Teradata Database

SQL Functions, Operators, Expressions, and Predicates

Release 14.10
B035-1145-112A
October 2015

The product or products described in this book are licensed products of Teradata Corporation or its affiliates.

Teradata, Active Data Warehousing, Active Enterprise Intelligence, Applications-Within, Aprimo Marketing Studio, Aster, BYNET, Claraview, DecisionCast, Gridscale, MyCommerce, QueryGrid, SQL-MapReduce, Teradata Decision Experts, "Teradata Labs" logo, Teradata ServiceConnect, Teradata Source Experts, WebAnalyst, and Xkoto are trademarks or registered trademarks of Teradata Corporation or its affiliates in the United States and other countries.
Adaptec and SCSISelect are trademarks or registered trademarks of Adaptec, Inc.
AMD Opteron and Opteron are trademarks of Advanced Micro Devices, Inc.
Apache, Apache Avro, Apache Hadoop, Apache Hive, Hadoop, and the yellow elephant logo are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries.
Apple, Mac, and OS X all are registered trademarks of Apple Inc.
Axeda is a registered trademark of Axeda Corporation. Axeda Agents, Axeda Applications, Axeda Policy Manager, Axeda Enterprise, Axeda Access, Axeda Software Management, Axeda Service, Axeda ServiceLink, and Firewall-Friendly are trademarks and Maximum Results and Maximum Support are servicemarks of Axeda Corporation.
Data Domain, EMC, PowerPath, SRDF, and Symmetrix are registered trademarks of EMC Corporation.
GoldenGate is a trademark of Oracle.
Hewlett-Packard and HP are registered trademarks of Hewlett-Packard Company.
Hortonworks, the Hortonworks logo and other Hortonworks trademarks are trademarks of Hortonworks Inc. in the United States and other countries.
Intel, Pentium, and XEON are registered trademarks of Intel Corporation.
IBM, CICS, RACF, Tivoli, and z/OS are registered trademarks of International Business Machines Corporation.
Linux is a registered trademark of Linus Torvalds.
LSI is a registered trademark of LSI Corporation.
Microsoft, Active Directory, Windows, Windows NT, and Windows Server are registered trademarks of Microsoft Corporation in the United States and other countries.
NetVault is a trademark or registered trademark of Dell Inc. in the United States and/or other countries.
Novell and SUSE are registered trademarks of Novell, Inc., in the United States and other countries.
Oracle, Java, and Solaris are registered trademarks of Oracle and/or its affiliates.
QLogic and SANbox are trademarks or registered trademarks of QLogic Corporation.
Quantum and the Quantum logo are trademarks of Quantum Corporation, registered in the U.S.A. and other countries.
Red Hat is a trademark of Red Hat, Inc., registered in the U.S. and other countries. Used under license.
SAP is the trademark or registered trademark of SAP AG in Germany and in several other countries.
SAS and SAS/C are trademarks or registered trademarks of SAS Institute Inc.
SPARC is a registered trademark of SPARC International, Inc.
Symantec, NetBackup, and VERITAS are trademarks or registered trademarks of Symantec Corporation or its affiliates in the United States and other countries.
Unicode is a registered trademark of Unicode, Inc. in the United States and other countries.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Other product and company names mentioned herein may be the trademarks of their respective owners.

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 TERADATA CORPORATION 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 Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation 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. Teradata Corporation 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 email: teradata-books@lists.teradata.com.
Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed non-confidential. Teradata Corporation 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, Teradata Corporation 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 © 2000 – 2015 by Teradata. All Rights Reserved.

Preface

Purpose

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

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

Audience

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

Supported Software Releases and Operating Systems

This book supports Teradata® Database 14.10.

Teradata Database 14.10 is supported on:

- SUSE Linux Enterprise Server (SLES)10
- SLES 11

Teradata Database client applications support other operating systems.

Prerequisites

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

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

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

Changes to This Book

Release	Description
Teradata Database 14.10 October 2015	Added compatibility information for Calendar functions. Original function names from Release 13.10 were changed in Release 14.0.
Teradata Database 14.10 June 2015	Updated the LOAD_FROM_HCATALOG (SQL-H for Teradata) section in Chapter 26 to add compatibility with MapR 3.1.1 and added information about the isNested name value pair.
Teradata Database 14.10 January 2015	Updated the LOAD_FROM_HCATALOG (SQL-H for Teradata) table operator to add compatibility with Cloudera CDH 4.3 and 5.1. Revised documentation of REGEXP functions in Chapter 23.
Teradata Database 14.10 December 2013	Revised syntax diagram for PERCENTILE_CONT / PERCENTILE_DISC: added parentheses around ORDER BY clause. Revised definition of *value_expression* for the following functions: PERCENTILE_CONT / PERCENTILE_DISC and MEDIAN. Revised documentation of the RESPECT NULLS argument for FIRST_VALUE / LAST_VALUE. Revised documentation for REGEXP_INSTR. *subexpr_arg* is not an argument for this function. Clarified the definition of a derived_period in the syntax for the following Period functions: BEGIN, CONTAINS, END, EQUALS, IMMEDIATELY PRECEDES, and IMMEDIATELY SUCCEEDS. Revised documentation of RESULT VALUE for IMMEDIATELY PRECEDES and IMMEDIATELY SUCCEEDS. “CASE statement” changed to “CASE expression” throughout.
Teradata Database 14.10 September 2013	• Updated documentation and indexing for LOAD_FROM HCATALOG to make clear the association with SQL-H. • Revised documentation of REGEXP_INSTR and REGEXP_REPLACE. • Clarified that the return format is the same as the format of the *dependent_variable_expression* for appropriate REGR functions. • Clarified which rows, when the ROWS phrase is omitted, are returned for FIRST_VALUE/LAST_VALUE. • Revised result type and format documentation of REGR_AVGX and REGR_AVGY. • Revised documentation of ARRAY_AGG. The function does not support NULLS FIRST/NULLS LAST. • Updated explanation of UNNEST example.

Release	Description
Teradata Database 14.10 May 2013	Added the following: • Aggregate functions: DENSE_RANK (ANSI), FIRST_VALUE, LAST_VALUE, MEDIAN, PERCENTILE_CONT and PERCENTILE_DISC. • New ordered-analytical functions. • Support for derived periods, including new period predicate operators: EQUALS, IMMEDIATELY PRECEDES and IMMEDIATELY SUCCEEDES. • Revised syntax for the ARRAY_AGG, PERCENT_RANK, query_expression, RANK(ANSI), ROW_NUMBER, and window aggregate functions.
Teradata Database 14.10 May 2013 (continued)	• Three table operator functions: CALCMATRIX, LOAD_FROM HCATALOG, and TD_UNPIVOT • New business calendar functions. • Two compression functions: TS_COMPRESS and TS_DECOMPRESS. • Revised syntax for the TO_TIMESTAMP function. • Teradata-supported TO_NUMBER syntax. • New syntax diagrams for the following functions: ARRAY Concatenation Operator, GREATEST, LEAST, DECODE, and CSV. • UNICODE normalization functions to TRANSLATE and TRANSLATE_CHK. • New result types of ARRAY_SUM. • New examples of the TO_BYTE and the TO_BYTES functions. • New examples of 3 special *data_time_term* syntax cases. • Clarification about the limitation on the use of TimeZone strings. • Clarification that the CARDINALITY function is an exception to the rule for ARRAY functions called with a scope reference. • Clarification about the possible overflow result with a SELECT Sum. • Clarification that timestamps without a time zone are compared in the local time zone. • Clarification that any arguments other than P, Y, and M for the ISO duration format in the YMINTERVAL function are ignored. • The following functions support compressing / uncompressing LOBS: TD_LZ_COMPRESS and TD_LZ_DECOMPRESS • The following restriction has been removed: that window feature is supported only for aggregate UDFs written in C or C++.
Teradata Database 14.10 November 2011	Renamed calendar functions introduced in Release 13.10. All calendar functions now carry the td_ prefix.

Additional Information

URL	Description
www.info.teradata.com/	Use the Teradata Information Products Publishing Library site to: • View or download a manual: 1 Under **Online Publications**, select **General Search**. 2 Enter your search criteria and click **Search**. • Download a documentation CD-ROM: 1 Under **Online Publications**, select **General Search**. 2 In the **Title or Keyword** field, enter *CD-ROM*, and click **Search**.
www.teradata.com	The Teradata home page provides links to numerous sources of information about Teradata. Links include: • Executive reports, white papers, case studies of customer experiences with Teradata, and thought leadership • Technical information, solutions, and expert advice • Press releases, mentions and media resources
www.teradata.com/t/TEN/	Teradata Customer Education delivers training that builds skills and capabilities for our customers, enabling them to maximize their Teradata investment.
tays.teradata.com/	Use Teradata @ Your Service to access Orange Books, technical alerts, and knowledge repositories, view and join forums, and download software patches.
developer.teradata.com/	Teradata Developer Exchange provides articles on using Teradata products, technical discussion forums, and code downloads.

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 email teradata-books@lists.teradata.com.

Teradata Database Optional Features

This book may include descriptions of the following optional Teradata Database features and products:

- Teradata Columnar
- Teradata Row Level Security
- Teradata SQL-H

- Teradata Temporal
- Teradata Virtual Storage (VS)
- Unity Source Link

You may not use these features without the appropriate licenses. The fact that these features may be included in product media or downloads, or described in documentation that you receive, does not authorize you to use them without the appropriate licenses.

Contact your Teradata sales representative to purchase and enable optional features.

Table of Contents

Preface 3

Purpose 3
Audience 3
Supported Software Releases and Operating Systems 3
Prerequisites 3
Changes to This Book 4
Additional Information 6
Teradata Database Optional Features 6

Chapter 1: Introduction 25

SQL Functions 25
SQL Operators 27
SQL Expressions 28
SQL Predicates 29

Chapter 2: Aggregate Functions 31

Aggregate Functions 31
AVG 36
CORR 39
COUNT 42
COVAR_POP 46
COVAR_SAMP 49
GROUPING 52
KURTOSIS 55
MAX 57
MIN 60
REGR_AVGX 63
REGR_AVGY 66
REGR_COUNT 69

REGR_INTERCEPT 73
REGR_R2 77
REGR_SLOPE 80
REGR_SXX 84
REGR_SXY 87
REGR_SYY 90
SKEW 93
STDDEV_POP 96
STDDEV_SAMP 99
SUM 102
VAR_POP 105
VAR_SAMP 108

Chapter 3: Arithmetic, Trigonometric, Hyperbolic Operators/ Functions 111

Arithmetic Operators 111
Binary Arithmetic Result Data Types 112
Structure of Arithmetic Expressions 122
Arithmetic Functions 123
ABS 124
CASE_N 126
CEILING 136
EXP 138
FLOOR 140
LN 142
LOG 144
NULLIFZERO 146
POWER 149
RANDOM 151
RANGE_N 155
ROUND 169
SIGN 172
SQRT 174
TRUNC 176
WIDTH_BUCKET 178
ZEROIFNULL 182

Trigonometric Functions
(COS, SIN, TAN, ACOS, ASIN, ATAN, ATAN2) 184
DEGREES
RADIANS 187
Hyperbolic Functions
(COSH, SINH, TANH, ACOSH, ASINH, ATANH) 190

Chapter 4: ARRAY / VARRAY Functions and Operators 193

ARRAY Element Reference 194
ARRAY Constructor Expression 197
ARRAY Scope Reference 200
ARRAY_AGG 202
UNNEST 206
CARDINALITY 209
ARRAY Concatenation Operator 212
ARRAY Concatenation Function 214
ARRAY Comparison Functions 217
ARRAY Arithmetic Functions 222
ARRAY_SUM 226
ARRAY_AVG 229
ARRAY_MAX 232
ARRAY_MIN 235
ARRAY_COUNT_DISTINCT 238
ARRAY_GET 241
ARRAY_COMPARE 244
ARRAY_UPDATE 247
ARRAY_UPDATE_STRIDE 250
OEXISTS 253
OCOUNT 256
OLIMIT 259
OFIRST 262
OLAST 265
OPRIOR 268
ONEXT 271
OEXTEND 274
OTRIM 279
ODELETE 282

Chapter 5: Attribute Functions 285

Attribute Function Overview 285
BYTES 286
CHARACTER_LENGTH 288
CHARACTERS 291
DEFAULT 293
FORMAT 297
OCTET_LENGTH 298
TITLE 301
TYPE 303

Chapter 6: Bit/Byte Manipulation Functions 307

Bit and Byte Numbering Model 307
Performing Bit-Byte Operations against Arguments with Non-Equal Lengths 311
BITAND 313
BITNOT 316
BITOR 318
BITXOR 321
COUNTSET 324
GETBIT 326
ROTATELEFT 328
ROTATERIGHT 331
SETBIT 334
SHIFTLEFT 337
SHIFTRIGHT 340
SUBBITSTR 343
TO_BYTE 346

Chapter 7: Built-In Functions 349

ACCOUNT 350
CURRENT_DATE 351
CURRENT_ROLE 355
CURRENT_TIME 357

CURRENT_TIMESTAMP 361
CURRENT_USER 365
DATABASE 366
DATE 367
PROFILE 371
ROLE 372
SESSION 374
TEMPORAL_DATE 375
TEMPORAL_TIMESTAMP. 376
TIME 378
USER 381

Chapter 8: Business Calendars 383

About the Business Calendars 383
About ISO Computation 384
About Business Calendar Macros 386
About the Business Calendar Tables 390
About the Business Calendar Views 391
About the Business Calendar Functions 395
td_week_begin 405
td_week_end 407
td_sunday 409
td_monday 411
td_tuesday 413
td_wednesday 415
td_thursday 417
td_friday 419
td_saturday 421
DayNumber_Of_Week 423
td_month_begin 425
td_month_end 427
DayNumber_Of_Month 429
DayOccurrence_Of_Month 431
WeekNumber_Of_Month 433
td_year_begin 435
td_year_end 437

DayNumber_Of_Year 439
WeekNumber_Of_Year 441
MonthNumber_Of_Year 443
td_quarter_begin 445
td_quarter_end 447
WeekNumber_Of_Quarter 449
MonthNumber_Of_Quarter 451
QuarterNumber_Of_Year 453
DayNumber_Of_Calendar 455
WeekNumber_Of_Calendar 457
MonthNumber_Of_Calendar 459
QuarterNumber_Of_Calendar 461
YearNumber_Of_Calendar 463

Chapter 9: Calendar Functions 465

Calendar Function Compatibility with Previous Releases 465
td_day_of_week 466
td_day_of_month 468
td_day_of_year 470
td_day_of_calendar 472
td_weekday_of_month 474
td_week_of_month 476
td_week_of_year 478
td_week_of_calendar 480
td_month_of_quarter 482
td_month_of_year 484
td_month_of_calendar 486
td_quarter_of_year 488
td_quarter_of_calendar 490
td_year_of_calendar 492

Chapter 10: Comparison Operators and Functions 495

Comparison Operators 495
Comparisons That Produce TRUE Results 498

Data Type Evaluation 500
Implicit Type Conversion of Comparison Operands 502
Comparison of ANSI DateTime and Interval in USING Clause 505
Proper Forms of DATE Types in Comparisons 505
Character String Comparisons 507
Comparison of KANJI1 Characters 510
Comparison Operators and the DEFAULT Function in Predicates 512
DECODE 513
GREATEST 516
LEAST 519

Chapter 11: Compression/Decompression Functions 523

CAMSET 524
CAMSET_L 527
DECAMSET 530
DECAMSET_L 532
LZCOMP 534
LZCOMP_L 536
LZDECOMP 538
LZDECOMP_L 540
TD_LZ_COMPRESS 542
TD_LZ_DECOMPRESS 544
TS_COMPRESS 546
TS_DECOMPRESS 548
TransUnicodeToUTF8 550
TransUTF8ToUnicode 552

Chapter 12: CASE Expressions 555

CASE 555
Valued CASE Expression 556
Searched CASE Expression 561
Error Conditions 565
Rules for the CASE Expression Result Type 566
Format for a CASE Expression 571

CASE and Nulls 572
COALESCE Expression 575
NULLIF Expression 577

Chapter 13: Data Type Conversions 579

Forms of Data Type Conversions 579
Implicit Type Conversions 579
CAST in Explicit Data Type Conversions 586
Teradata Conversion Syntax in Explicit Data Type Conversions 589
Data Conversions in Field Mode 591
Byte Conversion 592
Character-to-Character Conversion 596
Implicit Character-to-Character Translation 599
Character-to-DATE Conversion 601
Character-to-INTERVAL Conversion 606
Character-to-Numeric Conversion 608
Character-to-Period Conversion 614
Character-to-TIME Conversion 617
Character-to-TIMESTAMP Conversion 623
Character-to-UDT Conversion 628
Character Data Type Assignment Rules 630
DATE-to-Character Conversion 631
DATE-to-DATE Conversion 635
DATE-to-Numeric Conversion 637
DATE-to-Period Conversion 640
DATE-to-TIMESTAMP Conversion 642
DATE-to-UDT Conversion 648
INTERVAL-to-Character Conversion 650
INTERVAL-to-INTERVAL Conversion 652
INTERVAL-to-Numeric Conversion 656
INTERVAL-to-UDT Conversion 658
Numeric-to-Character Conversion 660
Numeric-to-DATE Conversion 665
Numeric-to-INTERVAL Conversion 668
Numeric-to-Numeric Conversion 670
Numeric-to-UDT Conversion 674

Period-to-Character Conversion 676
Period-to-DATE Conversion 679
Period-to-Period Conversion 681
Period-to-TIME Conversion 686
Period-to-TIMESTAMP Conversion 688
Signed Zone DECIMAL Conversion 690
TIME-to-Character Conversion 694
TIME-to-Period Conversion 697
TIME-to-TIME Conversion 699
TIME-to-TIMESTAMP Conversion 707
TIME-to-UDT Conversion 721
TIMESTAMP-to-Character Conversion 723
TIMESTAMP-to-DATE Conversion 727
TIMESTAMP-to-Period Conversion 738
TIMESTAMP-to-TIME Conversion 740
TIMESTAMP-to-TIMESTAMP Conversion 748
TIMESTAMP-to-UDT Conversion 756
UDT-to-Byte Conversion 758
UDT-to-Character Conversion 761
UDT-to-DATE Conversion 765
UDT-to-INTERVAL Conversion 768
UDT-to-Numeric Conversion 771
UDT-to-TIME Conversion 774
UDT-to-TIMESTAMP Conversion 777
UDT-to-UDT Conversion 780

