th><th>Definition / Comments</th></tr>
<tr><td>Letter</td><td>An uppercase or lowercase alphabetic character ranging from A through Z.</td></tr>
<tr><td>Number</td><td>A digit ranging from 0 through 9.

Do not use commas when entering a number with more than three digits.</td></tr>
<tr><td>Word</td><td>Variables and reserved words.

<table>
<tr><th>IF a word is shown in . . .</th><th>THEN it represents . . .</th></tr>
<tr><td>UPPERCASE LETTERS</td><td>a keyword.

Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase.

If a keyword is shown in uppercase, you may enter it in uppercase or mixed case.</td></tr>
<tr><td>lowercase letters</td><td>a keyword that you must enter in lowercase, such as a UNIX command.</td></tr>
<tr><td>lowercase italic letters</td><td>a variable such as a column or table name.

You must substitute a proper value.</td></tr>
<tr><td>lowercase bold letters</td><td>a variable that is defined immediately following the diagram that contains it.</td></tr>
<tr><td>UNDERLINED LETTERS</td><td>the default value.

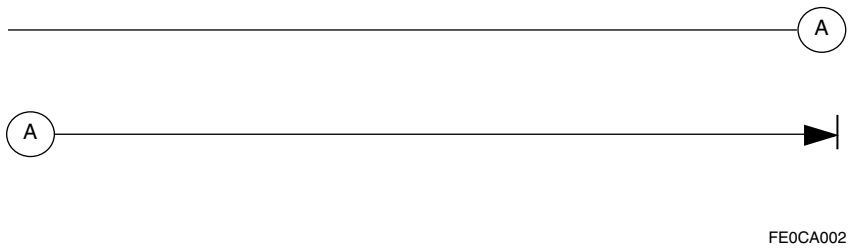
This applies both to uppercase and to lowercase words.</td></tr>
</table>
</td></tr>
<tr><td>Spaces</td><td>Use one space between items, such as keywords or variables.</td></tr>
<tr><td>Punctuation</td><td>Enter all punctuation exactly as it appears in the diagram.</td></tr>
</table>

Paths

The main path along the syntax diagram begins at the left and proceeds from 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.

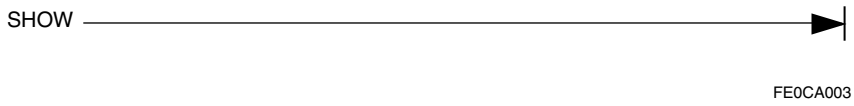
Paths that are too long for one line use continuation links. Continuation links are small circles with letters indicating the beginning and end of a link:



When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue.

Required Items

Required items appear on the main path:



If you can choose from more than one item, the choices appear vertically, in a stack. The first item appears on the main path:



Optional Items

Optional items appear below the main path:



If choosing one of the items is optional, all the choices appear below the main path:



You can choose one of the options, or you can disregard all of the options.

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.

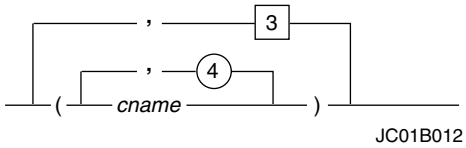


In the above syntax, the following formats are valid:

- 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.



The following rules apply to loops:

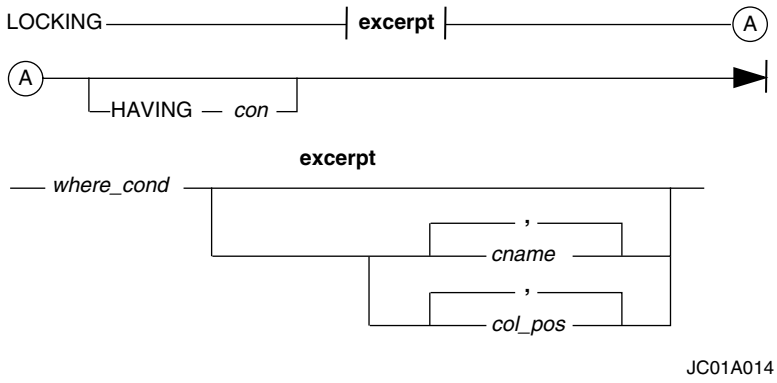
IF . . .	**THEN** . . .
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may enter *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must enter at least 3 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 either side of the break. A name for the excerpted piece appears between the break marks in boldface type.