Chapter 14: Data Type Conversion Functions 783

TO_BYTES 784
FROM_BYTES 786
TO_NUMBER 788
TO_CHAR(Numeric) 793
TO_CHAR(DateTime) 795
TO_DATE 797
TO_TIMESTAMP 803
TO_TIMESTAMP_TZ 805
TO_YMINTERVAL 807

TO_DSINTERVAL 810
NUMTODSINTERVAL 813
NUMTOYMINTERVAL 815

Chapter 15: DateTime and Interval Functions and Expressions 817

Overview 817
ANSI DateTime and Interval Data Type Assignment Rules 818
Scalar Operations on ANSI SQL:2011 DateTime and Interval Values. 821
ANSI DateTime Expressions 822
ANSI Interval Expressions 830
Arithmetic Operators and ANSI DateTime and Interval Data Types 838
Aggregate Functions and ANSI DateTime and Interval Data Types 839
Scalar Operations and DateTime Functions. 840
Teradata Date and Time Expressions 841
Scalar Operations on Teradata DATE Values 843
LAST_DAY 844
NEXT_DAY 846
MONTHS_BETWEEN 848
ADD_MONTHS 850
OADD_MONTHS 856
TRUNC(Date) 859
ROUND(Date) 862
EXTRACT 864
GetTimeZoneDisplacement 868

Chapter 16: Export Width Procedures 875

About Export Widths 875
ReplaceExportDefinition 877
RemoveExportDefinition 881

Chapter 17: Hash-Related Functions 883

HASHAMP 884

HASHBAKAMP 887
HASHBUCKET 890
HASHROW 893

Chapter 18: LOB Functions 895

EMPTY_BLOB 896
EMPTY_CLOB 897

Chapter 19: Logical Predicates 899

Logical Predicates 899
Logical Operators and Search Conditions 902
ANY / ALL / SOME Quantifiers 907
BETWEEN / NOT BETWEEN 912
EXISTS / NOT EXISTS 913
IN/NOT IN 919
IS NULL / IS NOT NULL 926
LIKE 928
OVERLAPS 938

Chapter 20: Null-Handling Functions 939

NVL 940
NVL2 943

Chapter 21: Ordered Analytical / Window Aggregate Functions 945

Ordered Analytical Functions 945
Characteristics of Ordered Analytical Functions 947
The Window Feature 953
Window Aggregate Functions 960
CSUM 984
CUME_DIST 987

DENSE_RANK (ANSI) 990
FIRST_VALUE / LAST_VALUE 993
MAVG 996
MDIFF 999
MEDIAN 1002
MLINREG 1003
MSUM 1006
PERCENT_RANK. 1008
PERCENTILE_CONT / PERCENTILE_DISC. 1012
QUANTILE 1015
RANK (ANSI) 1018
RANK (Teradata) 1022
ROW_NUMBER. 1025

Chapter 22: Period Functions and Operators 1029

Period Value Constructor 1030
Arithmetic Operators 1033
Comparison of Period Types 1034
BEGIN 1037
CONTAINS 1039
END 1042
EQUALS. 1044
IS UNTIL_CHANGED/IS NOT UNTIL_CHANGED 1047
IS UNTIL_CLOSED/IS NOT UNTIL_CLOSED 1049
IMMEDIATELY PRECEDES. 1051
IMMEDIATELY SUCCEEDS 1054
INTERVAL 1057
LAST. 1060
MEETS 1062
NEXT 1065
OVERLAPS 1067
P_INTERSECT 1072
P_NORMALIZE 1074
PRECEDES 1076
PRIOR 1079
LDIFF 1081

RDIFF 1083
SUCCEEDS 1085
TD_NORMALIZE_OVERLAP 1088
TD_NORMALIZE_MEET 1090
TD_NORMALIZE_OVERLAP_MEET 1092
TD_SUM_NORMALIZE_OVERLAP 1094
TD_SUM_NORMALIZE_MEET 1096
TD_SUM_NORMALIZE_OVERLAP_MEET 1098
TD_SEQUENCED_SUM 1100
TD_SEQUENCED_AVG 1102
TD_SEQUENCED_COUNT 1104

Chapter 23: Regular Expression Functions 1107

REGEXP_SUBSTR 1108
REGEXP_REPLACE 1111
REGEXP_INSTR 1115
REGEXP_SIMILAR 1118
REGEXP_SPLIT_TO_TABLE 1121

Chapter 24: Set Operators 1125

Overview of Set Operators 1125
Rules for Set Operators 1127
Precedence of Set Operators 1128
Retaining Duplicate Rows Using the ALL Option 1129
Attributes of a Set Result 1129
Set Operators With Derived Tables 1131
Set Operators in Subqueries 1133
Set Operators in INSERT … SELECT Statements 1134
Set Operators in View Definitions 1135
Queries Connected by Set Operators 1137
INTERSECT Operator 1141
MINUS/EXCEPT Operator 1144
UNION Operator 1146

Chapter 25: String Operators and Functions 1155

String Function Overview 1155
Effects of Server Character Sets on Character String Functions 1157
Concatenation Operator 1159
ASCII 1165
CHAR2HEXINT 1167
CHR 1170
CSV 1172
CSVLD 1174
EDITDISTANCE 1176
INDEX 1179
INITCAP 1185
INSTR 1187
LENGTH 1189
LOWER 1191
LPAD 1193
LTRIM 1195
NGRAM 1197
NVP 1200
OREPLACE 1203
OTRANSLATE 1206
POSITION 1209
RPAD 1212
RTRIM 1214
SOUNDEX 1216
STRING_CS 1220
STRTOK 1223
STRTOK_SPLIT_TO_TABLE 1225
SUBSTRING/SUBSTR 1228
TRANSLATE 1234
TRANSLATE_CHK 1244
TRIM 1248
UPPER 1252
VARGRAPHIC 1255
VARGRAPHIC Function Conversion Tables 1258

Chapter 26: Table Operators 1269

CALCMATRIX 1270
LOAD_FROM_HCATALOG (Teradata SQL-H) 1276
TD_UNPIVOT 1285

Chapter 27: User-Defined Functions 1289

SQL UDF 1290
Scalar UDF 1295
Aggregate UDF 1299
Window Aggregate UDF 1302
Table UDF 1310

Chapter 28: User-Defined Type Expressions/Methods 1313

UDT Expression 1314
NEW 1318
NEW VARIANT_TYPE 1321
Method Invocation 1324

Appendix A: Notation Conventions 1329

Syntax Diagram Conventions 1329
Character Shorthand Notation Used In This Book 1334
Predicate Calculus Notation Used In This Book 1336

Appendix B: Performance Considerations 1337

Using CASE Expressions to Optimize Query Design 1337
Working with Analytical Functions 1340
Using the BETWEEN Clause to Reduce Row Redistribution 1342

Glossary 1345

Index 1347

CHAPTER 1 Introduction

This chapter introduces SQL functions, operators, expressions, and predicates.

For information on special functions, operators, predicates, and methods for geospatial data, see *SQL Geospatial Types*.

For information on the XML data type and XML operations, functions, and methods, see *Teradata XML*.

SQL Functions

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

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

Types of SQL Functions

Function Type	Definition
Scalar	The arguments are individual scalar values of either the same or mixed type that can have different meanings. The result is a single value or null. Scalar functions can be used in any SQL statement where an expression can be used.
Aggregate	The argument is a group of rows. The result is a single value or null. Normally aggregate functions are used in the expression list of a SELECT statement and in the summary list of a WITH clause.
Table	The arguments are individual scalar values of either same or mixed type that can have different meanings. The result is a table. Table functions can be used only within the FROM clause of a SELECT statement. Table functions are a form of user-defined functions and are described in *SQL External Routine Programming*.

Function Type	Definition
Ordered Analytical	The arguments are any normal SQL expression. The result is handled the same way as any other SQL expression. It can be a result column or part of a more complex arithmetic expression. Ordered analytical functions are used in operations that require an ordered set of results rows or that depend on values in a previous row.

Examples of Functions

Function	Description
`SELECT CHARACTER_LENGTH(Details) FROM Orders;`	Scalar function taking the character or CLOB value in the Details column and returning a numeric value for each row in the Orders table.
`SELECT AVG(Salary) FROM Employee;`	Aggregate function returning a single numeric value for the group of numeric values specified by the Salary column in the Employee table.

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

Embedded Services System Functions

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

Activating Embedded Services System Functions

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

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

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

Invoking Embedded Services System Functions

You can invoke an embedded services function using the function name alone. For example, CEILING (*arg*).

You can also qualify the function name by adding the TD_SYSFNLIB database name. For example, you can invoke the CEILING function using the fully qualified syntax, TD_SYSFNLIB.CEILING(*arg*).

Note: If you try to invoke an embedded services function using the function name alone, but you also have a customer-developed UDF with the same name in the current database or in the SYSLIB database, Teradata Database will execute the customer-developed UDF instead of the embedded services function.

Therefore, to ensure that you are invoking the embedded services function, do one of the following:

- Remove any customer-developed UDFs with the same name from the normal UDF search path; that is, from the current database and from the SYSLIB database. For detailed information, see "Locations Where Teradata Database Looks for Functions" in *SQL External Routine Programming*.
- Use the fully qualified syntax to invoke the embedded services function. For example, TD_SYSFNLIB.*embedded_services_function*.

Implicit Data Type Conversion Rules

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

SQL Operators

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

Types of Operators

The following types of operators are available in SQL:

- Arithmetic operators such as + and - operate on numeric, DateTime, and Interval data types.
- The concatenation operator || operates on character and byte types.
- Comparison operators such as = and > test the truth of relations between their arguments. For information on comparison operator, see Chapter 10: "Comparison Operators and Functions."

  Because comparison operators are a type of logical predicate, see Chapter 19: "Logical Predicates." Comparison operators are also known as conditional expressions because they result in a value of TRUE, FALSE, or unknown (NULL).
- Set operators, or relational operators, such as INTERSECT and UNION combine result sets from multiple sources into a single result set.

SQL Expressions

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

- Column name
- Literal (sometimes referred to as a constant)
- Function
- USING variable
- Parameter
- Parameter marker (question mark (?) placeholder)
- Combination of column names, literals, and functions connected by operators

Types of Expressions

SQL expressions generally fall into the following categories.

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

Examples of Expressions

Expression	Description
`'Test Tech'`	Character string literal

Expression	Description
`1024`	Numeric literal
`Employee.FirstName`	Column name
`Salary * 12 + 100`	Arithmetic expression producing a numeric value
`INTERVAL '10' MONTH * 4`	Interval expression producing an interval value
`CURRENT_DATE + INTERVAL '2' DAY`	DateTime expression producing a DATE value
`CURRENT_TIME - INTERVAL '1' HOUR`	DateTime expression producing a TIME value
`'Last' || ' Order'`	String expression producing a character string value

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

SQL Predicates

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

- TRUE
- FALSE
- NULL (or unknown)

Predicates can appear in the following:

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

Types of Logical Predicates

SQL provides the following logical predicates:

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

Logical Operators that Operate on Predicates

- NOT
- AND
- OR

Predicate Quantifiers

- SOME
- ANY
- ALL

Examples of Predicates

Predicate	Description

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

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

CHAPTER 2 Aggregate Functions

This chapter describes SQL aggregate functions.

For information on:

- Window aggregate functions and their Teradata-specific equivalents, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."
- Aggregate user-defined functions (UDFs), see "Aggregate UDF" on page 1299.
- Window aggregate UDFs, see "Window Aggregate UDF" on page 1302.

Aggregate Functions

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

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

```
SELECT SUM(Total_Sales)
FROM Sales_Table;

Sum(Total_Sales)
----------------
         5192.40
```

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

```
SELECT Product_ID, SUM(Total_Sales)
FROM Sales_Table
GROUP BY Product_ID;

Product_ID  Sum(Total_Sales)
----------  ----------------
       101           2100.00
       107           1000.40
       102           2092.00
```

Aggregates in the Select List

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

Aggregates and GROUP BY

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

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

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

Aggregates and Date

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

Aggregates and Literal Expressions in the Select List

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

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

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

IF the literal expression ...	THEN the result of the literal expression in the query result is ...
does not contain a column reference	the value of the literal expression.

is a non-deterministic function, such as RANDOM	Functions such as RANDOM are computed in the immediate retrieve step of the request instead of in the aggregation step.
	Here is an example:
	`SELECT COUNT(*),`
	`SUBSTRING(CAST(CURRENT_TIME(0) AS CHAR(14))`
	`FROM 1 FOR 8)`
	`FROM Sales_Table;`
	
	`Count(*) Substring(Current Time(0) From 1 For 8)`
	`-------- --------------------------------------`
	`  0 09:01:43`

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

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

is a UDF	

Nesting Aggregates

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

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

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

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

For details on aggregate window functions, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."

Results of Aggregation on Zero Rows

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

This form of aggregate function ...	Returns this result when there are zero rows ...
COUNT(*expression*) WHERE …	0
all other forms of *aggregate_operator*(*expression*) WHERE …	Null
aggregate_operator(*expression*) … GROUP BY …	No Record Found
aggregate_operator(*expression*) … HAVING …	

Aggregates and Nulls

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

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

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

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

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

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

For examples that illustrate this behavior, see "Example 2" on page 44 and "Example 3" on page 44. 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)).

Aggregate Operations on Floating Point Data

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

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

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

Aggregates and LOBs

Aggregates do not operate on CLOB or BLOB data types.

Aggregates and Period Data Types

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

Aggregates and SELECT AND CONSUME Statements

Aggregates cannot appear in SELECT AND CONSUME statements.

Aggregates and Recursive Queries

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

Aggregates in WHERE and HAVING Clauses

Aggregates can appear in the following types of clauses:

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

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

DISTINCT Option

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

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

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

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

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

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

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

Aggregates and Row Level Security Tables

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

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

For more information about row level security, see *Security Administration*.

AVG

Purpose

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

Syntax

```
— AVERAGE —┐  (┌──────────────┐ value_expression ) —
— AVG ————┤   │ DISTINCT ────┤
— AVE ————┘   └──── ALL ─────┘
                                  1101B410
```

where:

Syntax element ...	Specifies ...
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the average computation for the group.
DISTINCT	that null and duplicate values specified by *value_expression* are eliminated from the average computation for the group.
value_expression	a literal or column expression for which an average is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

AVERAGE and AVE are Teradata extensions to the ANSI standard.

Result Type and Attributes

The following table lists the default attributes for the result of AVG.

Attribute	Value
Data Type	REAL
Title	Average(x)
Format	If the operand is numeric, date, or interval, the format is the same format as x. If the operand is character, the format is the default format for FLOAT. If the operand is a UDT, the format is the format for the data type to which the UDT is implicitly cast.

For an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Computation of INTEGER or DECIMAL Values

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

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

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

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

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

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

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

Restrictions

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

AVG is valid only for numeric data.

Nulls are not included in the result computation. For more information, see *SQL Fundamentals* and "Aggregates and Nulls" on page 33.

Example

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

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

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

AVG Window Function

For the AVG window function that computes a group, cumulative, or moving average, see "Window Aggregate Functions" on page 960.

CORR

Purpose

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

Syntax

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

1101B217

where:

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

ANSI Compliance

This is ANSI SQL:2011 compliant.

Definition

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

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

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

IF the correlation coefficient has this value ...	THEN the association between the variables ...
-1.00	is perfectly linear, but inverse. As the value for y varies, the value for x varies identically in the opposite direction.
0	does not exist and they are said to be uncorrelated.
+1.00	is perfectly linear. As the value for y varies, the value for x varies identically in the same direction.

IF the correlation coefficient has this value ...	THEN the association between the variables ...
NULL	cannot be measured because there are no non-null data point pairs in the data used for the computation.

Computation

The equation for computing CORR is defined as follows:

$$\text{CORR} = \frac{\text{COVAR_SAMP}(x,y)}{\text{STDDEV_SAMP}(x)\text{STDDEV_SAMP}(y)}$$

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 “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Combination With Other Functions

CORR can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For information on ordered analytical functions, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."

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

Example

This example uses the data from the HomeSales table.

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

Consider the following query.

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

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

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

CORR Window Function

For the CORR window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

COUNT

Purpose

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

Syntax

```
── COUNT ──── ( ┬──────────────┬ value_expression ──┬ ) ──
                │ ┬ DISTINCT ┬ │                    │
                │ └── ALL ───┘ │                    │
                └──────── * ───┘                    │
```

1101A411

where:

Syntax element ...	Specifies ...
ALL	that all non-null values of *value_expression*, including duplicates, are included in the total count. This is the default.
DISTINCT	that a *value_expression* that evaluates to NULL or to a duplicate value does not contribute to the total count.
value_expression	a literal or column expression for which the total count is computed. The expression cannot contain any ordered analytical or aggregate functions.
*	to count all rows in the group of rows on which COUNT operates.

Usage Notes

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

For COUNT functions that return the group, cumulative, or moving count, see "Window Aggregate Functions" on page 960.

COUNT is valid for any data type.

With the exception of COUNT(*), the computation does not include nulls. For more information, see *SQL Fundamentals* and "Aggregates and Nulls" on page 33.

For an example that uses COUNT(*) and nulls, see "Example 2" on page 44.

Result Type and Attributes

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

Mode	Data Type and Format
ANSI	If MaxDecimal in DBSControl is… • 0 or 15, then the result type is DECIMAL(15,0) and the format is -(15)9. • 18, then the result type is DECIMAL(18,0) and the format is -(18)9. • 38, then the result type is DECIMAL(38,0) and the format is -(38)9.
Teradata	INTEGER and the format is the default format for INTEGER.

For information on data type default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

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

Teradata mode uses INTEGER to avoid regression problems.

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

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

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

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

Example 1

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

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

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

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

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

Example 2

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

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

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

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

Example 3

If you ran the report in Example 2 using SELECT... COUNT … without grouping the results by department number, the results table would have only registered non-null occurrences of DeptNo and would not have included the two employees not yet assigned to a department(nulls). The counts differ (23 in Example 2 as opposed to 21 using the statement documented in this example).

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

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

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

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

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

Example 4

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

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

The result is as follows.

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

Example 5

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

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

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

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

Example 6

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

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

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

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

Example 7

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

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

The system responds with the following report.

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

COVAR_POP

Purpose

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

Syntax

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

1101B216

where:

Syntax element ...	Specifies ...
value_expression_2	a numeric expression to be paired with a second numeric expression to determine their covariance.
value_expression_1	The expressions cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 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 21: “Ordered Analytical / Window Aggregate 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 are as follows.