The named phrase appears immediately after the complete diagram, as illustrated by the following example.



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

For more information on these encodings, see Appendix B: “Japanese Character Sets," in *Teradata RDBMS International Character Set Support*.

Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a..z A..Z 0..9	Any	Any single byte Latin letter or digit.
a..z A..Z 0..9	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.

Symbol	Encoding	Meaning
T	Any	Any multibyte character. Its encoding depends on the current character set. For KanjiEUC, code set 3 characters sometimes appear preceded by "ss_3".
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 either 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).

Example1

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.

Example2

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	SPACE	0x20
KANJI1	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
∀	For all
∃	There exists

Appendix B:

SQL Functions, Operators, Expressions, and Predicates

This appendix provides a brief description of SQL functions, operators, expressions, and predicates.

SQL Functions

Definition

SQL functions are subroutines that 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 an input argument, 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 three types of SQL functions:

- Column
- Scalar
- Table

The following table defines these three types.

Function Type	Definition
Column	The argument is a set of like values. The result is a single value or null. Can be used in any SQL statement where an expression can be used.
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.
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.

SQL Operators

Definition

SQL operators are symbols that perform arithmetic or logical operations on their arguments.

Examples of Simple SQL Operators

- =
- >
- >=

Examples of SQL Set Operators

- INTERSECT
- UNION

Examples of SQL Comparison Operators

- BETWEEN
- OVERLAPS

SQL Expressions

Definition

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
- Combination of column names and constants connected by operators.

Examples of Expressions

The following are examples of expressions:

```
'Test Tech'
Employee.Name
Salary * 12 + 100
Salary / 52 > 500
Salary > 50000 OR Age > 60
```

SQL Predicates

Definition

SQL predicates specify a condition of a row or group that has one of three possible states:

- TRUE
- FALSE
- NULL (or unknown)

Examples of Simple Logical Predicates

- NOT
- AND
- OR

Examples of Quantifier Predicates

- EXISTS
- ANY
- ALL

Index

Symbols

|| , concatenation operator 8–6

A

ABS function 2–10
ACCOUNT function 12–3
Account string, get account string 12–3
ACOS inverse trigonometric function 2–46
ACOSH hyperbolic function 2–58
ADD_MONTHS function 5–30
Addition operator 2–2
Aggregate functions
  AVG 6–5
  CORR 6–7
  COUNT 6–10
  COVAR_POP 6–15
  COVAR_SAMP 6–17
  date and 6–2
  DateTime types and 5–25
  DISTINCT option and 6–4
  HAVING clause and 6–4
  interval types and 5–25
  KURTOSIS 6–20
  MAX 6–22
  MIN 6–24
  nesting 6–2
  nulls and 6–3
  REGR_AVGX 6–26
  REGR_AVGY 6–29
  REGR_COUNT 6–32
  REGR_INTERCEPT 6–34
  REGR_R2 6–37
  REGR_SLOPE 6–40
  REGR_SXX 6–43
  REGR_SXY 6–46
  REGR_SYY 6–49
  select list containing 6–2
  SKEW 6–52
  STDDEV_POP 6–54
  STDDEV_SAMP 6–57
  SUM 6–60
  VAR_POP 6–62
  VAR_SAMP 6–64
  when expression evaluates to zero 6–3
  WHERE clause and 6–4
ALL predicate 9–11
AMP
  identify maximum number in configuration 11–3
  identify with HASHAMP 11–3
AND logical operator 9–27
  truth table 9–29
ANY predicate 9–11
Arithmetic functions
  ABS function 2–10
  EXP function 2–17
  LN function 2–19
  LOG function 2–20
  RANDOM function 2–23
  SQRT function 2–33
Arithmetic operators
  - 2–2
  * 2–2
  ** 2–2
  + 2–2
  / 2–2
  addition operator 2–2
  division operator 2–2
  exponentiate 2–2
  MOD operator 2–2
  multiplication 2–2
  subtraction operator 2–2
  unary minus operator 2–2
  unary plus operator 2–2
ASIN inverse trigonometric function 2–48
ASINH hyperbolic function 2–60
ATAN inverse trigonometric function 2–50
ATAN2 inverse trigonometric function 2–52
ATANH hyperbolic function 2–62
Attribute functions 10–2
  BYTES 10–3
  CHARACTER_LENGTH 10–5
  CHARACTERS 10–8
  FORMAT 10–9
  MCHARACTERS 10–2, 10–5
  OCTET_LENGTH 10–10
  TYPE 10–12
AVERAGE aggregate function. See AVG aggregate function.
AVG aggregate function, described 6–5
AVG function
  DateTime types and 5–25
  Interval types and 5–25
AVG window function, described 7–25

B

BETWEEN predicate 9–10
Blank, as used in strings 9–35
Built-in functions 12–2
  ACCOUNT 12–3
  CURRENT_DATE 12–4
  CURRENT_TIME 12–5
  CURRENT_TIMESTAMP 12–7
  DATABASE function 12–9
  DATE function 12–10
  PROFILE 12–12
  ROLE 12–13
  SESSION 12–14
  TIME 12–15
  USER 12–16
Byte conversion 13–34
Byte functions, TRIM 8–47
BYTES function 10–3

C

CALENDAR system view
  cumulative sum, and calculating 7–38
  moving difference, and calculating 7–49
Case blind comparisons 9–4
CASE expression and nulls 1–25
CASE operation
  COALESCE expression 1–25
  data type of, rules governing 1–14
  defined 1–2
  NULLIF expression 1–27
  searched 1–7
  valued 1–3
Case sensitivity in comparisons 3–13
CASE_N function 2–12
CAST expression 13–8, 13–11
  ANSI DateTime conversion 13–39, 13–54
CAST, in data type conversions 13–8
CHAR function. See CHARACTERS function.
CHAR2HEXINT function 8–12
Character
  conversion to formatted DATE conversion 13–56
  translation 13–32
Character conversion 13–9, 13–23
  character-to-character 13–28
  character-to-numeric 13–24
  string functions and 8–5
Character data type
  assignability rules for 13–31
Character string functions. See String functions
Character to character conversion 13–28
CHARACTER_LENGTH function 10–5
CHARACTERS function 10–8
  ANSI equivalent 10–5
Character-to-DATE conversion 13–56
Character-to-INTERVAL conversion 13–64
Character-to-numeric conversion 13–24
Character-to-TIME conversion 13–58
Character-to-TIME conversion, implicit 13–4
Character-to-TIME WITH TIME ZONE conversion 13–58
Character-to-TIMESTAMP conversion 13–61
Character-to-TIMESTAMP conversion, implicit 13–4
CHARS function. See CHARACTERS function.
COALESCE expression 1–25
Comparison evaluations by data type 3–6
Comparison operators
  = 3–3
  > 3–3
  >= 3–3
  GE 3–3
  general rules 3–5
  GT 3–3
  Japanese character sets 3–11
  LE 3–3
  LT 3–3
  NE 3–3
  results 3–5
Comparison rules
  arithmetic and string 3–7
Concatenation operator 8–6
Conditional expressions 9–27
Conversion
  ANSI DateTime 13–39
  byte 13–34
  character 13–23
  character to character 13–28
  character to DATE 13–56
  character to formatted date 13–56
  character to INTERVAL 13–64
  character to numeric 13–24
  Character to TIME 13–58
  Character to TIME WITH TIME ZONE 13–58
  Character to TIME, implicit 13–4
  character to TIMESTAMP 13–61
  character to TIMESTAMP, implicit 13–4
  data type 13–2
  DATE to character 13–41
  DATE to TIMESTAMP 13–40
  explicit, using CAST 13–8
  explicit, using Teradata syntax 13–11