Data type: REAL

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

For information on the default format of data types and an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

COVAR_POP Window Function

For the COVAR_POP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

COVAR_SAMP

Purpose

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

Syntax

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

1101A456

where:

Syntax element ...	Specifies ...
value_expression_2	a numeric expression to be paired with a second numeric expression to determine their covariance.
value_expression_1	The expressions cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 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 21: "Ordered Analytical / Window Aggregate 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: REAL

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

For information on the default format of data types and an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

COVAR_SAMP Window Function

For the COVAR_SAMP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

Example

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

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

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

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

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

GROUPING

Purpose

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

Syntax

```
─── GROUPING ─── ( expression ) ───
```

1101A461

where:

Syntax element ...	Specifies ...
expression	A column in the result row that might have been excluded from a grouped query containing CUBE, ROLLUP, or GROUPING SET. The argument must be an item of a GROUP BY clause.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Usage Notes

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

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

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

Result Type and Attributes

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

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

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Result Value

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

Example

Suppose you have the following data in the sales_view table.

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

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

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

The query reports the following data:

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

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

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

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

The results are:

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

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

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

The query reports the following data:

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

Related Topics

For more information on GROUP BY, GROUPING SETS, ROLLUP, and CUBE, see *SQL Data Manipulation Language*.

KURTOSIS

Purpose

Returns the kurtosis of the distribution of *value_expression*.

Syntax

```
── KURTOSIS ── ( ┌──────────────┐ value_expression ) ──
               │┌─ DISTINCT ─┐│
               └└─ ALL ──────┘┘
                                          1101B215
```

where:

Syntax element ...	Specifies ...
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates specified by *value_expression* from the computation.
value_expression	a literal or column expression for which the kurtosis of the distribution of its values is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

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

Definition

Kurtosis is the fourth moment of a distribution. It is a measure of the relative peakedness or flatness compared with the normal, Gaussian distribution.

The normal distribution has a kurtosis of 0.

Positive kurtosis indicates a relative peakedness of the distribution, while negative kurtosis indicates a relative flatness.

Result Type and Attributes

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

Data Type	Format	Title
REAL	Default format of the REAL data type	Kurtosis(x)

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: “Data Type Conversions.”

Computation

The equation for computing KURTOSIS is defined as follows:

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

where:

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

Conditions That Produce a NULL Return Value

The following conditions produce a null return value:

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

MAX

Purpose

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

Syntax

```
 MAXIMUM ┬─ ( ┬──────────────┬ value_expression ) ─
 MAX     ┘   │ DISTINCT     │
             └ ALL          ┘
```

1101B412

where:

Syntax element ...	Specifies ...
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the maximum value computation for the group. This is the default.
DISTINCT	that duplicate and non-null values specified by *value_expression* are eliminated from the maximum value computation for the group.
value_expression	a literal or column expression for which the maximum value is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

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

Result Type and Attributes

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

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

Support for UDTs

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

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

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Usage Notes

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

Nulls are not included in the result computation. For more information, see *SQL Fundamentals* and "Aggregates and Nulls" on page 33.

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

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

MAX Window Function

For the MAX window function that computes a group, cumulative, or moving maximum value, see "Chapter 21 Ordered Analytical / Window Aggregate Functions" on page 945.

Example 1: CHARACTER Data

The following SELECT returns the immediately following result.

```
SELECT MAX(Name)
FROM Employee;

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

Example 2: Column Expressions

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

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

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

MIN

Purpose

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

Syntax

```
 MINIMUM ┬─┐ ( ┌──────────────┐ value_expression ) ─
 MIN     ┘   │ DISTINCT     │
             └ ALL          ┘
                                        1101B413
```

where:

Syntax element ...	Specifies ...
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the minimum value computation for the group. This is the default.
DISTINCT	that duplicate and non-null values specified by *value_expression* are eliminated from the minimum value computation for the group.
value_expression	a literal or column expression for which the minimum value is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

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

Result Type and Attributes

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

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

Support for UDTs

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

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

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Usage Notes

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

The computation does not include nulls. For more information, see "Manipulating Nulls" in *SQL Fundamentals* and "Aggregates and Nulls" on page 33.

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

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

MIN Window Function

For the MIN window function that computes a group, cumulative, or moving minimum value, see "Window Aggregate Functions" on page 960.

Example 1: MINIMUM Used With CHARACTER Data

The following SELECT returns the immediately following result.

```
SELECT MINIMUM(Name)
FROM Employee;

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

Example 2: JIT Inventory

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

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

The report is as follows:

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

REGR_AVGX

Purpose

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

Syntax

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

1101B414

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. A dependent variable is something that is measured in response to a treatment. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Setting Up Axes for Plotting

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

Combination With Other Functions

REGR_AVGX can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: "Ordered Analytical / Window Aggregate 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:

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

where:

This variable ...	Represents ...
x	*independent_variable_expression* x is the independent, or predictor, variable expression.
n	COUNT(x)

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

Data type: REAL

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

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

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_AVGX Window Function

For the REGR_AVGX window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

Example

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

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

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

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

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

REGR_AVGY

Purpose

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

Syntax

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

1101B415

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. A dependent variable is something that is measured in response to a treatment. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Setting Up Axes for Plotting

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

Combination With Other Functions

REGR_AVGY can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: “Ordered Analytical / Window Aggregate 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_variable_expression* y is the dependent, or response, variable expression.
n	COUNT(y)

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

Data type: REAL

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

For information on the default format of data types and an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_AVGY Window Function

For the REGR_AVGY window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

Example

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

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

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

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

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

REGR_COUNT

Purpose

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

Syntax

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

1101B416

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. A dependent variable is something that is measured in response to a treatment. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Setting Up Axes for Plotting

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

Combination With Other Functions

REGR_COUNT can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."

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

Result Type and Attributes

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

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

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

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

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

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

- If operand y is numeric or character, the format is:
  - For ANSI mode, if MaxDecimal in DBSControl is:
    - 0 or 15, the format is -(15)9
    - 18, the format is -(18)9
    - 38, the format is -(38)9
  - For Teradata mode, the format is the default format for INTEGER
- If operand y is UDT, the format is the format for the data type to which the UDT is implicitly cast.

For information on data type default formats, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_COUNT Window Function

For the REGR_COUNT window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

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;
```

Here is the result:

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

REGR_INTERCEPT

Purpose

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

Syntax

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

1101B417

where:

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

ANSI Compliance

This is ANSI SQL:2011 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 21: "Ordered Analytical / Window Aggregate 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_variable_expression*
y	*dependent_variable_expression*

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

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

For information on the default format of data types and an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For details on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_INTERCEPT Window Function

For the REGR_INTERCEPT window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

Example

This example uses the data from the HomeSales table.

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

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

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

Here is the result:

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

REGR_R2

Purpose

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

Syntax

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

1101B418

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. A dependent variable is something that is measured in response to a treatment. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Setting Up Axes for Plotting

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

Combination With Other Functions

REGR_R2 can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: "Ordered Analytical / Window Aggregate 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_variable_expression* x is the independent, or predictor, variable expression.
y	*dependent_variable_expression* y is the dependent, or response, variable expression.

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

Data type: REAL

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

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

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_R2 Window Function

For the REGR_R2 window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

Example

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

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

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

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

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

REGR_SLOPE

Purpose

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

Syntax

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

1101B419

where:

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

ANSI Compliance

This is ANSI SQL:2011 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 21: "Ordered Analytical / Window Aggregate 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_variable_expression*
y	*dependent_variable_expression*

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

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

For information on the default format of data types and the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_SLOPE Window Function

For the REGR_SLOPE window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

Example

This example uses the data from the HomeSales table.

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

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

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

Here is the result:

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

REGR_SXX

Purpose

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

Syntax

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

1101B420

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. A dependent variable is something that is measured in response to a treatment. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Setting Up Axes for Plotting

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

Combination With Other Functions

REGR_SXX can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: “Ordered Analytical / Window Aggregate Functions.”

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

Computation

The equation for computing REGR_SXX is defined as follows:

$$\text{REGR_SXX} = (\text{SUM}(x^{**}2)) - \left(\text{SUM}(x) \bullet \left(\frac{\text{SUM}(x)}{n}\right)\right)$$

where:

This variable ...	Represents ...
x	*independent_variable_expression* x is the independent, or predictor, variable expression.
n	COUNT(x)

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

Data type: REAL

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

For information on the default format of data types and an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_SXX Window Function

For the REGR_SXX window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

Example

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

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

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

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

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

REGR_SXY

Purpose

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

Syntax

——— REGR_SXY — (*dependent_variable_expression, independent_variable_expression*) ——

1101B421

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. A dependent variable is something that is measured in response to a treatment. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Setting Up Axes for Plotting

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

Combination With Other Functions

REGR_SXY can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."

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

Computation

The equation for computing REGR_SXY is defined as follows:

$$\text{REGR_SXY} = (\text{SUM}(x*y)) - \left((\text{SUM}(x)) \bullet \left(\frac{\text{SUM}(y)}{n}\right)\right)$$

This variable ...	Represents ...
x	*independent_variable_expression* x is the independent, or predictor, variable expression.
y	*dependent_variable_expression* y is the dependent, or response, variable expression.
n	COUNT(x,y)

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

Data type: REAL

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

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_SXY Window Function

For the REGR_SXY window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

Example

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

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

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

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

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

REGR_SYY

Purpose

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

Syntax

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

1101B422

where:

Syntax element ...	Specifies ...
dependent_variable_expression	the dependent variable for the regression. A dependent variable is something that is measured in response to a treatment. The expression cannot contain any ordered analytical or aggregate functions.
independent_variable_expression	the independent variable for the regression. An independent variable is a treatment: something that is varied under your control to test the behavior of another variable. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Setting Up Axes for Plotting

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

Combination With Other Functions

REGR_SYY can be combined with any of the ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: “Ordered Analytical / Window Aggregate Functions.”

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

Computation

The equation for computing REGR_SYY is defined as follows:

$$\text{REGR_SYY} = (\text{SUM}(y^{**}2)) - \left(\text{SUM}(y) \bullet \left(\frac{\text{SUM}(y)}{n}\right)\right)$$

where:

This variable ...	Represents ...
y	*dependent_variable_expression* y is the dependent, or response, variable expression.
n	COUNT(y)

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

Data type: REAL

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

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

REGR_SYY Window Function

For the REGR_SYY window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

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

```
── SKEW ─ ( ┌──────────────┐ value_expression ) ──
        ├─ DISTINCT ─┤
        └──── ALL ───┘
                                  1101B428
```

where:

Syntax element ...	Specifies ...
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the computation for the group. This is the default.
DISTINCT	that null and duplicate values specified by *value_expression* are eliminated from the computation for the group.
value_expression	a literal or column expression for which the skewness of the distribution of its values is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Definition

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

The normal distribution has a skewness of 0.

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

Result Type and Attributes

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

Data Type	Format	Title
REAL	Default format of the REAL data type	SKEW(x)

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Computation

The equation for computing SKEW is defined as follows:

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

where:

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

Conditions That Produce a Null Result

The following conditions product a null result:

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

STDDEV_POP

Purpose

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

Syntax

```
── STDDEV_POP ── ( ┌──────────────┐ value_expression ) ──
                   ├─ DISTINCT ─┤
                   └─ ALL ──────┘
                                                1101B424
```

where:

Syntax element ...	Specifies ...
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates of *value_expression* from the computation.
value_expression	a numeric literal or column expression whose population standard deviation is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 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 21: “Ordered Analytical / Window Aggregate Functions.”

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

How GROUP BY Affects Report Breaks

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

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

Measuring the Standard Deviation of a Population

If your data represents only a sample of the entire population for the variable, then use the STDDEV_SAMP function. For information, see "STDDEV_SAMP" on page 99.

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

Computation

The equation for computing STDDEV_POP is as follows:

$$\text{STDDEV_POP} = \text{SQRT}\left(\frac{\text{COUNT(x)SUM(x**2)} - \text{(SUM(x)**2)}}{\text{(COUNT(x)**2)}}\right)$$

where:

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

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

Data type: REAL

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

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

STDDEV_POP Window Function

For the STDDEV_POP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

STDDEV_SAMP

Purpose

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

Syntax

```
── STDDEV_SAMP ── ( ┌──────────────┐ value_expression ) ──
                    ├─ DISTINCT ─┤
                    └──── ALL ───┘
                                                1101B425
```

where:

Syntax element ...	Specifies ...
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates of *value_expression* from the computation.
value_expression	a numeric literal or column expression whose sample standard deviation is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Definition

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

Computation

The equation for computing STDDEV_SAMP is as follows:

$$\text{STDDEV_SAMP} = \text{SQRT}\left(\frac{\text{COUNT(x)SUM(x**2)} - \text{(SUM(x)**2)}}{\text{COUNT(x)(COUNT(x)} - 1)}\right)$$

where:

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

Division by zero results in NULL rather than an error.

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

Result Type and Attributes

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

Data type: REAL

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

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Combination With Other Functions

STDDEV_SAMP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."

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

How GROUP BY Affects Report Breaks

The GROUP BY clause affects the STDDEV_SAMP operation.

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

Measuring the Standard Deviation of a Population

If your data represents the entire population for the variable, then use the STDDEV_POP function. For information, see "STDDEV_POP" on page 96.

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

STDDEV_SAMP Window Function

For the STDDEV_SAMP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

SUM

Purpose

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

Syntax

```
— SUM ———— ( ———————————— value_expression ) —
             ┌— DISTINCT —┐
             └——— ALL ———┘
                                  1101B423
```

where:

Syntax element ...	Specifies ...
ALL	that all non-null values specified by *value_expression*, including duplicates, are included in the sum computation for the group. This is the default.
DISTINCT	that duplicate and non-null values specified by *value_expression* are eliminated from the sum computation for the group.
value_expression	a literal or column expression for which the sum is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Result Type and Attributes

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

Data Type of Operand	Data Type of Result	Format	Title
BYTEINT or SMALLINT	Same as the operand	Default format of the INTEGER data type	Sum(x)
character	Same as the operand	Default format for FLOAT	
UDT	Same as the operand	Format for the data type to which the UDT is implicitly cast	

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

For an explanation of the formatting characters in the format, and information on data type default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Usage Notes

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

SUM is valid only for numeric data.

Nulls are not included in the result computation. For details, see "Manipulating Nulls" in *SQL Fundamentals* and "Aggregates and Nulls" on page 33.

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

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

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

Possible Result Overflow with SELECT Sum

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

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

The following query:

```
SELECT SUM (Salary) FROM Employee;
```

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

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

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

SUM Window Function

For the SUM function that returns the cumulative, group, or moving sum, see "Window Aggregate Functions" on page 960.

Example 1: Accounts Receivable

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

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

Example 2: Face Value of Inventory

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

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

Sum((QUANTITY * Price))
-----------------------
          38,525,151.91
```

VAR_POP

Purpose

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

Syntax

```
── VAR_POP ─ ( ┌──────────────┐ value_expression ) ──
            │ DISTINCT │
            └──── ALL ─┘
                                          1101B426
```

where:

Syntax element ...	Specifies ...
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates of *value_expression* from the computation.
value_expression	a numeric literal or column expression whose population variance is to be computed. The expression cannot contain any ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Definition

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

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

Computation

The equation for computing VAR_POP is as follows:

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

where:

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

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

Division by zero results in NULL rather than an error.

Result Type and Attributes

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

Data type: REAL

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

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Combination With Other Functions

VAR_POP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."

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

GROUP BY Affects Report Breaks

The GROUP BY clause affects the VAR_POP operation.

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

Measuring the Standard Deviation of a Population

If your data represents the only a sample of the entire population for the variable, then use the VAR_SAMP function. For information, see "VAR_SAMP" on page 108.

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

VAR_POP Window Function

For the VAR_POP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

VAR_SAMP

Purpose

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

Syntax

```
── VAR_SAMP ── ( ┌──────────────┐ value_expression ) ──
               │ DISTINCT     │
               └──── ALL ─────┘
                                          1101B427
```

where:

Syntax element ...	Specifies ...
ALL	to include all non-null values specified by *value_expression*, including duplicates, in the computation. This is the default.
DISTINCT	to exclude duplicates of *value_expression* from the computation.
value_expression	a numeric literal or column expression whose sample variance is to be computed. The expression cannot contain ordered analytical or aggregate functions.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Definition

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

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

Computation

The equation for computing VAR_SAMP is as follows:

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

where:

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

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

Division by zero results in NULL rather than an error.

Combination With Other Functions

VAR_SAMP can be combined with ordered analytical functions in a SELECT list, QUALIFY clause, or ORDER BY clause. For more information on ordered analytical functions, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."

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

GROUP BY Affects Report Breaks

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

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

Measuring the Variance of a Population

If your data represents the entire population for the variable, then use the VAR_POP function. For information, see "VAR_POP" on page 105.

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 are as follows.

Data type: REAL

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

Support for UDTs

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

- Numeric
- Character
- DATE
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

VAR_SAMP Window Function

For the VAR_SAMP window function that performs a group, cumulative, or moving computation, see "Window Aggregate Functions" on page 960.

CHAPTER 3 Arithmetic, Trigonometric, Hyperbolic Operators/Functions

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

Arithmetic Operators

Teradata SQL supports the following arithmetic operators.

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

ANSI Compliance

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

Arithmetic Operators and LOBs

Arithmetic operators do not support BLOB or CLOB types.

Arithmetic Operators and DateTime and Interval Data Types

For details on the arithmetic operators permitted for DateTime and Interval data types, see "Arithmetic Operators and ANSI DateTime and Interval Data Types" on page 838.

Arithmetic Operators and Period Data Types

For details on the arithmetic operators permitted for Period data types, see "Arithmetic Operators" on page 1033.

Arithmetic Operators and UDTs

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

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see "Implicit Type Conversions" on page 579.

Binary Arithmetic Result Data Types

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

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

Result Data Type

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

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

For details on the result data type for binary arithmetic operations involving DateTime and Interval types, see "Arithmetic Operators and Result Types" on page 838.

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
any data type	any data type	**	FLOAT
DATE	BYTEINT SMALLINT INTEGER BIGINT	+ -	DATE If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
	BYTEINT SMALLINT INTEGER	* / MOD	INTEGER These operations on DATE do not report an error, but results are generally not meaningful.