field mode 13–13
implicit 13–3
Interval 13–39
Interval to character 13–55
INTERVAL to INTERVAL 13–52
interval to numeric 13–54
numeric 13–14, 13–15
numeric to character 13–17
numeric to INTERVAL 13–65
signed zone decimal 13–66
table showing results of 13–3
Teradata DATE 13–36
TIME to character 13–45
TIME to TIME 13–43
TIME to TIMESTAMP 13–44
TIMESTAMP to character 13–50
TIMESTAMP to DATE 13–48
TIMESTAMP to TIMESTAMP 13–47
Timestamp to timestamp 13–49
using CAST to convert 13–8
using Teradata syntax to convert 13–11
CORR aggregate function, described 6–7
COS trigonometric function 2–40
COSH hyperbolic function 2–55
COUNT aggregate function, described 6–10
COUNT function
DateTime types and 5–25
Interval types and 5–25
COUNT window function, described 7–31
COVAR_POP aggregate function, described 6–15
COVAR_SAMP aggregate function, described 6–17
CSUM function, described 7–37
Cumulative sum
CALENDAR view, computing with 7–38
computing 7–37, 7–74
CURRENT_DATE function 12–4
CURRENT_TIME function 12–5
CURRENT_TIMESTAMP function 12–7

D

Data conversion rules, explicit 13–11
Data conversion rules, implicit 13–3
Data definition
byte conversion 13–34
CAST in data type conversions 13–8
character conversion 13–23
character to character 13–28
character-to-numeric 13–24
character-to-DATE conversion 13–56
character-to-formatted DATE conversion 13–56
character-to-INTERVAL conversion 13–64
character-to-TIME conversion 13–58
character-to-TIMESTAMP conversion 13–61
conversion rules
implicit rules 13–3
data type conversions 13–2
DATE conversion 13–39
DATE conversion (Teradata) 13–36
DATE-to-character conversion 13–41
DATE-to-TIMESTAMP conversion 13–40
Exact numeric-to-INTERVAL conversion 13–65
Interval-to-character conversion 13–55
interval-to-exact numeric conversion 13–54
INTERVAL-to-INTERVAL conversion 13–52
numeric conversion 13–14
numeric-to-character 13–17
numeric-to-numeric 13–15
signed zone decimal conversion 13–66
TIMESTAMP-to-character conversion 13–50
TIMESTAMP-to-DATE conversion 13–48
TIMESTAMP-to-TIMESTAMP conversion 13–47, 13–49
TIME-to-character conversion 13–45
TIME-to-TIME conversion 13–43
TIME-to-TIMESTAMP conversion 13–44
type conversion rules
explicit 13–11
Data type, conversions 13–2
Database, get default database 12–9
Date
get current date (Teradata) 12–10
get system date 12–4
DATE conversion (ANSI) 13–39
DATE conversion (Teradata) 13–36
Date expressions, Teradata 5–27
DATE values, scalar operations on 5–29
Date, aggregate operations and 6–2
DATE, as logical predicate 9–6
DATE, conversion to character 13–41
DateTime conversion 13–39
DateTime expressions 5–7
rules for, ANSI 5–9
DateTime functions, and scalar operations 5–26
DateTime scalar operations
arithmetic 5–23
restrictions on 5–6
DateTime types
aggregate functions and 5–25
assignment rules 5–3, 5–4
DATE-to-TIMESTAMP conversion 13–40
DECIMAL/NUMERIC types
arithmetic expression and rounding 2–6

defined B-5
DISTINCT, SELECT option 6–4
Division operator 2–2

E

ESCAPE, with LIKE predicate 9–34, 9–40
Exact numeric-to-INTERVAL conversion 13–65
EXCEPT operator 4–25
EXISTS predicate 9–14
EXP function 2–17
Exponentiation operator 2–2
Expressions, defined B-4
EXTRACT function 5–36