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
DATE (*continued*)	BIGINT	* / MOD	BIGINT These operations on DATE do not report an error, but results are generally not meaningful.
	DECIMAL(*k*,*j*)	+ -	DATE These operations on DATE do not report an error, but results are generally not meaningful. Fractions of decimal values are truncated when added to or subtracted from date values. If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		* / MOD	DECIMAL(*p*,*j*) These operations on DATE do not report an error, but results are generally not meaningful. For details about the value of *p*, see "DECIMAL Result Data Type" on page 119.
	NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER	+ -	DATE These operations on DATE do not report an error, but results are generally not meaningful. Fractions of decimal values are truncated when added to or subtracted from date values. If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		* / MOD	NUMBER
	FLOAT	* / + - MOD	FLOAT
	DATE	-	INTEGER The difference between two dates is the number of days between those dates. Note that this is not the numeric difference between the values.
		+ * / MOD	INTEGER These operations on DATE do not report an error, but results are generally not meaningful.
	CHAR(*n*) VARCHAR(*n*)	* / + - MOD	FLOAT These operations on DATE do not report an error, but results are generally not meaningful. If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
BYTEINT SMALLINT INTEGER	BYTEINT SMALLINT INTEGER	* / + - MOD	INTEGER
	BIGINT	* / + - MOD	BIGINT
	DECIMAL(*k*,*j*)	* / + - MOD	DECIMAL(*p*,*j*) For details about the value of *p*, see "DECIMAL Result Data Type" on page 119.
	NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER	* / + - MOD	NUMBER
	FLOAT	* / + - MOD	FLOAT
	CHAR(*n*) VARCHAR(*n*)	* / + - MOD	FLOAT If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.
	DATE	+	DATE If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		-	error
		* / MOD	INTEGER These operations on DATE do not report an error, but results are generally not meaningful.

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
BIGINT	BYTEINT SMALLINT INTEGER BIGINT	* / + - MOD	BIGINT
	DECIMAL(*k*,*j*)	* / + - MOD	DECIMAL(*p*,*j*) For details about the value of *p*, see "DECIMAL Result Data Type" on page 119.
	NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER	* / + - MOD	NUMBER
	FLOAT	* / + - MOD	FLOAT
	CHAR(*n*) VARCHAR(*n*)	* / + - MOD	FLOAT If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.
	DATE	+	DATE If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		-	error
		* / MOD	BIGINT These operations on DATE do not report an error, but results are generally not meaningful.

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
DECIMAL(*m*,*n*)	BYTEINT SMALLINT INTEGER BIGINT	+ - *	DECIMAL(*p*,*n*) For details about the value of *p*, see "DECIMAL Result Data Type" on page 119.
		/ MOD	DECIMAL(*m*,*n*)
	DECIMAL(*k*,*j*)	+ -	DECIMAL(min(*p*,(1+max(*n*,*j*)+max(*m*-*n*,*k*-*j*))),max(*n*,*j*)) For details about the value of *p*, see "DECIMAL Result Data Type" on page 119.
		*	DECIMAL(min(*p*,*m*+*k*),(*n*+*j*)) For details about the value of *p*, see "DECIMAL Result Data Type" on page 119.
		/ MOD	DECIMAL(*p*,max(*n*,*j*)) For details about the value of *p*, see "DECIMAL Result Data Type" on page 119.
	NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER	* / + - MOD	NUMBER
	FLOAT	* / + - MOD	FLOAT
	CHAR(*n*) VARCHAR(*n*)	* / + - MOD	FLOAT If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.
	DATE	+	DATE Fractions of decimal values are truncated when added to or subtracted from date values. If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		-	error
		*	DECIMAL(*p*,*n*) These operations on DATE do not report an error, but results are generally not meaningful. For details about the value of *p*, see "DECIMAL Result Data Type" on page 119.
		/ MOD	DECIMAL(*m*,*n*) These operations on DATE do not report an error, but results are generally not meaningful.

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
NUMBER(*m*,*n*) NUMBER(*m*) NUMBER(*,*n*) NUMBER	BYTEINT SMALLINT INTEGER BIGINT	* / + - MOD	NUMBER
	DECIMAL(*k*,*j*)		
	FLOAT	* / + - MOD	FLOAT
	CHAR(*n*) VARCHAR(*n*)		FLOAT If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.
	DATE	+	DATE Fractions of decimal values are truncated when added to or subtracted from date values. If the value of a date result is not in the range of values allowed for the DATE type, an error is reported. The range is any date on the Gregorian calendar from year 1 to year 9999.
		-	error
		*	NUMBER These operations on DATE do not report an error, but results are generally not meaningful.
		/ MOD	NUMBER These operations on DATE do not report an error, but results are generally not meaningful.
	NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER	* / + - MOD	NUMBER

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
FLOAT	BYTEINT SMALLINT INTEGER BIGINT DECIMAL(*k*,*j*) NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER FLOAT	* / + - MOD	FLOAT
	DATE	* / + - MOD	FLOAT These operations on DATE do not report an error, but results are generally not meaningful.
	CHAR(*n*) VARCHAR(*n*)	* / + - MOD	FLOAT If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.
CHAR(*n*) VARCHAR(*n*)	BYTEINT SMALLINT INTEGER BIGINT DECIMAL(*k*,*j*) NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER FLOAT CHAR(*n*) VARCHAR(*n*)	* / + - MOD	FLOAT If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.
	DATE	* / + - MOD	FLOAT These operations on DATE do not report an error, but results are generally not meaningful. If an argument of an arithmetic operator is a character string, the first action is to attempt to convert the character string to a floating value. If this conversion fails, an error is reported.

DECIMAL Result Data Type

The result data type for binary arithmetic operations involving DECIMAL operands is as follows:

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
DATE	DECIMAL(*k*,*j*)	* / MOD	DECIMAL(*p*,*j*) These operations on DATE do not report an error, but results are generally not meaningful.
BYTEINT SMALLINT INTEGER BIGINT		* / + - MOD	DECIMAL(*p*,*j*)
DECIMAL(*m*,*n*)	BYTEINT SMALLINT INTEGER BIGINT	+ - *	DECIMAL(*p*,*n*)
	DATE	*	DECIMAL(*p*,*n*) These operations on DATE do not report an error, but results are generally not meaningful.

In these cases the value of *p*, the number of digits in the decimal result, depends on:

- The value specified for MaxDecimal in DBSControl.

  For more information on DBSControl and MaxDecimal, see “DBS Control utility” in the *Utilities* book.
- The number of digits in the decimal operand, where the number of digits is *k* for a DECIMAL(*k*,*j*) operand on the right side of the operator or *m* for a DECIMAL(*m*,*n*) operand on the left side of the operator.

IF MaxDecimal is ...	AND the number of digits in the decimal operand is ...	THEN *p* is ...
0 or 15	<= 15	15
	> 15 and <=18	18
	> 18	38
18	<= 18	18
	> 18	38
38	any value	38

When the operand on the left is ...	And the operand on the right is ...	And the operator is ...	Then the result data type is ...
DECIMAL(*m*,*n*)	DECIMAL(*k*,*j*)	+ -	DECIMAL(min(*p*,(1+max(*n*,*j*)+max(*m*-*n*,*k*-*j*))),max(*n*,*j*))
		*	DECIMAL(min(*p*,*m*+*k*),(*n*+*j*))
		/ MOD	DECIMAL(*p*,max(*n*,*j*))

In these cases, the value of *p* in the definition of the decimal result data type depends on the value specified for MaxDecimal in DBSControl and the number of digits in the DECIMAL(*m*,*n*) and DECIMAL(*k*,*j*) operands.

IF MaxDecimal is ...	AND ...	THEN *p* is ...
0 or 15	*m* and *k* <= 15	15
	(*m* or *k* > 15) and (*m* and *k* <=18)	18
	m or *k* > 18	38
18	*m* and *k* <= 18	18
	m or *k* > 18	38
38	*m* and *k* = any value	38

Numeric Results and Rounding

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

For more information on rounding rules and how the RoundHalfwayMagUp and RoundNumberAsDec fields in DBSControl affect rounding, see “Numeric Data Types” in *SQL Data Types and Literals* and “DBS Control utility” in *Utilities*.

Error Conditions

An error is reported when any of the following events occurs:

- Division by zero is attempted.
- The numeric range is exceeded.
- The exponentiation operator is used with a negative left argument and a right argument that is not a whole number.

Integer Division and Truncation

Integer division yields whole results, truncated toward zero.

Structure of Arithmetic Expressions

Order of Evaluation

The following table lists the precedence of operations in arithmetic expressions.

Precedence	Operation
Highest	+ operand (unary plus) - operand (unary minus)
Intermediate	operand ** operand (exponentiation)
	operand * operand (multiplication) operand / operand (division) operand MOD operand (modulo operator)
	operand + operand (addition) operand - operand (subtraction)

In general, the order of evaluation is:

1. Operations enclosed in parentheses are performed first.
2. When no parentheses are present, operations are performed in order of precedence.
3. Operators of the same precedence are evaluated from left to right.

The Optimizer may reorder evaluations based on associative and commutative properties of the operations involved.

Format

The format of an arithmetic expression is the same as the default format of the result data type.

You can use the FORMAT phrase to change the default format of the result data type. The FORMAT phrase is relevant only in field mode, such as BTEQ applications, and in conversion to a character data type.

Example

You want to raise the salary for each employee in department 600 by $200 for each year spent with the company (up to a maximum of $2500 per month).

To determine who is eligible, and the new salary, enter the following statement:

```
SELECT Name, (Salary+(YrsExp*200))/12 AS Projection
FROM Employee
WHERE Deptno = 600
AND Projection < 2500 ;
```

This statement returns the following response:

```
Name        Projection
--------    ----------
Newman P       2483.33
```

The statement uses parentheses to perform the operation YrsExp * 200 first. Its result is then added to Salary and the total is divided by 12.

The parentheses enclosing YrsExp * 200 are not strictly necessary, but the parentheses enclosing Salary + (YrsExp * 200) are necessary, because, if no parentheses were used in this expression, the operation YrsExp * 200 would be divided by 12 and the result added to Salary, producing an erroneous value.

The phrase AS Projection in this example associates the arithmetic expression (Salary + (YrsExp * 200)/12) with Projection. Using the AS phrase lets you use the name Projection in the WHERE clause to refer to the entire expression.

The result is formatted without a comma separating thousands from hundreds.

Arithmetic Functions

The next sections describe the following arithmetic functions:

- ABS
- CASE_N
- CEILING
- EXP
- FLOOR
- LN
- LOG
- NULLIFZERO
- RANDOM
- RANGE_N
- SQRT
- WIDTH_BUCKET
- ZEROIFNULL

ABS

Purpose

Computes the absolute value of an argument.

Syntax

```
── ABS ─( arg )──
```

1101A480

where:

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

ANSI Compliance

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

Result Type and Attributes

Here are the default attributes for the result of ABS.

- If the operand is numeric, the format is the default format for the resulting data type.
- If the operand is character, the format is the default format for FLOAT.
- If the operand is a UDT, the format is the default format for the predefined type to which the UDT is implicitly cast.

Note: The NULL keyword has a data type of INTEGER.

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

Argument Types and Rules

If the argument is not numeric, it is converted to a numeric value, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a character string, it is converted to a numeric value of the FLOAT data type.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character

- DateTime
- Interval

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

ABS cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Examples

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

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

CASE_N

Purpose

Evaluates a list of conditions and returns the position of the first condition that evaluates to TRUE, provided that no prior condition in the list evaluates to UNKNOWN.

Syntax

where:

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

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 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	If NO CASE OR UNKNOWN is specified, CASE_N returns $n + 1$. If UNKNOWN is specified and NO CASE is not specified, CASE_N returns $n + 1$. If NO CASE and UNKNOWN are specified, CASE_N returns $n + 2$. If neither UNKNOWN nor NO CASE OR UNKNOWN is specified, CASE_N returns NULL.
every *conditional_expression* evaluates to FALSE	If NO CASE or NO CASE OR UNKNOWN is specified, CASE_N returns $n + 1$. If neither NO CASE nor NO CASE OR UNKNOWN is specified, CASE_N returns NULL

Result Type and Attributes

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

Data Type	Format	Title
INTEGER	Default format for INTEGER	<CASE_N function>

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

Using CASE_N to Define Partitioned Primary Indexes

The primary index for a table or join index controls the distribution and retrieval of the data for that table or join index across the AMPs. If the primary index is a *partitioned* primary index (PPI), the data can be assigned to user-defined partitions on the AMPs.

To define a primary index for a table or join index, you specify the PRIMARY INDEX phrase in the CREATE TABLE or CREATE JOIN INDEX data definition statement. To define a partitioned primary index, you include the PARTITION BY phrase when you define the primary index.

The PARTITION BY phrase requires one or more partitioning expressions that determine the partition assignment of a row. You can use CASE_N to construct a partitioning expression such that a row with any value or NULL for the partitioning columns is assigned to some partition.

You can also use RANGE_N to construct a partitioning expression. For more information, see "RANGE_N" on page 155.

If the PARTITION BY phrase specifies a list of partitioning expressions, the PPI is a *multilevel* PPI, where each partition for a level is subpartitioned according to the next partitioning expression in the list. Unlike the partitioning expression for a single-level PPI, which can consist of any valid SQL expression (with some exceptions), each expression in the list of partitioning expressions for a multilevel PPI must be a CASE_N or RANGE_N function.

You cannot ADD or DROP partitioning expressions that are based on a CASE_N function. To modify a partitioning expression that is based on a CASE_N function, you must use the ALTER TABLE statement with the MODIFY PRIMARY INDEX option to redefine the entire PARTITION BY clause, and the table must be empty. For more information, see "ALTER TABLE" in *SQL Data Definition Language*.

Using CASE_N with CURRENT_DATE or CURRENT_TIMESTAMP in a PPI

You can define a partitioning expression that uses CASE_N with the built-in functions CURRENT_DATE or CURRENT_TIMESTAMP. Subsequently, you can use the ALTER TABLE TO CURRENT statement to re-partition the table data using a newly resolved current date or timestamp. For more information, see "Rules and Guidelines for Optimizing the Reconciliation of CASE_N PPI Expressions Based On Updatable Current Date and Updatable Current Timestamp" in *SQL Data Definition Language Detailed Topics*.

Using CASE_N with Character Comparison

You can specify conditional expressions in the CASE_N function that compare CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data types. The following usage rules apply:

- A CASE_N partitioning expression can use character or graphic comparison except when the comparison involves KANJI1 or KANJISJIS columns or literal expressions.
- A CASE_N partitioning expression can use the UPPERCASE qualifier and the following functions: LOWER, UPPER, TRANSLATE, TRIM, VARGRAPHIC, INDEX, MINDEX, POSITION, TRANSLATE_CHK, CHAR2HEXINT.
- Any string literal referenced within a CASE_N expression must be less than 31,000 bytes.
- The order of character data used in evaluating the conditional expressions in a CASE_N function is determined by the session collation and case specificity of the expression.
  - If the expression is not part of a PPI, the current session collation is used.
  - If the expression is part of a PPI, evaluation is done using the session collation that was in effect when the table or join index was created, or when the partitioning was modified using the ALTER TABLE statement.
  - The case specificity of column references and literals is determined based on the session default, or an explicit CAST, or a specification in the CREATE TABLE

statement when the table was created. The column can be explicitly assigned to be CASESPECIFIC or NOT CASESPECIFIC, and literal expressions can be CAST with these qualifiers.

If not explicitly specified, the default of NOT CASESPECIFIC is used if Teradata session transaction semantics are in effect. If ANSI session transaction semantics are in effect, the default is CASESPECIFIC.

For example, if a conditional expression is a combination of NOT CASESPECIFIC expressions and a literal with no case specific qualifier (CASESPECIFIC, NOT CASESPECIFIC), the case specificity will be case specific in ANSI mode sessions and not case specific in Teradata mode sessions.

Note: All character string comparisons involving graphic data are case specific.

- In character comparison operations (=, <, >, <=, >=, <>, BETWEEN, LIKE), if a string literal is shorter than the column data to which it is compared, the string literal is treated as if it is padded with a pad character specific to the character set (for example, a <space> character).

  Note that the pad character might not collate to the lowest code point in the collation. For a literal of length *n*, if the column value being compared precisely matches the literal for the first *n* characters, but contains a character that collates less than the pad character at position *n*+1, then the column value will collate less than the string literal. See "Example 9" on page 134.

Restrictions

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

- Can specify a maximum of 65533 conditions (unless it is part of a larger partitioning expression)
- Must not contain the system-derived columns PARTITION or PARTITION#L1 through PARTITION#L15
- Must not use Period data types, but can use the following:
  - BEGIN bound function for which input is a Period data type column and not a Period expression.
  - END bound function for which input is a Period data type column and not a Period expression.
  - IS [NOT] UNTIL_CHANGED.
  - IS [NOT] UNTIL_CLOSED.

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

Note that partition elimination for queries is often limited to literal or using value equality conditions on the partitioning columns, and the Optimizer may not eliminate some partitions when it possibly could. Also, evaluating a complex CASE_N may be costly in terms of CPU cycles and the overhead of CASE_N may cause the table header to be excessively large.

Example 1

Here is an example that uses CASE_N and the value of the totalorders column to define the partition to which a row is assigned:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER)
 PRIMARY INDEX (storeid, productid)
  PARTITION BY CASE_N(totalorders < 100, totalorders < 1000,
                      NO CASE, UNKNOWN);
```

In the example, CASE_N specifies four partitions to which a row can be assigned, based on the value of the totalorders column.

Partition Number	Condition
1	The value of the totalorders column is less than 100.
2	The value of the totalorders column is less than 1000, but greater than or equal to 100.
3	The value of the totalorders column is greater than or equal to 1000.
4	The totalorders column is NULL.

Example 2

Here is an example that modifies "Example 1" to use CASE_N in a list of partitioning expressions that define a multilevel PPI:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER NOT NULL)
 PRIMARY INDEX (storeid, productid)
  PARTITION BY (CASE_N(totalorders < 100, totalorders < 1000,
                       NO CASE)
               ,CASE_N(orderdate <= '2005-12-31', NO CASE) );
```

The example defines six partitions to which a row can be assigned. The first CASE_N expression defines three partitions based on the value of the totalorders column. The second CASE_N expression subdivides each of the three partitions into two partitions based on the value of the orderdate column.

Level 1 Partition Number	Level 2 Partition Number	Condition
1	1	The value of the totalorders column is less than 100 and the value of the orderdate column is less than or equal to '2005-12-31'.
	2	The value of the totalorders column is less than 100 and the value of the orderdate column is greater than '2005-12-31'.
2	1	The value of the totalorders column is less than 1000 but greater than or equal to 100, and the value of the orderdate column is less than or equal to '2005-12-31'.
	2	The value of the totalorders column is less than 1000 but greater than or equal to 100, and the value of the orderdate column is greater than '2005-12-31'.
3	1	The value of the totalorders column is greater than or equal to 1000 and the value of the orderdate column is less than or equal to '2005-12-31'.
	2	The value of the totalorders column is greater than or equal to 1000 and the value of the orderdate column is greater than '2005-12-31'.

Example 3

The following example shows the count of rows in each partition if the orders table were to be partitioned using the CASE_N expression.

```
CREATE TABLE orders
 (orderkey INTEGER NOT NULL
 ,custkey INTEGER
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL)
 PRIMARY INDEX (orderkey);

INSERT INTO orders (1, 1, '1996-01-01');
INSERT INTO orders (2, 1, '1997-04-01');
```

The CASE_N expression in the following SELECT statement specifies three conditional expressions and the NO CASE condition.

```
SELECT COUNT(*),
       CASE_N(orderdate >= '1996-01-01' AND
              orderdate <= '1996-12-31' AND
              custkey <> 999999,
              orderdate >= '1997-01-01' AND
              orderdate <= '1997-12-31' AND
              custkey <> 999999,
              orderdate >= '1998-01-01' AND
              orderdate <= '1998-12-31' AND
              custkey <> 999999,
              NO CASE
       ) AS Partition_Number
FROM orders
GROUP BY Partition_Number
ORDER BY Partition_Number;
```

The results look like this:

```
      Count(*)  Partition_Number
   -----------  ----------------
             1                 1
             1                 2
```

Example 4

The following example creates a table partitioned with orders data for each quarter in 2008.

```
CREATE TABLE Orders
  (O_orderkey INTEGER NOT NULL,
   O_custkey INTEGER,
   O_orderperiod PERIOD (DATE) NOT NULL,
   O_orderpriority CHAR (21),
   O_comment VARCHAR (79))
  PRIMARY INDEX (O_orderkey)
  PARTITION BY
     CASE_N (END (O_orderperiod) <= date'2008-03-31', /* First Quarter */
             END (O_orderperiod) <= date'2008-06-30', /* Second Quarter */
             END (O_orderperiod) <= date'2008-09-30', /* Third Quarter */
             END (O_orderperiod) <= date'2008-12-31' /* Fourth Quarter */
            );
```

The following SELECT statement scans two partitions and displays the details of the orders placed for the first two quarters.

```
SELECT *
FROM Orders
WHERE END (O_orderperiod) > date'2008-06-30';
```

Example 5

The following example uses IS [NOT] UNTIL_CHANGED in the PPI partitioning expression to check whether or not the ending bound of a Period expression is UNTIL_CHANGED.

```
CREATE TABLE TESTUC
  (A  INTEGER,
   B  PERIOD (DATE),
   C INTEGER)
  PRIMARY INDEX (A)
  PARTITION BY
     CASE_N (END (b) IS UNTIL_CHANGED,
             END (b) IS NOT UNTIL_CHANGED, UNKNOWN);
```

Example 6

The following example uses IS [NOT] UNTIL_CLOSED in the PPI partitioning expression to check whether or not the ending bound of a transaction time column is UNTIL_CLOSED.

```
CREATE TABLE TESTUCL
 (A  INTEGER,
  B  PERIOD (TIMESTAMP (6) WITH TIME ZONE) NOT NULL AS TRANSACTIONTIME,
  C INTEGER)
 PRIMARY INDEX (A)
 PARTITION BY
    CASE_N (END (b) IS UNTIL_CLOSED,
```

```
END (b) IS NOT UNTIL_CLOSED, UNKNOWN);
```

Example 7

In this example, the session collation is ASCII.

```
CASE_N (a<'b', a>='ba' and a<'dogg' and b<>'cow', c<>'boy', NO CASE OR
UNKNOWN)
```

The following table shows the result value returned by the above CASE_N function given the specified values for *a*, *b*, and *c*. *x* and *y* represent any value or NULL. The value 4 is returned when all the conditions are FALSE, or a condition is UNKNOWN with all preceding conditions evaluating to FALSE.

a	b	c	Result
'a'	*x*	*y*	1
'boy'	'girl'	*y*	2
'boy'	NULL	*y*	4
'boy'	'cow'	'man'	3
'boy'	'cow'	'boy'	4
'dog'	'ball'	*y*	2
'dogg'	*x*	NULL	4
'dogg'	*x*	'man'	3
'egg'	*x*	'boy'	4
'egg'	*x*	NULL	4
'egg'	*x*	'girl'	3

Example 8

In this example, the session collation is ASCII.

```
CASE_N (a<'b', a>='ba' and a<'dogg' and b<>'cow', c<>'boy', UNKNOWN)
```

The following table shows the result value returned by the above CASE_N function given the specified values for *a*, *b*, and *c*. The *x* and *y* represent any value or NULL. The value 4 is returned if a condition is UNKNOWN with all preceding conditions evaluating to FALSE. NULL is returned if all the conditions are false.

a	b	c	Result
'a'	*x*	*y*	1
'boy'	'girl'	*y*	2
'boy'	NULL	*y*	4

a	b	c	Result
'boy'	'cow'	'man'	3
'boy'	'cow'	'boy'	NULL
'dog'	'ball'	*y*	2
'dogg'	*x*	NULL	4
'dogg'	*x*	'man'	3
'egg'	NULL	'boy'	NULL
'egg'	*x*	'boy'	NULL
'egg'	*x*	NULL	4
'egg'	*x*	'girl'	3

Example 9

In this example, the session collation is ASCII when submitting the CREATE TABLE statement, and the pad character is <space>. The example defines two partitions (numbered 1 and 2) based on the value of *a*:

- The value of *a* is between 'a ' (a followed by 9 spaces) and 'b '.
- The value of *a* is between 'b ' and 'c '.

```
CREATE SET TABLE t2
   (a VARCHAR(10) CHARACTER SET UNICODE NOT CASESPECIFIC,
    b INTEGER)
PRIMARY INDEX (a)
PARTITION BY CASE_N(a BETWEEN 'a' AND 'b', a BETWEEN 'b' AND 'c');
```

The following INSERT statement inserts a character string consisting of a single <tab> character between the 'b' and '1'.

```
INSERT t2 ('b1', 1);
```

The following INSERT statement inserts a character string consisting of a single <space> character between the 'b' and '1'.

```
INSERT t2 ('b 1', 2);
```

The following SELECT statement shows the result of the INSERT statements. Since the <tab> character has a lower code point than the <space> character, the first string inserted maps to partition 1.

```
SELECT PARTITION, a, b FROM t2 ORDER BY 1;

*** Query completed. 2 rows found. 3 columns returned.
*** Total elapsed time was 1 second.

  PARTITION  a          b
-----------  ------ -----
           1  b 1        1   (string contains single <tab> character)
           2  b 1        2   (string contains single <space> character)
```

Related Topics

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

CEILING

Purpose

Returns the smallest integer value that is not less than the input argument.

Syntax

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
arg	a numeric expression.

ANSI Compliance

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

Invocation

CEILING is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, FLOAT/REAL/DOUBLE PRECISION, DECIMAL/NUMERIC, or NUMBER

Return Value

CEILING is a scalar function that returns the following values:

IF *arg* is...	THEN CEILING returns...
a non-exact number	the next integer value that is greater than *arg*.
an exact number	the input argument *arg*.

IF *arg* is...	THEN CEILING returns...
NULL	NULL.

Result Type and Attributes

The result type is the same data type as that of the numeric input argument.

If the input argument is defined as a DECIMAL/NUMERIC with a precision less than 38, the return DECIMAL/NUMERIC value will have its precision increased by 1. For example, if DECIMAL(6,4) is passed in, it will be increased and returned as a DECIMAL(7,4). If the precision is 38, the scale will be reduced by 1 unless the scale is 0. For example, a DECIMAL(38,38) results in a return data type of DECIMAL(38,37).

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

Example 1

The following query returns the FLOAT value 16E0, since 16 is the smallest integer that is not less than the FLOAT value 15.7E0.

```
SELECT CEILING(157E-1);
```

Example 2

The following query returns a DECIMAL value of 16.0 since 16 is the smallest integer that is not less than the DECIMAL literal 15.7.

```
SELECT CEILING(15.7);
```

Example 3

The following query returns a DECIMAL value of -12.0 since -12 is the smallest integer that is not less than the DECIMAL literal -12.3.

```
SELECT CEILING(-12.3);
```

Example 4

The following query returns the value 10.00 with a data type of DECIMAL(4,2), since 10 is the smallest integer that is not less than 9.99. Note that the precision of the return value is increased by 1.

```
SELECT CEIL( CAST(9.99 AS DECIMAL(3,2)) );
```

EXP

Purpose

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

Syntax

```
── EXP ─( arg )──
```

1101A484

where:

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

ANSI Compliance

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

Result Type and Attributes

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

Data Type	Format	Title
FLOAT	Default format for the resulting data type	EXP(*arg*)

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

Argument Types and Rules

If *arg* is not FLOAT, it is converted to FLOAT, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

EXP cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Usage Notes

Executing EXP may sometimes result in a numeric overflow error.

Examples

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

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

FLOOR

Purpose

Returns the largest integer equal to or less than the input argument.

Syntax

```
                                  FLOOR ── ( arg ) ──────────────────────────────────────
┬──────────────────────┬
└─ TD_SYSFNLIB. ───────┘
                                                                                  1101A663
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
arg	a numeric expression.

ANSI Compliance

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

Invocation

FLOOR is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, FLOAT/REAL/DOUBLE PRECISION, DECIMAL/NUMERIC, or NUMBER

Return Value

FLOOR is a scalar function that returns the following values:

IF *arg* is...	THEN FLOOR returns...
a non-exact number	the next largest integer that is equal to or less than *arg*.
an exact number	the input argument *arg*.

IF *arg* is...	THEN FLOOR returns...
NULL	NULL.

Result Type and Attributes

The result type is the same data type as that of the numeric input argument.

If the input argument is defined as a DECIMAL/NUMERIC with a precision less than 38, the return DECIMAL/NUMERIC value will have its precision increased by 1. For example, if DECIMAL(6,4) were passed in, it would be increased and returned as a DECIMAL(7,4). If the precision is 38, the scale will be reduced by 1 unless the scale is 0. For example, a DECIMAL(38,38) results in a return data type of DECIMAL(38,37).

The default title for FLOOR is FLOOR(*arg*).

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

Example 1

The following query returns the FLOAT value 13E0, since 13 is the largest integer that is less than the FLOAT value 13.6E0.

```
SELECT FLOOR (136E-1);
```

Example 2

The following query returns a DECIMAL value of -7.0 since -7 is the largest integer that is less than the DECIMAL literal -6.5.

```
SELECT FLOOR(-6.5);
```

Example 3

The following query returns a value of -7.0 with a data type of DECIMAL(3,1), since -7 is the largest integer less than DECIMAL -6.5. Note that the precision of the return value is increased by 1.

```
SELECT FLOOR (CAST(-6.5 AS DECIMAL(2,1)));
```

LN

Purpose

Computes the natural logarithm of the argument.

Syntax

```
── LN ─( arg )──
```

1101A485

where:

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

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for LN(*arg*) are as follows.

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

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

Argument Types and Rules

If *arg* is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

LN cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Examples

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

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

LOG

Purpose

Computes the base 10 logarithm of an argument.

Syntax

```
── LOG ─ ( arg ) ──
```

1101A486

where:

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

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for LOG(*arg*) are as follows.

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

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

Argument Types and Rules

If *arg* is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

LOG cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Examples

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

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

NULLIFZERO

Purpose

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

Syntax

```
— NULLIFZERO – ( arg ) —
```

1101F225

where:

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

ANSI Compliance

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

The ANSI form of this function is the CASE shorthand expression NULLIF. For more information, see "NULLIF Expression" on page 577.

Result Type and Attributes

Here are the default attributes for the result of NULLIFZERO.

- If *arg* is numeric, the data type is the same types as *arg* and the format is the same format as *arg*.
- If *arg* is character, the data type is FLOAT and the default format is FLOAT.
- If *arg* is a UDT, the data type is the type of which the UDT is implicitly cast and the format is the format of the data type to which the UDF is implicitly cast.

Note: The NULL keyword has a data type of INTEGER.

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

Result Value

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

Argument Types and Rules

If *arg* is not numeric, it is converted to a numeric value, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a character string, it is converted to a numeric value of FLOAT data type.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE
  - Interval

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.
- Implicit type conversion of UDTs for system operators and functions, including NULLIFZERO, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

NULLIFZERO cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Example 1

The following expressions return an error if the value of *x* or *expression* is zero.

```
6 / x
6 / expression
```

On the other hand, the following expressions return null, which is not an error because there is no violation of the divide by zero rule.

```
6 / NULLIFZERO(x)
6 / NULLIFZERO(expression)
```

Example 2

The following request returns a null in the second column because the HCap field value for Newman is zero. In BTEQ (field mode) this appears as a '?'.

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

Related Topics

For additional expressions involving checks for nulls, see:

- "COALESCE Expression" on page 575
- "NULLIF Expression" on page 577
- "ZEROIFNULL" on page 182

POWER

Purpose

Returns *base_value* raised to the power of *exponent_value*.

Syntax

```
┬──────────────┬── POWER ── ( base_value, exponent_value ) ──────────────
└─ SYSLIB. ────┘
```

1145-054A

where:

Syntax element ...	Specifies ...
SYSLIB	the name of the database where the function is located.
base_value	a numeric argument. If *base_value* is negative, *exponent_value* must be an integer. If any input argument is NULL, the function returns NULL.
exponent_value	a numeric argument.

ANSI Compliance

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

Invocation

POWER is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER

Result Type

If either of the input arguments is a FLOAT type, the result data type is FLOAT. Otherwise, the result data type is NUMBER.

For information on the default data type format for NUMBER or FLOAT, see *SQL Data Types and Literals*.

Example 1

The following query returns the result 8.0.

```
SELECT POWER(2, 3);
```

Example 2

The following query returns the result 0.125.

```
SELECT POWER(2, -3);
```

Example 3

The following query returns the result 10.648.

```
SELECT POWER(2.2, 3);
```

Example 4

The following query returns an error because the base value is negative and the exponent value is not an integer.

```
SELECT POWER(-2.2, 3.1);
```

RANDOM

Purpose

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

Syntax

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

1101C025

where:

Syntax element ...	Specifies ...
lower_bound	an integer literal to define the lower bound on the closed interval over which a random number is to be selected. The limits for *lower_bound* range from -2147483648 to 2147483647, inclusive. *lower_bound* must be less than or equal to *upper_bound*.
upper_bound	an integer literal to define the upper bound on the closed interval over which a random number is to be selected. The limits for *upper_bound* range from -2147483648 to 2147483647, inclusive. *upper_bound* must be greater than or equal to *lower_bound*.

ANSI Compliance

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

Result Type and Attributes

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

Data Type	Format	Title
INTEGER	Default format for INTEGER	Random(x,y)

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

Computation

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

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

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

where:

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

Multiple RANDOM Calls Within a SELECT List

You can call RANDOM any number of times in the SELECT list, for example:

```
SELECT RANDOM(1,100), RANDOM(1,100);
```

Each call defines a new random value.

Restrictions

The following rules and restrictions apply to the use of the RANDOM function.

- RANDOM can only be called in one of the following SELECT query clauses:
  - WHERE
  - GROUP BY
  - ORDER BY
  - HAVING/QUALIFY
- RANDOM cannot be referenced by position in a GROUP BY or ORDER BY clause.
- RANDOM cannot be nested inside aggregate or ordered analytical functions.
- RANDOM cannot be used in the expression list of an INSERT statement to create a primary index or partitioning column value.

  For example:

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

  RANDOM causes an error to be reported in this case if the first column in the table is a primary index or partitioning column.

Using RANDOM as a Condition on an Index

Because the RANDOM function is evaluated for each selected row, a condition on an index column that includes the RANDOM function results in an all-AMP operation.

For example, consider the following table definition:

```
CREATE TABLE t1
   (c1 INTEGER
   ,c2 VARCHAR(9))
PRIMARY INDEX ( c1 );
```

The following SELECT statement results in an all-AMP operation:

```
SELECT *
FROM t1
WHERE c1 = RANDOM(1,12);
```

Example

Suppose you have a table named sales_table with the following subset of columns.

Store_ID	Product_ID	Sales
1003	C	20000
1002	C	35000
1001	C	60000
1002	D	50000
1003	D	50000
1001	D	35000
1001	A	100000
1002	A	40000
1001	E	30000

The following SELECT statement returns a random number between 1 and 3, inclusive, for each row in the results table.