F

Fallback AMP
  identify maximum number with HASHBAKAMP 11–5
  identify with HASHBAKAMP 11–5
FALSE 9–28
Field mode
  data type conversions and 13–13
FORMAT phrase 10–9
Functions
  defined B-2
  types of B-2

G

Group count, example 7–35
Group sum, computing 7–74

H

Hash index
  ordered analytical functions and 7–3
HASHAMP function 11–3
HASHBAKAMP function 11–5
HASHBUCKET function 11–7
Hash-related functions 11–2
  HASHAMP 11–3
  HASHBAKAMP 11–5
  HASHBUCKET 11–7
  HASHROW 11–9
HASHROW function 11–9
Hyperbolic functions 2–54
  ACOSH 2–58
  ASINH 2–60
  ATANH 2–62
  COSH 2–55
  SINH 2–56
  TANH 2–57

I

Implicit type conversion 13–3
IN predicate 9–20
INDEX function 8–15
  ANSI equivalent 8–3
INTERSECT operator 4–22
Interval expressions 5–12
  rules for, ANSI 5–20
Interval scalar operations
  arithmetic 5–23
  restrictions on 5–6
Interval types
  aggregate functions and 5–25
  assignment rules 5–3, 5–4
Interval-to-character conversion 13–55
Interval-to-exact numeric conversion 13–54
INTERVAL-to-INTERVAL conversion 13–52
Inverse trigonometric functions 2–39
  ACOS 2–46
  ASIN 2–48
  ATAN 2–50
  ATAN2 2–52
IS NOT NULL predicate 9–25
IS NULL predicate 9–25

J

Japanese character code notation, how to read A–6
Join index
  ordered analytical functions and 7–3

K

KURTOSIS aggregate function, described 6–20

L

Least squares, computing 7–56
LIKE predicate 9–32
Linear regression, computing 7–56
LN function 2–19
LOG function 2–20
Logical expressions
  BETWEEN predicate 9–10
  FALSE result 9–28
  NOT BETWEEN predicate 9–10
  TRUE result 9–28
  UNKNOWN result 9–28
Logical operators
  AND 9–27
  defined 9–27
  NOT 9–27
  OR 9–27
  search conditions and 9–27
Logical predicate
  conditional expression as 9–2
  DATE as 9–6
  defined 9–2
  order of evaluation 9–28
  primitives, tabular summary of 9–3
  SQL use of 9–2
LOWER function 8–21

M

MAVG function, described 7–40
MAX aggregate function, described 6–22
MAX function
  DateTime types and 5–25
  Interval types and 5–25
MAX window function, described 7–44
MAXIMUM aggregate function. See MAX aggregate function.
MCHARACTERS function 10–2, 10–5
  ANSI equivalent 10–2
MDIFF function, described 7–48
MIN aggregate function, described 6–24
MIN function
  DateTime types and 5–25
  Interval types and 5–25
MIN window function, described 7–52
MINDEX function 8–3, 8–23
  ANSI equivalent 8–3
MINIMUM aggregate function. See MIN aggregate function.
MINUS operator 4–25
MLINREG function, described 7–56
MOD operator 2–2
Moving average, computing 7–40
Moving difference
  CALENDAR view, computing with 7–49
  computing 7–48
Moving sum, computing 7–59, 7–74
MSUBSTR function 8–3, 8–33
  ANSI equivalent 8–3
MSUM function, described 7–59
Multiplication operator 2–2