```
SELECT store_id, product_id, sales, RANDOM(1,3)
FROM sales_table;
```

The results table might look like this.

Store_ID	Product_ID	Sales	RANDOM(1,3)
1003	C	20000	1
1002	C	35000	2
1001	C	60000	2

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

RANGE_N

Purpose

Evaluates an expression and maps the result into one of a list of specified ranges and returns the position of the range in the list.

Syntax

where:

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

Syntax element ...	Specifies ...
start_expression *	a literal or literal expression that defines the starting boundary of a range. The data type of *start_expression* must be the same as the data type of *test_expression*, or must be such that it can be implicitly cast to the same data type as *test_expression*. For a starting boundary without an ending boundary, the range is defined by its starting boundary (and the starting boundary is included in the range), up to but not including the starting boundary of the next range. Use an asterisk (*) for the starting boundary of the first range in the list to indicate the lowest possible value (all values and NULL are greater than a starting boundary specified as an asterisk). An asterisk is compatible with any data type.
end_expression *	a literal or literal expression that defines the ending boundary of a range. The data type of *end_expression* must be the same as the data type of *test_expression*, or must be 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 (and the starting boundary is included in the range), up to but not including the starting boundary of the next range. Use an asterisk (*) for the ending boundary of the last range in the list to indicate the highest possible value (all values and NULL are less than an ending boundary specified as an asterisk).
EACH *range_size*	a literal or literal expression with a value greater than zero. A range that specifies an EACH phrase is equivalent to a series of ranges, where the first range in the series starts at *start_expression*, and subsequent ranges start at *start_expression* + (*range_size* * *n*), where *n* starts at one and increments by one while *start_expression* + (*range_size* * *n*) is less than or equal to *end_expression*, or less than the next *start_expression* in the list of ranges. For DATE types, the calculation of valid dates in subsequent ranges uses ADD_MONTHS instead of the + arithmetic operator. For more information on ADD_MONTHS, see "ADD_MONTHS" on page 850. The data type of *range_size* must be compatible for adding to *test_expression*. Note: If the data type of *test_expression* is a character type (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC), you cannot specify the EACH phrase.
NO RANGE	an optional range to handle a *test_expression* that does not map into any of the specified ranges.
OR UNKNOWN	an option to use with NO RANGE. The NO RANGE OR UNKNOWN option handles a *test_expression* that does not map into any of the specified ranges, or a *test_expression* that evaluates to NULL when RANGE_N does not specify the range BETWEEN * AND *.
UNKNOWN	an option to handle a *test_expression* that evaluates to NULL when RANGE_N does not specify the range BETWEEN * AND *.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 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.
the result of *test_expression* is NULL	If RANGE_N does not specify one of the following: • BETWEEN * AND * • UNKNOWN • NO RANGE OR UNKNOWN RANGE_N returns NULL. If RANGE_N specifies the range BETWEEN * AND *, RANGE_N returns 1, regardless of whether NO RANGE, NO RANGE OR UNKNOWN, or UNKNOWN is specified. If RANGE_N does not specify the range BETWEEN * AND * and • If NO RANGE OR UNKNOWN is specified, RANGE_N returns $n + 1$. • If UNKNOWN is specified and NO RANGE is not specified, RANGE_N returns $n + 1$. • NO RANGE and UNKNOWN are specified, RANGE_N returns $n + 2$.
test_expression is outside all the ranges in the list	If NO RANGE or NO RANGE OR UNKNOWN is specified, RANGE_N returns $n + 1$. If neither NO RANGE nor NO RANGE OR UNKNOWN is specified, RANGE_N returns NULL.

Result Type and Attributes

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

Data Type	Format	Title
INTEGER	Default format of the INTEGER data type	<RANGE_N function>

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

Using RANGE_N to Define Partitioned Primary Indexes

The primary index for a table or join index controls the distribution of the data for that table or join index across the AMPs, as well as its retrieval. If the primary index is a *partitioned* primary index (PPI), the data can be assigned to user-defined partitions on the AMPs.

To define a primary index for a table or join index, you specify the PRIMARY INDEX phrase in the CREATE TABLE or CREATE JOIN INDEX data definition statement. To define a partitioned primary index, you include the PARTITION BY phrase when you define the primary index.

The PARTITION BY phrase requires one or more partitioning expressions that determine the partition assignment of a row. You can use RANGE_N to construct a partitioning expression such that a row with any value or NULL for the partitioning columns is assigned to some partition.

You can also use CASE_N to construct a partitioning expression. For more information, see "CASE_N" on page 126.

If the PARTITION BY phrase specifies a list of partitioning expressions, the PPI is a *multilevel* PPI, where each partition for a level is subpartitioned according to the next partitioning expression in the list. Unlike the partitioning expression for a single-level PPI, which can consist of any valid SQL expression (with some exceptions), each expression in the list of partitioning expressions for a multilevel PPI must be a CASE_N or RANGE_N function.

Using RANGE_N with CURRENT_DATE or CURRENT_TIMESTAMP in a PPI

You can define a partitioning expression that uses RANGE_N with the built-in functions CURRENT_DATE or CURRENT_TIMESTAMP. Use of CURRENT_DATE or CURRENT_TIMESTAMP in a partitioning expression is most appropriate when the data must be partitioned as one or more current partitions and one or more history partitions where the current and history partitions are based on the resolved CURRENT_DATE or CURRENT_TIMESTAMP in the partitioning expression. This allows you to periodically reconcile the table to move older data from the current partition into one or more history partitions using the ALTER TABLE TO CURRENT statement instead of redefining the partitioning using explicit dates which must be determined each time the ALTER TABLE DROP/ADD RANGE is done.

For more information, see "Rules and Guidelines for Optimizing the Reconciliation of RANGE_N PPI Expressions Based On Updatable Current Date and Updatable Current Timestamp" in *SQL Data Definition Language Detailed Topics*.

Using RANGE_N with Character Data

You can specify character expressions (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) as the *test_expression* and/or the range boundaries in a RANGE_N function. The following usage rules apply:

- A RANGE_N partitioning expression can use the UPPERCASE qualifier and the following functions: LOWER, UPPER, TRANSLATE, TRIM, VARGRAPHIC, INDEX, MINDEX, POSITION, TRANSLATE_CHK, CHAR2HEXINT.
- If *test_expression* is a character data type, you cannot specify the EACH phrase.
- Any string literal referenced within a RANGE_N expression must be less than 31,000 bytes.
- If *test_expression* is a character data type, and the length of any of the range boundaries (minus trailing pad characters) is greater than the length of *test_expression*, an error is returned.
- For character RANGE_N partitioning, the increasing order of ranges is determined by the session collation and case specificity of the *test_expression*. If the *test_expression* is a combination of NOT CASESPECIFIC expressions and a literal with no case specific qualifier (CASESPECIFIC, NOT CASESPECIFIC), the case specificity will be case specific in ANSI mode sessions and not case specific in Teradata mode sessions.

  Note: All character string comparisons involving graphic data are case specific.
- An error is returned if any of the specified ranges are defined with null boundaries, are not increasing, or overlap. For character test values, increasing order is determined by the session collation and case specificity of the *test_expression*.
- In character comparison operations (=, <, >, <=, >=, <>, BETWEEN, LIKE), if a string literal is shorter than the column data to which it is compared, the string literal is treated as if it is padded with a pad character specific to the character set (for example, a <space> character). Therefore, if a character *test_expression* is defined with a longer length than a character range boundary, comparison of the *test _expression* to that range boundary will behave as if the range boundary is padded with pad characters.

  Note that the pad character might not collate to the lowest code point in the collation. For a range boundary of length *n*, if the *test_expression* precisely matches that range boundary for the first *n* characters, but contains a character that collates less than the pad character at position *n*+1, then the *test_expression* will collate less than the range boundary. See "Example 10" on page 167.

Restrictions

If RANGE_N appears in a PARTITION BY phrase, it:

- Can specify a maximum of 65,533 ranges (unless it is part of a larger partitioning expression)
- Must not contain the system-derived columns PARTITION or PARTITION#L1 through PARTITION#L15
- Must not use Period data types, but can use the BEGIN or END bound functions on a Period data type column when they result in a DATE data type.

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

If RANGE_N specifies CURRENT_DATE or CURRENT_TIMESTAMP in a partitioning expression, you cannot use ALTER TABLE to add or drop ranges for the table. You must use the ALTER TABLE TO CURRENT statement to achieve this function.

Using a UDT as the Test Expression

The *test_expression* should not be an expression that results in a UDT data type. An error is reported if this occurs when RANGE_N is used to define a PPI. If RANGE_N is not used to define a PPI, you should explicitly cast the expression so that it is BYTEINT, SMALLINT, INTEGER, DATE, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC instead of depending upon any implicit conversions.

Example 1

Here is an example that uses RANGE_N and the value of the totalorders column to define the partition to which a row is assigned:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER)
 PRIMARY INDEX (storeid, productid)
  PARTITION BY RANGE_N(totalorders BETWEEN *, 100, 1000 AND *,
                       UNKNOWN);
```

In the example, RANGE_N specifies four partitions to which a row can be assigned, based on the value of the totalorders column:

Partition Number	Condition
1	The value of the totalorders column is less than 100.
2	The value of the totalorders column is less than 1000, but greater than or equal to 100.
3	The value of the totalorders column is greater than or equal to 1000.
4	The totalorders column is NULL, so the range is UNKNOWN.

Example 2

Here is an example that modifies "Example 1" to use RANGE_N in a list of partitioning expressions that define a multilevel PPI:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER NOT NULL)
 PRIMARY INDEX (storeid, productid)
```

```
PARTITION BY (RANGE_N(totalorders BETWEEN *, 100, 1000 AND *)
             ,RANGE_N(orderdate BETWEEN *, '2005-12-31' AND *) );
```

The example defines six partitions to which a row can be assigned. The first RANGE_N expression defines three partitions based on the value of the totalorders column. The second RANGE_N expression subdivides each of the three partitions into two partitions based on the value of the orderdate column.

Level 1 Partition Number	Level 2 Partition Number	Condition
1	1	The value of the totalorders column is less than 100 and the value of the orderdate column is less than '2005-12-31'.
	2	The value of the totalorders column is less than 100 and the value of the orderdate column is greater than or equal to '2005-12-31'.
2	1	The value of the totalorders column is less than 1000 but greater than or equal to 100, and the value of the orderdate column is less than '2005-12-31'.
	2	The value of the totalorders column is less than 1000 but greater than or equal to 100, and the value of the orderdate column is greater than or equal to '2005-12-31'.
3	1	The value of the totalorders column is greater than or equal to 1000 and the value of the orderdate column is less than '2005-12-31'.
	2	The value of the totalorders column is greater than or equal to 1000 and the value of the orderdate column is greater than or equal to '2005-12-31'.

Example 3

Here is an example that defines a partitioned primary index that specifies one partition to which rows are assigned, for any value of the totalorders column, including NULL:

```
CREATE TABLE orders
 (storeid INTEGER NOT NULL
 ,productid INTEGER NOT NULL
 ,orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL
 ,totalorders INTEGER)
 PRIMARY INDEX (storeid, productid)
  PARTITION BY RANGE_N(totalorders BETWEEN * AND *);
```

Example 4

The following example shows the count of rows in each partition if the table were to be partitioned using the RANGE_N expression.

```
CREATE TABLE orders
 (orderkey INTEGER NOT NULL
 ,custkey INTEGER
 ,orderdate DATE FORMAT 'yyyy-mm-dd')
 PRIMARY INDEX (orderkey);
```

```
INSERT INTO orders (1,  100, '1998-01-01');
INSERT INTO orders (2,  100, '1998-04-01');
INSERT INTO orders (3,  109, '1998-04-01');
INSERT INTO orders (4,  101, '1998-04-10');
INSERT INTO orders (5,  100, '1998-07-01');
INSERT INTO orders (6,  109, '1998-07-10');
INSERT INTO orders (7,  101, '1998-08-01');
INSERT INTO orders (8,  101, '1998-12-01');
INSERT INTO orders (9,  111, '1999-01-01');
INSERT INTO orders (10, 111, NULL);
```

The RANGE_N expression in the following SELECT statement uses the EACH phrase to define a series of 12 ranges, where the first range starts at '1998-01-01' and the ranges that follow have starting boundaries that increment sequentially by one month intervals.

```
SELECT COUNT(*),
       RANGE_N(orderdate
               BETWEEN DATE '1998-01-01' AND DATE '1998-12-31'
               EACH INTERVAL '1' MONTH
       ) AS Partition_Number
FROM orders
GROUP BY Partition_Number
ORDER BY Partition_Number;
```

The results look like this:

```
   Count(*) Partition_Number
----------- ----------------
          2                ?
          1                1
          3                4
          2                7
          1                8
          1               12
```

Example 5

The following example creates a table with partitioning defined using a RANGE_N expression involving the END bound function. The table creates 10 partitions where each partition represents the sales history for one year.

```
CREATE TABLE SalesHistory
  (product_code CHAR (8),
   quantity_sold INTEGER,
   transaction_period PERIOD (DATE))
  PRIMARY INDEX (product_code)
  PARTITION BY
     RANGE_N (END (transaction_period) BETWEEN date'2006-01-01'
              AND date '2015-12-31' EACH INTERVAL'1' YEAR);
```

The following SELECT statement scans five partitions of the sales history before the year 2010.

```
SELECT *
FROM SalesHistory
WHERE transaction_period < period (date'2010-01-01');
```

Example 6: Start_expression with CURRENT_DATE

If CURRENT_DATE or CURRENT_TIMESTAMP is specified in the *start_expression* of the first range in RANGE_N, and if this *start_expression* when resolved with a new CURRENT_DATE or CURRENT_TIMESTAMP falls on a partition boundary, then all partitions prior to the partition matched are dropped. Otherwise, the entire table is re-partitioned with the new partitioning expression.

Consider the following CREATE TABLE statement submitted on April 1, 2006:

```
CREATE TABLE ppi (i INT, j DATE)
PRIMARY INDEX (i)
PARTITION BY
   RANGE_N (j BETWEEN CURRENT_DATE AND
            CURRENT_DATE + INTERVAL '1' YEAR - INTERVAL '1' DAY
            EACH INTERVAL '1' MONTH);
```

The last resolved date is April 1, 2006. If you submit an ALTER TABLE TO CURRENT statement on June 1, 2006, the *start_expression*, newly resolved to CURRENT_DATE ('2006-06-01'), falls on a partition boundary of the third partition. Therefore, partitions 1 and 2 are dropped, and the last reconciled date is set to the newly resolved CURRENT_DATE.

However, if you submitted the ALTER TABLE TO CURRENT statement on June 10, 2006 instead of June 1, 2006, the *start_expression*, newly resolved to CURRENT_DATE ('2006-06-10'), does not fall on a partition boundary. Therefore, all rows are scanned and the rows are repartitioned based on the new partitioning expression. The partition boundary after this statement aligns with the 10th day of the month instead of the earlier 1st day of the month.

Example 7

The following table definition is created in the year 2007 (the current year at the time). The table is partitioned to record 5 years of order history plus orders for the current year and one future year.

```
CREATE TABLE Orders
  (o_orderkey INTEGER NOT NULL,
   o_custkey INTEGER,
   o_orderstatus CHAR(1) CASESPECIFIC,
   o_totalprice DECIMAL(13,2) NOT NULL,
   o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
   o_orderpriority CHAR(21),
   o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(
   o_orderdate BETWEEN DATE '2002-01-01' AND DATE '2008-12-31'
   EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);
```

If, in 2008, you want to alter the table such that it continues to maintain 5 years of history plus the current year and one future year, you can submit the following statement in 2008:

```
ALTER TABLE Orders MODIFY PRIMARY INDEX (o_orderkey)
   DROP RANGE WHERE PARTITION BETWEEN 1 AND 12
   ADD RANGE BETWEEN DATE '2009-01-01' AND DATE '2009-12-31'
      EACH INTERVAL '1' MONTH
WITH DELETE;
```

In this case, you must compute the new dates and specify them explicitly in the ADD RANGE clause. This requires manual intervention every year the statement is submitted.

Alternatively, you can define the table using CURRENT_DATE as follows. This makes it easier to alter the partitioning.

```
CREATE TABLE Orders
  (o_orderkey INTEGER NOT NULL,
   o_custkey INTEGER,
   o_orderstatus CHAR(1) CASESPECIFIC,
   o_totalprice DECIMAL(13,2) NOT NULL,
   o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
   o_orderpriority CHAR(21),
   o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
  PARTITION BY RANGE_N(o_orderdate BETWEEN
   CAST(((EXTRACT(YEAR FROM CURRENT_DATE)-5-1900)*10000+0101) AS DATE)
      AND
   CAST(((EXTRACT(YEAR FROM CURRENT_DATE)+1-1900)*10000+1231) AS DATE)
   EACH INTERVAL '1' MONTH)
 UNIQUE INDEX (o_orderkey);
```

You can schedule the following ALTER TABLE statement to occur yearly. This statement rolls the partition window forward by efficiently dropping and adding partitions.

```
ALTER TABLE Orders TO CURRENT WITH DELETE;
```

With the use of CURRENT_DATE, you do not need to modify the ALTER TABLE statement each time you want to repartition the data based on the new dates.

In both cases, the partitioning starts on a year boundary. In the first example, the ALTER TABLE statement does not change this, so partitioning continues to start on a year boundary. However, you can specify an ALTER TABLE statement that changes the partitioning to start on a different boundary. For example, you can roll forward to start on a particular month in a year by specifying the desired dates in the ALTER TABLE statement.

In the second example, which uses CURRENT_DATE, you can only roll forward to start on a year boundary. However, you can modify the example as follows so that partitioning can be used to roll forward to start at the beginning of a month. This case assumes that, as of the CREATE TABLE date, the Orders table will contain the last 71 months of history plus the current month and 12 months in the future (a total of 84 months).

```
CREATE TABLE Orders
  (o_orderkey INTEGER NOT NULL,
   o_custkey INTEGER,
   o_orderstatus CHAR(1) CASESPECIFIC,
   o_totalprice DECIMAL(13,2) NOT NULL,
   o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
   o_orderpriority CHAR(21),
   o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN
   CAST(((EXTRACT(YEAR FROM CURRENT_DATE)-1900)*10000 +
          EXTRACT(MONTH FROM CURRENT_DATE)*100 + 01) AS DATE) -
       INTERVAL '71' MONTH
   AND
   CAST(((EXTRACT(YEAR FROM CURRENT_DATE)+1-1900)*10000 +
```

```
            EXTRACT(MONTH FROM CURRENT_DATE)*100 + 01) AS DATE)+
         INTERVAL '13' MONTH - INTERVAL '1' DAY
      EACH INTERVAL '1' MONTH)
   UNIQUE INDEX (o_orderkey);
```

You can schedule the following ALTER TABLE statement to occur monthly or less frequently (but before running out of future months). This statement rolls the partition window forward by dropping and adding partitions so that the Orders table continues to contain the last 71 months of history plus the current month and 12 months in the future.

```
ALTER TABLE Orders TO CURRENT WITH DELETE;
```

You can define the following simpler partitioning, but it might not be optimized, and the entire table might be scanned to reconcile rows when you submit an ALTER TABLE TO CURRENT statement. This case assumes that, as of the CREATE TABLE date, the Orders table will contain about 2,191 days of history plus the current day and about 365 days in the future (a total of about 7 years).

```
CREATE TABLE Orders
  (o_orderkey INTEGER NOT NULL,
   o_custkey INTEGER,
   o_orderstatus CHAR(1) CASESPECIFIC,
   o_totalprice DECIMAL(13,2) NOT NULL,
   o_orderdate DATE FORMAT 'yyyy-mm-dd' NOT NULL,
   o_orderpriority CHAR(21),
   o_comment VARCHAR(79))
PRIMARY INDEX (o_orderkey)
PARTITION BY RANGE_N(o_orderdate BETWEEN
   CURRENT_DATE - INTERVAL '6' YEAR
      AND
   CURRENT_DATE + INTERVAL '1' YEAR
   EACH INTERVAL '1' MONTH)
UNIQUE INDEX (o_orderkey);
```

You can schedule the following ALTER TABLE statement to occur daily or less frequently (but before running out of future days). This statement rolls the partition window forward by dropping and adding partitions *only* if the CURRENT_DATE is the same day of the month as the day when the last CREATE TABLE or ALTER TABLE TO CURRENT statement was submitted. Otherwise, the entire table is scanned to reconcile the rows.

```
ALTER TABLE Orders TO CURRENT WITH DELETE;
```

This can be very inefficient if the ALTER TABLE statement is not submitted on the same day of the month as the day when the last CREATE TABLE or ALTER TABLE TO CURRENT statement was submitted. Performance degrades as the number of days between the last resolved date and the new resolved date increases due to the increasing number of rows that must be moved.

For example, if the last resolved date was January 1, 2008, and the next ALTER TABLE TO CURRENT statement is submitted on February 2, 2008, all the rows of the table will be moved to new partitions.

Example 8

The following example defines 5 ranges. The session collation is ASCII.