N

Name, get user name 12–16
NOT BETWEEN predicate 9–10
NOT EXISTS predicate 9–14
NOT IN predicate 9–20
  NOT EXISTS predicate and 9–16
NOT logical operator 9–27
NULLIF expression 1–27
NULLIFZERO function 2–21
Nulls
  CASE expression 1–25
Nulls, aggregate operations and 6–3
Nulls, searching for/excluding 9–25
Numeric conversion 13–14
  numeric-to-character 13–17
  numeric-to-date 13–22, 13–37
  numeric-to-numeric 13–15
Numeric-to-character conversion 13–17
Numeric-to-date conversion 13–22, 13–37
Numeric-to-numeric conversion 13–15

O

OCTET_LENGTH function 10–10
OLAP functions. See Ordered analytical functions.
Operator, defined B-3
OR logical operator 9–27
  truth table 9–29
Order of evaluation. See Logical predicate
Ordered analytical functions 7–1
  aggregates and 7–16
  AVG window function 7–25
  common characteristics of 7–13
  COUNT window function 7–31
  CSUM 7–37

derived tables and 7–16
description 7–2
examples 7–20
extending Teradata queries 7–3
GROUP BY clause 7–18
hash indexes and 7–3
HAVING clause 7–16
join indexes and 7–3
MAVG 7–40
MAX window function 7–44
MDIFF 7–48
MIN window function 7–52
MLINREG 7–56
MSUM 7–59
PERCENT_RANK window function 7–61
QUALIFY clause 7–16
QUANTILE 7–64
RANK 7–67
RANK window function 7–70
result order 7–14
ROW_NUMBER window function 7–72
SUM window function 7–74
syntax alternatives for 7–5
Teradata OLAP functions 7–5
Teradata Warehouse Miner and 7–3
views and 7–16
window 7–7
window functions 7–5
OVERLAPS predicate 9–43

P

Partitioned primary index. See PPI.
PERCENT_RANK window function, described 7–61
POSITION function 8–3, 8–23, 8–36
PPI
defined 2–14, 2–30
maximum partitions when using CASE_N 2–14
maximum partitions when using RANGE_N 2–30
PPI functions
CASE_N 2–12
RANGE_N 2–26
Precedence
arithmetic expressions 2–7
logical operators 9–28
operator 2–7
set operators 4–6
Predicates B-5
ALL 9–11
ANY 9–11
BETWEEN 9–10
EXISTS 9–14
IN 9–20
IS NOT NULL 9–25
IS NULL 9–25
LIKE 9–32
logical 9–2
NOT BETWEEN 9–10
NOT EXISTS 9–14
NOT IN 9–20
OVERLAPS 9–43
quantifier 9–11
SOME 9–11
PROFILE function 12–12
Profiles, getting the current profile 12–12

Q

Quantifier predicates
ALL 9–11
ANY 9–11
SOME 9–11
QUANTILE function, described 7–64

R

RANDOM function 2–23
RANDOM function, and valued CASE 1–4
RANGE_N function 2–26
RANK function, described 7–67
RANK window function, described 7–70
REGR_AVGX aggregate function, described 6–26
REGR_AVGY aggregate function, described 6–29
REGR_COUNT aggregate function, described 6–32
REGR_INTERCEPT aggregate function, described 6–34
REGR_R2 aggregate function, described 6–37
REGR_SLOPE aggregate function, described 6–40
REGR_SXX aggregate function, described 6–43
REGR_SXY aggregate function, described 6–46
REGR_SYY aggregate function, described 6–49
Remaining average, example 7–14
Remaining count, example 7–35
Remaining sum, example 7–81
ROLE function 12–13
Roles, getting the current role 12–13
Rounding
arithmetic operators and DECIMAL/NUMERIC data 2–6
ROW_NUMBER window function, described 7–72