```
RANGE_N(animal BETWEEN *, 'ape', 'bird', 'bull' AND 'cow',
        'dog' AND *, NO RANGE, UNKNOWN)
```

where:

Range	Includes...
1	all values less than 'ape'.
2	strings greater than or equal to 'ape' and less than 'bird'.
3	strings greater than or equal to 'bird' and less than 'bull'.
4	strings between 'bull' and 'cow'.
5	strings greater than or equal to 'dog'.

If the value of *animal* matches one of the defined ranges, RANGE_N returns the number of the matched range.

If the value of *animal* is greater than 'cow' but less than 'dog', it does not match any of the ranges, so RANGE_N returns 6 because NO RANGE is specified.

If the value of *animal* is NULL, RANGE_N returns 7 because UNKNOWN is specified.

Example 9

The following example defines 5 ranges. The session collation is ASCII.

```
RANGE_N(animal BETWEEN *, 'ape', 'bird', 'bull' AND 'cow',
        'dog' AND *, UNKNOWN)
```

where:

Range	Includes...
1	all values less than 'ape'.
2	strings greater than or equal to 'ape' and less than 'bird'.
3	strings greater than or equal to 'bird' and less than 'bull'.
4	strings between 'bull' and 'cow'.
5	strings greater than or equal to 'dog'.

If the value of *animal* matches one of the defined ranges, RANGE_N returns the number of the matched range.

If the value of *animal* is greater than 'cow' but less than 'dog', it does not match any of the ranges, so RANGE_N returns NULL because NO RANGE is not specified.

If the value of *animal* is NULL, RANGE_N returns 6 because UNKNOWN is specified.

Example 10

In this example, the session collation is ASCII when submitting the CREATE TABLE statement, and the pad character is <space>. The example defines two ranges (numbered 1 and 2):

- Any values greater than or equal to 'a ' (a followed by 9 spaces) or less than 'b ' are mapped to partition 1.
- Any values greater than or equal to 'b ' or less than 'c ' are mapped to partition 2.

```
CREATE SET TABLE t2
   (a VARCHAR(10) CHARACTER SET UNICODE NOT CASESPECIFIC,
    b INTEGER)
PRIMARY INDEX (a)
PARTITION BY RANGE_N(a BETWEEN 'a','b' AND 'c');
```

The following INSERT statement inserts a character string consisting of a single <tab> character between the 'b' and '1'.

```
INSERT t2 ('b1', 1);
```

The following INSERT statement inserts a character string consisting of a single <space> character between the 'b' and '1'.

```
INSERT t2 ('b 1', 2);
```

The following SELECT statement shows the result of the INSERT statements. Since the <tab> character has a lower code point than the <space> character, the first string inserted maps to partition 1.

```
SELECT PARTITION, a, b FROM t2 ORDER BY 1;

*** Query completed. 2 rows found. 3 columns returned.
*** Total elapsed time was 1 second.

PARTITION  a          b
-----------  ------ -----
          1  b 1        1   (string contains single <tab> character)
          2  b 1        2   (string contains single <space> character)
```

Related Topics

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

For information on ...	See ...
the reconciliation of the partitioning based on newly resolved CURRENT_DATE and CURRENT_TIMESTAMP values	ALTER TABLE TO CURRENT in *SQL Data Definition Language*

ROUND

Purpose

Returns *numeric_value* rounded *places_value* places to the right or left of the decimal point.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── ROUND ── ( numeric_value ┌──────────────────┐ ) ──────────
                                                   │                  │
                                                   └─ , places_value ─┘
                                                                          1101A878
```

where:

Syntax element …	Specifies …
TD_SYSFNLIB	the name of the database where the function is located.
numeric_value	a numeric argument.
places_value	the number of places to round. If not specified, *numeric_value* is rounded to 0 places by default.

ANSI Compliance

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

Invocation

ROUND is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *places_value* = NUMBER

For the *places_value* argument, you can also pass values with data types that can be converted to INTEGER using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *numeric_value* argument.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result data type is the same type as that of the *numeric_value* argument.

If the data type of *numeric_value* is DECIMAL/NUMERIC with a precision less than 38, the return DECIMAL/NUMERIC value will have its precision increased by 1. For example, a DECIMAL(6,4) argument is returned as a DECIMAL(7,4). If the precision is 38, the scale will be reduced by 1 unless the scale is 0. For example, a DECIMAL(38,38) argument is returned as DECIMAL(38,37).

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

Usage Notes

ROUND functions as follows:

- It rounds *places_value* places to the right of the decimal point if *places_value* is positive.
- It rounds *places_value* places to the left of the decimal point if *places_value* is negative.
- It rounds to 0 places if *places_value* is zero or is omitted.
- If *numeric_value* or *places_value* is NULL, the function returns NULL.
- ROUND rounds the value away from zero and it only rounds when the next digit is a value of 5 or greater.

Note: The rounding behavior always follows the above formula regardless of the setting of the DBS Control Record RoundHalfWayMagUp field. That is, the rounding behavior always functions as if the RoundHalfWayMagUp field is set to TRUE.

Example 1

The following query returns the result 32.4000.

```
SELECT ROUND(32.4467, 1);
```

Example 2

The following query returns the result 32.4600.

```
SELECT ROUND(32.4567, 2);
```

Example 3

The following query returns the result 100.0000.

```
SELECT ROUND(99.9999, 3);
```

Example 4

The following query returns the result 30.0000.

```
SELECT ROUND(32.4567, -1);
```

Example 5

The following query returns the result 100.0000.

```
SELECT ROUND(55.4567, -2);
```

Example 6

The following query returns the result 0.0000.

```
SELECT ROUND(55.4567, -3);
```

Example 7

The following query returns the result -5.00.

```
SELECT ROUND(-5.35, 0);
```

Example 8

The following query returns the result -6.00.

```
SELECT ROUND(-5.55, 0);
```

SIGN

Purpose

Returns the sign of *numeric_value*.

Syntax

```
┬──────────────────┬── SIGN ─── ( numeric_value ) ──────────────────────────
└── TD_SYSFNLIB. ──┘
```

1101A880

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
numeric_value	a numeric argument.

ANSI Compliance

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

Invocation

SIGN is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER

Result Type

The result data type is NUMBER.

For information on the default data type format for NUMBER, see *SQL Data Types and Literals*.

Usage Notes

For all numeric types except FLOAT/REAL/DOUBLE PRECISION, SIGN will return the following:

- If *numeric_value* is < 0, -1 is returned.
- If *numeric_value* is = 0, 0 is returned.
- If *numeric_value* is > 0, 1 is returned.

For FLOAT/REAL/DOUBLE PRECISION, SIGN will return the following:

- If *numeric_value* is < 0, -1 is returned.
- If *numeric_value* is >= 0, 1 is returned.

If the input argument is NULL, the function returns NULL.

Example 1

The following query returns the result -1.

```
SELECT SIGN(-2);
```

Example 2

The following query returns the result 0 since the value is an integer and equal to 0.

```
SELECT SIGN(CAST(0 AS INTEGER));
```

Example 3

The following query returns the result 1 since the data type is FLOAT and the value is >= 0.

```
SELECT SIGN(CAST(0 as FLOAT));
```

Example 4

The following query returns the result 1.

```
SELECT SIGN(3.74);
```

SQRT

Purpose

Computes the square root of an argument.

Syntax

```
── SQRT ─ ( arg ) ──
```

1101A487

where:

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

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for SQRT(*arg*) are as follows.

Data Type	Format	Title
FLOAT	Default format for FLOAT	SQRT(*arg*)

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

Argument Types and Rules

If *arg* is not FLOAT, it is converted to FLOAT based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

SQRT cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Examples

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

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

TRUNC

Purpose

Returns *numeric_value* truncated *places_value* places to the right or left of the decimal point.

Syntax

```
┬──────────────────┬─ TRUNC ─── ( numeric_value ┬──────────────┬ ) ──────────
└─ TD_SYSFNLIB. ───┘                           └─ , places_value ─┘
                                                                  1101A879
```

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
numeric_value	a numeric argument.
places_value	the number of places to truncate. If not specified, *numeric_value* is truncated to 0 places by default.

ANSI Compliance

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

Invocation

TRUNC is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *places_value* = NUMBER

For the *places_value* argument, you can also pass values with data types that can be converted to INTEGER using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *numeric_value* argument.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The result data type is the same type as that of the *numeric_value* argument. For example, if the data type of *numeric_value* is DECIMAL/NUMERIC, the return type is DECIMAL/NUMERIC with the same precision and scale as the *numeric_value* argument.

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

Usage Notes

TRUNC functions as follows:

- It truncates *places_value* places to the right of the decimal point if *places_value* is positive.
- It truncates (makes 0) *places_value* places to the left of the decimal point if *places_value* is negative.
- It truncates to 0 places if *places_value* is zero or is omitted.
- If *numeric_value* or *places_value* is NULL, the function returns NULL.

Examples

The following query returns the result 32.4500.

```
SELECT TRUNC(32.4567, 2);
```

The following query returns the result 30.0000.

```
SELECT TRUNC(32.4567, -1);
```

WIDTH_BUCKET

Purpose

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

Syntax

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

1101A492

where:

Syntax element ...	Specifies the ...
value_expression	value for which a partition number is to be returned.
lower_bound	lower boundary for the range of values to be partitioned equally.
upper_bound	upper boundary for the range of values to be partitioned equally.
partition_count	number of partitions to be created. This value also specifies the width of the partitions by default. The number of partitions created is *partition_count* + 2. Partition 0 and partition *partition_count* + 1 account for values of *value_expression* that are outside the lower and upper boundaries.

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for WIDTH_BUCKET(*x*, *l*, *u*, *y*) are as follows.

Data Type	Format	Title
INTEGER	the default format for INTEGER	Width_bucket(x, l, u, y)

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

Argument Types and Rules

Use the following table for rules concerning WIDTH_BUCKET arguments.

Data Type	Rules
Numeric	WIDTH_BUCKET accepts all numeric data types as arguments. The arguments *value_expression*, *lower_bound*, and *upper_bound* are converted to REAL before processing. The *partition_count* argument is converted to INTEGER before processing.
Character	WIDTH_BUCKET accepts character strings that represent numeric values, and converts them to the appropriate numeric type.
• TIME, TIMESTAMP, or Period • INTERVAL • BYTE or VARBYTE • BLOB or CLOB • CHARACTER or VARCHAR if the server character set is GRAPHIC	WIDTH_BUCKET does not accept these types of arguments.

UDT	• The UDT must have an implicit cast to any of the following predefined types:
	  • Numeric
	  • Character
	  • DATE
	  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.
	• Implicit type conversion of UDTs for system operators and functions, including WIDTH_BUCKET, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

If an argument cannot be implicitly converted to an acceptable type, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

Rules

The following rules apply to WIDTH_BUCKET:

- If any argument is null, then the result is also null.
- If *partition_count* <=0 or if *partition_count* > 2147483646, an error is returned to the requestor.
- If *lower_bound* = *upper_bound*, an error is returned to the requestor.

- If *lower_bound* < *upper_bound*, then the rules in the following table apply.

IF ...	THEN the result is ...
value_expression < *lower_bound*	0.
value_expression >= *upper_bound*	*partition_count* +1. If the result cannot be represented by the data type specified for the result, then an error is returned.
anything else	the greatest exact numeric value with scale 0 that is less than or equal to the following expression. $\left(\frac{(\text{partition_count})(\text{value_expression} - \text{lower_bound})}{(\text{upper_bound} - \text{lower_bound})}\right) + 1$

- If *lower_bound* > *upper_bound*, then the rules in the following table apply.

IF ...	THEN the result is ...
value_expression > *lower_bound*	0.
value_expression <= *upper_bound*	*partition_count* +1. If the result cannot be represented by the data type specified for the result, then an error is returned.
anything else	the least exact numeric value with scale 0 that is less than or equal to the following expression. $\left(\frac{(\text{partition_count})(\text{lower_bound} - \text{value_expression})}{(\text{lower_bound} - \text{upper_bound})}\right) + 1$

Example

You want to create a histogram for the salaries of all employees whose salary amount ranges between $70000 and $200000. The width of each partition, or bucket, within the specified range is to be $32500.

The employee salary table contains eight employees:

```
salary   first_name   last_name
-------- ------------ -----------
50000    William      Crawford
150000   Todd         Crawford
220000   Bob          Stone
199999   Donald       Stone
70000    Betty        Crawford
70000    James        Crawford
70000    Mary         Lee
120000   Mary         Stone
```

You perform the following SELECT statement.

```
SELECT salary, WIDTH_BUCKET(salary,70000,200000,4),COUNT(salary)
FROM emp_salary
GROUP BY 1
ORDER BY 1;
```

The report produced by this statement looks like this.

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

ZEROIFNULL

Purpose

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

Syntax

```
— ZEROIFNULL —( arg )—
```

1101F226

where:

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

ANSI Compliance

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

Result Type and Attributes

Here are the default attributes for the result of ZEROIFNULL.

- If the operand is numeric, the format is the same format as *arg*.
- If the operand is character, the format is the default format for FLOAT.
- If the operand is a UDT, the format is the format of the predefined type to which the UDT is implicitly cast.

Note: The NULL keyword has a data type of INTEGER.

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

Argument Types and Rules

IF the value of arg is ...	THEN ZEROIFNULL returns ...
not null	the value of the numeric argument.
null or zero Note: A structured UDT column value is null only when you explicitly place a NULL in the column, not when a structured UDT instance has an attribute that is set to NULL.	zero.

If the argument is not numeric, it is converted to a numeric value according to implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a character string, it is converted to a numeric value of FLOAT data type.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE
  - Interval

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.
- Implicit type conversion of UDTs for system operators and functions, including ZEROIFNULL, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

ZEROIFNULL cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Example

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

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

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

Related Topics

For additional expressions involving checks for nulls, see:

- "COALESCE Expression" on page 575
- "NULLIF Expression" on page 577
- "NULLIFZERO" on page 146

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

Purpose

Performs the trigonometric or inverse trigonometric function of an argument.

Syntax

where:

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

ANSI Compliance

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

Definitions

Function	Definition
Arccosine	The arccosine is the angle whose cosine is the argument.
Arcsine	The arcsine is the angle whose sine is the argument.
Arctangent	The arctangent is the angle whose tangent is the argument.
Cosine	The cosine of an angle is the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.

Function	Definition
Sine	The sine of an angle is the ratio of two sides of a right triangle. The ratio is the length of the side opposite to the angle divided by the length of the hypotenuse.
Tangent	The tangent of an angle is the ratio of two sides of a right triangle. The ratio is the length of the side opposite to the angle divided by the length of the side adjacent to the angle.

Result Type and Attributes

Here are the default data type, format, and title for the result of the trigonometric and inverse trigonometric functions.

Data Type	Format	Title
FLOAT	Default format for FLOAT	Cos(arg) Sin(arg) Tan(arg) ArcCos(arg) ArcSin(arg) ArcTan(arg) Atan2(x,y)

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

Result Value

Function	Result Value
COS(*arg*)	The cosine of *arg* in radians in the range -1 to 1, inclusive.
SIN(*arg*)	The sine of *arg* in radians in the range -1 to 1, inclusive.
TAN(*arg*)	The tangent of *arg* in radians.
ACOS(*arg*)	An angle in the range 0 to π radians, inclusive.
ASIN(*arg*)	An angle in the range -π/2 to π/2 radians, inclusive.
ATAN(*arg*)	An angle in the range -π/2 to π/2 radians, inclusive.
ATAN2(*x*,*y*)	An angle between -π and π radians, excluding -π. A positive result represents a counterclockwise angle from the x-axis. A negative result represents a clockwise angle. ATAN2(x,y) equals ATAN(y/x), except that x can be 0 in ATAN2(x,y) and x cannot be 0 in ATAN(y/x) since this results in a divide by zero error. If both x and y are 0, an error is returned.

Argument Types and Rules

Arguments that are not FLOAT are converted to FLOAT based on implicit type conversion rules. If an argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If an argument is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.
- Implicit type conversion of UDTs for system operators and functions, including trigonometric and inverse trigonometric functions, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

Trigonometric and inverse trigonometric functions cannot take the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Examples

The following are representative function expressions and results.

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

DEGREES
RADIANS

Purpose

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

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

Syntax

```
┬─ DEGREES ─┬─ ( arg ) ─
└─ RADIANS ─┘
```

1101A481

where:

Syntax element ...	Specifies ...
arg	a numeric expression. • If the function is DEGREES, the *arg* is interpreted as an angle in radians. • If the function is RADIANS, the *arg* is interpreted as an angel in degrees.

ANSI Compliance

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

Result Title

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

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

Result Type and Format

Here are the result type and format of DEGREES and RADIANS.

- If the operand is numeric, the format is the default format for the resulting data type.
- If the operand is character, the format is the default format for FLOAT

- If the operand is a UDT, the format is the default format for the predefined type to which the UDT is implicitly cast

Note: The NULL keyword has a data type of INTEGER.

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

Argument Types and Rules

If the argument is not numeric, it is converted to a numeric value, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a character string, it is converted to a numeric value of the FLOAT data type.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DateTime
  - Interval

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.
- Implicit type conversion of UDTs for system operators and functions, including DEGREES and RADIANS, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

Neither DEGREES nor RADIANS can be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Usage Notes

DEGREES and RADIANS are useful when working with trigonometric functions such as SIN and COS, which expect arguments to be specified in radians, and inverse trigonometric functions such as ASIN and ACOS, which return values specified in radians.

Examples

Representative DEGREES and RADIANS function expressions and the results are as follows.

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

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

Purpose

Performs the hyperbolic or inverse hyperbolic function of an argument.

Syntax

```
┬─ COSH ─┬─ ( arg ) ──
├─ SINH ─┤
├─ TANH ─┤
├─ ACOSH ─┤
├─ ASINH ─┤
└─ ATANH ─┘
```

1101A483

where:

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

ANSI Compliance

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

Result Type and Attributes

Here are the default attributes for the result of hyperbolic and inverse hyperbolic functions.