Rowhash, identify with HASHROW function 11–9

S

Scalar, converting scalar value expressions 13–8
SDF
  and data type default formats 13–6
  Currency 13–26
  CurrencyName 13–26
  GroupingRule 13–26
  GroupSeparator 13–26
  RadixSeparator 13–26
  relationship to FORMAT phrase 13–26
Search conditions
  logical operators and 9–27
SESSION function 12–14
Session, get session number 12–14
Set operators
  ALL option 4–8
  derived tables and 4–12
  described 4–2
  EXCEPT operator 4–25
  INSERT...SELECT statements containing 4–15
  INTERSECT operator 4–22
  MINUS operator 4–25
  precedence 4–6
  rules for 4–5
  rules for connecting queries by 4–19
  set result, attributes of 4–9
  subqueries containing 4–13
  UNION operator 4–27
  view definitions containing 4–17
Signed zone decimal conversion 13–66
SIN trigonometric function 2–42
SINH hyperbolic function 2–56
SKEW aggregate function, described 6–52
SOME predicate 9–11
SOUNDEX function, described 8–26
Specification for data formatting, see SDF
SQL expressions
  aggregate functions
    AVG 6–5
    CORR 6–7
    COUNT 6–10
    COVAR_POP 6–15
    COVAR_SAMP 6–17
    DISTINCT option 6–4
    HAVING clause and 6–4
    KURTOSIS 6–20
    MAX 6–22
    MIN 6–24
    REGR_AVGX 6–26
    REGR_AVGY 6–29
    REGR_COUNT 6–32
    REGR_INTERCEPT 6–34
    REGR_R2 6–37
    REGR_SLOPE 6–40
    REGR_SXX 6–43
    REGR_SXY 6–46
    REGR_SYY 6–49
    SKEW 6–52
    STDDEV_POP 6–54
    STDDEV_SAMP 6–57
    SUM 6–60
    VAR_POP 6–62
    VAR_SAMP 6–64
    WHERE clause and 6–4
  arithmetic functions
    ABS function 2–10
    EXP function 2–17
    LN function 2–19
    LOG function 2–20
    NULLIFZERO function 2–21
    RANDOM function 2–23
    SQRT function 2–33
    ZEROIFNULL function 2–37
  arithmetic operators
    addition operator 2–2
    division operator 2–2
    exponentiation 2–2
    MOD operator 2–2
    multiplication operator 2–2
    precedence 2–7
    subtraction operator 2–2
    unary minus operator 2–2
    unary plus operator 2–2
  case blind comparisons 9–4
  CASE operation 1–2
    COALESCE expression 1–25
    NULLIF expression 1–27
    searched CASE 1–7
    valued CASE 1–3
  comparison operators
    = 3–3
    > 3–3
    >= 3–3
    EQ 3–3
    GE 3–3
    GT 3–3
    Japanese character set comparison operators 3–11
    LE 3–3
    LT 3–3
    NE 3–3
  conditional expressions 9–27
  hyperbolic functions 2–54
    ACOSH 2–58
    ASINH 2–60

ATANH 2–62
COSH 2–55
SINH 2–56
TANH 2–57
inverse trigonometric functions 2–39
ACOS 2–46
ASIN 2–48
ATAN 2–50
ATAN2 2–52
logical expressions
BETWEEN 9–10
NOT BETWEEN 9–10
trigonometric functions 2–39
COS 2–40
SIN 2–42
TAN 2–44
SQL functions
attribute functions 10–2
BYTES 10–3
CHARACTER_LENGTH 10–5
CHARACTERS 10–8
FORMAT 10–9
MCHARACTERS 10–2, 10–5
OCTET_LENGTH 10–10
TYPE 10–12
built-in functions
ACCOUNT 12–3
CURRENT_DATE 12–4
CURRENT_TIME 12–5
CURRENT_TIMESTAMP 12–7
DATABASE 12–9
DATE 12–10
PROFILE 12–12
ROLE 12–13
SESSION 12–14
TIME 12–15
USER 12–16
byte strings
BYTES 10–3
byte strings, TRIM 8–47
hash-related functions 11–2
HASHAMP 11–3
HASHBAKAMP 11–5
HASHBUCKET 11–7
HASHROW 11–9
Ordered analytical functions
AVG window function 7–25
COUNT window function 7–31
CSUM 7–37
MAVG 7–40
MAX window function 7–44
MDIFF 7–48
MIN window function 7–52
MLINREG 7–56
MSUM 7–59
PERCENT_RANK window function 7–61
QUANTILE 7–64
RANK 7–67
RANK window function 7–70
ROW_NUMBER window function 7–72
SUM window function 7–74
partitioning functions
CASE_N 2–12
RANGE_N 2–26
string functions 8–2
CHAR2HEXINT 8–12
concatenation operator 8–6
INDEX 8–15
LOWER 8–21
MINDEX 8–23
MSUBSTR 8–33
POSITION 8–23, 8–36
SOUNDEX 8–26
SUBSTR 8–30, 8–33
SUBSTRING 8–30
TRANSLATE_CHK 8–43
TRIM 8–47
TRIM and concatenation 8–48
UPPER 8–51
VARGRAPHIC 8–53
WIDTH_BUCKET function 2–34
SQRT function 2–33
STDDEV_POP aggregate function, described 6–54
STDDEV_SAMP aggregate function, described 6–57
String functions
CHAR2HEXINT 8–12
implicit character type conversion 8–5
INDEX 8–15
LOWER 8–21
MINDEX 8–3, 8–23
MSUBSTR 8–3, 8–33
POSITION 8–23, 8–36
rules 8–5
server character sets and 8–5
SOUNDEX 8–26
SUBSTR 8–30, 8–33
SUBSTRING 8–30
TRANSLATE_CHK 8–43
TRIM 8–47
UPPER 8–51
VARGRAPHIC 8–53
Subqueries, comparison operators and 9–8
SUBSTR function 8–30, 8–33
ANSI equivalent 8–3
SUBSTRING function 8–3, 8–30
Subtraction operator 2–2
SUM aggregate function, described 6–60
SUM function, Interval types and 5–25
SUM window function, described 7–74
Syntax, how to read A–1
SYS_CALENDAR system database

CALENDAR view
cumulative sum, and calculating 7–38
moving difference, and calculating 7–49

T

TAN trigonometric function 2–44
TANH hyperbolic function 2–57
Teradata conversion syntax 13–11
Teradata OLAP functions. See Ordered analytical functions.
Teradata Warehouse Miner 7–3
Time
get current time (Teradata) 12–15
get system time 12–5
Time expressions, Teradata 5–27
TIME function 12–15
Time stamp, get system time stamp 12–7
Time zone comparisons 3–9
Time zone, get time zone displacement 12–5
TIME, conversion to character 13–45
TIMESTAMP
arithmetic 5–19, 5–23
TIMESTAMP, conversion to character 13–50
TIMESTAMP-to-DATE conversion 13–48
TIMESTAMP-to-TIMESTAMP conversion 13–47, 13–49
TIME-to-TIME conversion 13–43
TIME-to-TIMESTAMP conversion 13–44
TRANSLATE_CHK function 8–43
Translation, character 13–32
Trigonometric functions 2–39
COS 2–40
SIN 2–42
TAN 2–44
TRIM function 8–47
TRIM function and concatenation 8–48
TRUE 9–28
Type conversion, implicit 13–3
TYPE function 10–12

U

Unary minus operator 2–2
Unary plus operator 2–2
Unicode, notation A–6
UNION
Effect of sort key and row size on ability to perform 4–28
UNION operator 4–27
outer join and 4–32
Universal Coordinated Time, see UTC
UNKNOWN 9–28
UPPER function 8–51
USER function 12–16
Username, get user name 12–16
UTC
and time conversions 13–59, 13–62

V

VAR_POP aggregate function, described 6–62
VAR_SAMP aggregate functions, described 6–64
VARGRAPHIC function 8–53
VARGRAPHIC function conversion tables 8–56

W

WIDTH_BUCKET function, described 2–34
Wildcards, used with LIKE predicate 9–33
Window
defined 7–7
Window aggregate functions
defined 7–12
difference between aggregate functions and 7–12
Window functions. See Ordered analytical functions.

Z

ZEROIFNULL function 2–37