Data Type	Format	Title
FLOAT	Default format for FLOAT	Hyperbolic Cos(arg) Hyperbolic Sin(arg) Hyperbolic Tan(arg) Hyperbolic ArcCos(arg) Hyperbolic ArcSin(arg) Hyperbolic ArcTan(arg)

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

Result Value

Function	Result
COSH(*arg*)	Hyperbolic cosine of *arg*.
SINH(*arg*)	Hyperbolic sine of *arg*.
TANH(*arg*)	Hyperbolic tangent of *arg*.
ACOSH(*arg*)	Inverse hyperbolic cosine of *arg*. The inverse hyperbolic cosine is the value whose hyperbolic cosine is a number so that: `acosh(cosh(arg)) = arg`
ASINH(*arg*)	Inverse hyperbolic sine of *arg*. The inverse hyperbolic sine is the value whose hyperbolic sine is a number so that: `asinh(sinh(arg)) = arg`
ATANH(*arg*)	Inverse hyperbolic tangent of *arg*. The inverse hyperbolic tangent is the value whose hyperbolic tangent is a number so that: `atanh(tanh(arg)) = arg`

Argument Types and Rules

If *arg* is not FLOAT, it is converted to a FLOAT value, based on implicit type conversion rules. If the argument cannot be converted, an error is reported. For more information on implicit type conversion, see "Implicit Type Conversions" on page 579.

If *arg* is a UDT, the following rules apply:

- The UDT must have an implicit cast to any of the following predefined types:
  - Numeric
  - Character
  - DATE

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.
- Implicit type conversion of UDTs for system operators and functions, including hyperbolic and inverse hyperbolic functions, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

Hyperbolic and inverse hyperbolic functions cannot be applied to the following types of arguments:

- BYTE or VARBYTE
- BLOB or CLOB
- CHARACTER or VARCHAR if the server character set is GRAPHIC

Examples

The following are representative hyperbolic and inverse hyperbolic function expressions and results.

Expression	Result
COSH(EXP(1))	7.61012513866229E 000
SINH(1)	1.17520119364380E 000
TANH(0)	0.00000000000000E 000
ACOSH(3)	1.76274717403909E 000
ASINH(LOG(0.1))	-8.81373587019543E -001
ATANH(LN(0.5))	-8.53988047997524E -001

CHAPTER 4 ARRAY / VARRAY Functions and Operators

This chapter describes functions, operators, expressions and methods that operate on either one-dimensional (1-D), multidimensional (n-D), or both ARRAY data types. These functions allow you to perform arithmetic, relational, and aggregate operations on an array in its entirety or on a subset of the elements composing an array.

For information on the ARRAY data type, see "ARRAY/VARRAY Data Type" in *SQL Data Types and Literals*. For information on creating one-dimensional and multi-dimensional arrays, see "CREATE TYPE (ARRAY/VARRAY Form)" in *SQL Data Definition Language Syntax and Examples*.

Note: The functions in this chapter are embedded services system functions. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

ARRAY Element Reference

Purpose

Accesses the value of a specified element in an ARRAY data type value.

Syntax

1101A863

where:

Syntax element...	Specifies...
array_expression	an expression that evaluates to an ARRAY data type.
index_value	the index to the element in the array whose value you want to access. For a 1-D ARRAY type, *index_value* must be a positive integer in the range from 1 to *n*, where *n* is the declared size of the ARRAY type. For an n-D ARRAY type, *index_value* must be a positive or negative integer in the range from *m* to *n*, where *m* is the declared lower bound of a dimension of the array, and *n* is the declared upper bound.

ANSI Compliance

ARRAY element reference for a one-dimensional ARRAY data type is ANSI SQL:2011 compliant.

ARRAY element reference for an multidimensional ARRAY data type is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

The element of the array being referenced must be initialized to a value or to null. If the referenced array element is in an uninitialized state, an error is returned. You can use the DEFAULT NULL clause to initialize all elements of an array to null at the time the ARRAY data type is created, or you can use the ARRAY constructor expression to initialize the array. For information about the DEFAULT NULL clause, see “CREATE TYPE (Array Form)” in *SQL Data Definition Language*. For information about the ARRAY constructor expression, see “ARRAY Constructor Expression” on page 197.

For a 1-D ARRAY type, you can reference a single element using the ARRAY element reference syntax with one set of square brackets. For example, MyArray[*n*], where MyArray is a 1-D ARRAY data type.

For an n-D ARRAY type, you can reference a single element using the ARRAY element reference syntax with *n* sets of square brackets, where *n* corresponds to the number of dimensions of the n-D ARRAY type. For example, if my3DArray defines a 3-D ARRAY data type, then the ARRAY element reference syntax for accessing one element of this array would be my3DArray[*x*][*y*][*z*]. Note that the maximum number of sets of square brackets is five, which corresponds to the maximum number of dimensions that Teradata supports for n-D ARRAY types.

The number of dimensions referenced in an ARRAY element reference is validated when you attempt to access the element in the array value using the element reference. This requires that an ARRAY value to be referenced already exists, and that the SQL statement actually executes the element reference. For this reason, there are some rare cases where an SQL statement that includes an ARRAY element reference may not actually be executed. For example, a SELECT from a table where the table contains no rows, or a WHERE clause in a SELECT which contains an element reference on the right-hand side of an OR clause, where the left-hand side evaluates to true. In these types of cases, the behavior of an ARRAY element reference is similar to a UDF invocation.

Using ARRAY Element Reference with a SET Clause

You can use an ARRAY element reference as the target value in a SET clause of an UPDATE or MERGE statement to set the value of an individual element in an ARRAY.

If you set an element in the middle of a 1-D array to a specific value, and the elements previous to this one are not yet initialized, then these elements will be automatically initialized to null. For an n-D ARRAY type, if you set an element in the middle of the array to a specific value, and the elements previous to this one, across all dimensions in row-major order, are not yet initialized, then these elements will be automatically initialized to null.

Note that if the element type of the array is a variable-length type, and if the value passed to initialize the element is larger than the maximum size of the element type, Teradata Database automatically truncates the passed value to the maximum size of the element type. This truncation behavior occurs with transaction processing in Teradata session mode only. In ANSI session mode, Teradata Database does not truncate the passed value, and returns an error instead. This behavior is identical to that of distinct and structured UDTs.

In addition, existing implicit CAST functionality, such as casting from CHAR to Timestamp or Timestamp to DATE, is not supported for the ARRAY element reference. The source data type for the SET clause must be the same as the element data type of the array. When the target of the SET clause is an ARRAY element reference, you must use an explicit CAST for the source value of a SET clause that has a different data type than the element type of the array.

Example 1

If a table contains a column named ephone, and the data type of the ephone column is a 1-D ARRAY, then the following shows the ARRAY element reference syntax to access the value of the 5th element of the array:

```
ephone[5]
```

Example 2

Consider the following 1-D ARRAY data type:

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];
```

The following statement creates a table with a column named ephone with a data type of phonenumbers_ary.

```
CREATE TABLE my_table (eno INTEGER, ephone phonenumbers_ary);
```

The following statement uses an ARRAY element reference to set element 3 of the array to the value '5551234567'. If elements 1 and 2 in the array were not initialized, this UPDATE statement automatically sets those elements to null.

```
UPDATE my_table
SET ephone[3] = '5551234567';
```

Example 3

Consider the following structured UDT:

```
CREATE TYPE measures_UDT AS(amplitude INTEGER,
                            phase     INTEGER,
                            frequency INTEGER);
```

The following statement creates a 3-D ARRAY data type named shots with an element type of measures_UDT.

```
CREATE TYPE shots AS measures_UDT ARRAY[-2:2][-5:5][-3:3];
```

The following statement creates a table with a column named shot_ary. The data type of the shot_ary column is shots, which is a 3-D ARRAY type.

```
CREATE TABLE seismic_table (id INTEGER, shot_ary shots);
```

The following query selects the value of an element from the 3-D array.

```
SELECT shot_ary[-1][1][3] FROM seismic_table;
```

ARRAY Constructor Expression

Purpose

Constructs a new instance of an ARRAY type and initializes it using the ARRAY constructor method or function.

Syntax

where:

Syntax element...	Specifies...
array_name	the name of the ARRAY data type that you want to initialize.
element_expression	an expression that evaluates to a literal value. The data type of the value must be the same as the element type of the *array_name* array. *element_expression* can be repeated *n* times, where *n* is the declared size of the *array_name* array. The maximum limit for *n* is 2559.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to the ARRAY constructor must be able to be implicitly converted to the data type of the element type of the *array_name* array. Existing implicit CAST functionality, such as casting from CHAR to Timestamp or Timestamp to DATE, is supported for the ARRAY constructor expression.

The ARRAY constructor expression accepts zero or more arguments. If there are zero arguments, the ARRAY constructor initializes an empty ARRAY type value and sets all the element values of the array to an uninitialized state if the ARRAY data type was created without the DEFAULT NULL clause. If the ARRAY data type was created with a DEFAULT NULL clause, then the ARRAY constructor initializes all the element values of the array to null. Note that if you try to access an array element that is in an uninitialized state, you will get an error.

You can pass up to *n* arguments to the ARRAY constructor expression, where *n* is the declared size of the ARRAY data type. For a one-dimensional ARRAY type, the ARRAY constructor fills the elements of the array with the argument values sequentially starting from the first element, in the order that the arguments are passed in. For a multidimensional ARRAY type, pass the arguments to the ARRAY constructor expression in row-major order because elements in a multidimensional array are filled in row-major order.

If the number of arguments are less than the number of elements in the array, then the remaining elements for which an argument value is not provided are set to an uninitialized state. However, if the ARRAY data type was created with the DEFAULT NULL clause, then those elements are set to null.

A constructor method for a UDT may not accept more than 128 arguments; however, this limit does not apply to ARRAY data types. The number of arguments that you can specify for the ARRAY constructor is limited only by the maximum number of elements that you can define for the ARRAY type or 2559, whichever comes first.

Note that if the element type of the array is a variable-length type, and if the value passed to initialize the element is larger than the maximum size of the element type, Teradata Database automatically truncates the passed value to the maximum size of the element type. This truncation behavior occurs with transaction processing in Teradata session mode only. In ANSI session mode, Teradata Database does not truncate the passed value, and returns an error instead. This behavior is identical to that of distinct and structured UDTs.

Invoking the ARRAY Constructor Expression Without the NEW Keyword

You can invoke the ARRAY constructor expression without using the NEW keyword. This is compatible with Oracle syntax. When invoked without the NEW keyword, Teradata Database handles the expression like a UDF expression. This restricts the places an ARRAY constructor expression can be used to the same usage as a scalar UDF. This means that in some scenarios, the ARRAY constructor expression with NEW keyword can be used in an SQL statement where the ARRAY constructor expression without NEW keyword cannot. For example, you can use the ARRAY constructor expression with NEW keyword in the DEFAULT clause of a CREATE TABLE statement. However you cannot use the ARRAY constructor expression without NEW keyword in the DEFAULT clause.

Additionally, since the ARRAY constructor expression without NEW keyword is treated like a scalar UDF, the existing search order rules for UDFs will apply in this case. Teradata Database searches for UDFs in the following databases in the following order:

1. Default database
2. SYSLIB database
3. TD_SYSFNLIB database
4. SYSUDTLIB database

If a UDF exists in the default database, SYSLIB, or TD_SYSFNLIB and has the same name as a defined ARRAY data type, and if you invoke the ARRAY constructor expression without the NEW keyword, then Teradata Database invokes the UDF instead of the ARRAY constructor. To avoid this conflict, use the NEW keyword when invoking the ARRAY constructor.

Example

Consider the following 1-D ARRAY data type:

```
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];
```

The following shows an ARRAY constructor expression that sets the first two elements in *phonenumbers_ary* to the provided values, and the rest of the elements in the array are set to an uninitialized state.

```
NEW phonenumbers_ary('5551234567', '8585551234')
```

or

```
phonenumbers_ary('5551234567', '8585551234')
```

ARRAY Scope Reference

Purpose

References one or more elements of an ARRAY data type when invoking an ARRAY function.

Syntax

array_lower_bounds, array_upper_bounds

1101A864

where:

Syntax element...	Specifies...
array_lower_bounds	an integer value or an array instance of the predefined ARRAY type ArrayVec, with a comma-separated list of integer values to define the lower bounds.
array_upper_bounds	an integer value or an array instance of the predefined ARRAY type ArrayVec, with a comma-separated list of integer values to define the upper bounds.

ANSI Compliance

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

Usage Notes

To reference one or more elements of an ARRAY data type when invoking an ARRAY function, you can use an integer value for 1-D arrays, but you must use the ArrayVec ARRAY data type for n-D arrays. The ArrayVec ARRAY data type is automatically created by the system and is defined as:

```
CREATE TYPE ArrayVec AS INTEGER ARRAY[1000];
```

For all ARRAY functions that contain an optional *scope_reference* or *array_bound* argument, you can use one or more instances of the ArrayVec ARRAY type to describe the boundaries of each dimension of an n-D ARRAY data type.

Rules

When the optional *scope_reference* parameter is specified in an ARRAY function, the following rules apply:

- You can pass two integer values or two ArrayVec values for the *scope_reference*. If the ARRAY function is called with one-dimensional ARRAY arguments, you can pass a

combination of integer and ArrayVec values for the *scope_reference*. For example, the *scope_reference* can be <*integer_value*, *ArrayVec_value*> or <*ArrayVec_value*, *integer_value*>.

- If you use an ArrayVec instance for specifying the upper or lower bounds of a *scope_reference*, the following are not allowed:
  - Passing in NULL for any of the ArrayVec elements that are needed to determine the bound of the array argument.
  - Passing an insufficient number of elements in the ArrayVec to determine the bound of the array argument.
  - Passing too many elements in the ArrayVec to determine the bound of the array argument.
- For all ARRAY functions that define the RETURNS NULL ON NULL INPUT clause, if one of the optional parameters to define the *scope_reference* is passed as NULL, then NULL is returned as the result value.
- ARRAY functions that are called with a *scope_reference* specified are considered to have their scope explicitly defined, so these functions return an error if any uninitialized elements fall within the range of the *scope_reference* in any of the ARRAY input arguments. The CARDINALITY function is an exception to this rule because an explicit scope that includes uninitialized elements is an acceptable input to the function.

If your application requires operations between 2 arrays, consider creating these arrays using the DEFAULT NULL clause at creation time. This will prevent the system from returning errors due to illegal element references since all elements of the array will be initialized.

When the optional *scope_reference* parameter is *not* specified in an ARRAY function, the following rules apply:

- If you call an ARRAY function without specifying a *scope_reference*, the default scope used is the number of initialized elements in the input ARRAY argument. If an ARRAY function accepts two ARRAY arguments and the number of initialized elements is different between the two arrays, then an error is returned.
- When an ARRAY function returns an ARRAY value, the resultant ARRAY value will have the same number of initialized elements as the input ARRAY argument.

Example

The following query returns the number of distinct elements within the range from 5 to 10 on each dimension of the phonelist array.

```
SELECT CARDINALITY(phonelist, NEW ArrayVec(5,5), NEW ArrayVec(10,10));
```

ARRAY_AGG

Purpose

An aggregate function for use with ARRAY data types that allows arrays to be created as part of the SELECT list of a query.

Syntax

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
element_value_expr	an expression that evaluates to an ARRAY element type.
value_expression	an expression that evaluates to a Teradata data type that can be compared.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
array_expression	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression

ANSI Compliance

ARRAY_AGG for a one-dimensional (1-D) ARRAY data type is ANSI SQL:2011 compliant. However, ARRAY_AGG includes an additional parameter containing an ARRAY expression of the target ARRAY type, and this is a Teradata extension to the ANSI standard. This was done to eliminate ambiguity since you can define multiple ARRAY types that have the same element data type.

ARRAY_AGG for a multidimensional (n-D) ARRAY data type is a Teradata extension to the ANSI SQL:2011 standard.

Argument Types

ARRAY_AGG accepts two arguments. The first argument is an expression evaluating to a data type that is currently being used as the element value of a defined ARRAY data type. The second argument is an expression evaluating to a defined ARRAY data type whose element type matches that of the first argument.

Result Type

The data type information from the *array_expression* which resolves to an ARRAY data type is used to set the return type of ARRAY_AGG.

Result Value and Rules

ARRAY_AGG returns an ARRAY type value whose elements are the aggregated set of element expression values that were passed in as the first parameter. For an n-D ARRAY result value, the *value_expression* values that are aggregated into the array are entered in row-major order.

If you specify the ORDER BY clause in the first argument, then the values input to the resultant array are ordered by the column values specified in this clause. The following rules apply to the ORDER BY clause of ARRAY_AGG:

- Only one value may be passed in the ORDER BY clause of ARRAY_AGG. This is different from an ORDER BY in a SELECT statement, where multiple values may be passed for ORDER BY.
- An ARRAY column value may not be passed as the sort key in the ORDER BY clause. You cannot make relational comparisons of ARRAY values. Any other data type that cannot be compared, such as a BLOB or CLOB, may not be passed as the sort key in the ORDER BY clause.
- For the single value passed to ORDER BY, any expression that may be relationally comparable may be passed. However, if the value passed is not a column value, then sorting of the resultant array elements will not occur and the results will be written to the resultant array in random order. Use a column value from one of the referenced tables in the SQL statement for proper sorting to occur.

Usage Notes

Because you can pass any *array_expression*, the ARRAY type value that you pass may or may not contain any initialized elements. However, ARRAY_AGG ignores any initialized elements

in the *array_expression*. The *array_expression* is used only for providing the data type of the result value.

Recommendation: Pass a blank instance of the desired ARRAY type value for this parameter, such as NEW myArray().

If the *array_expression* argument passed to ARRAY_AGG is a NULL literal, then an error is returned. If the *array_expression* argument can be resolved to an ARRAY data type, then the function executes normally, even if the data contains a NULL.

If the *value_expression* argument for the current row being processed evaluates to NULL, then that expression will not be eliminated from the list of values being aggregated, but will be used to set the appropriate ARRAY element value just as would be done with a non-NULL element. This follows the ANSI SQL:2011 standard and is noted in the standard as a difference between ARRAY_AGG and other aggregate functions.

ARRAY_AGG is created with a CLASS AGGREGATE value of 64000, meaning that the intermediate aggregate storage will be allocated at that size. Since the intermediate aggregate storage is a fixed-length value, it may be possible in cases of processing a very large input set that this buffer may overflow. All aggregate UDFs have this limitation.

Example

This example takes data from a source table which records a phone number associated with an employee and inserts it into a new table which records phone numbers in an ARRAY type. In many cases, an employee may have multiple phone numbers, such as office phone, mobile phone, home phone, etc. A better way to represent this type of data is with a 1-D ARRAY type which records multiple phone numbers. This reduces the number of rows in the table.

Note: the same type of behavior can be illustrated with an n-D ARRAY type. The only difference is that the storage of elements is done in row-major order.

The following statement creates a 1-D ARRAY type that can hold up to 100 phone number values:

```
CREATE TYPE emp_phone_array AS VARCHAR(14) ARRAY[100];
```

The following source table contains one row per employee phone number.

```
CREATE SET TABLE employee
   (emp_id INTEGER,
    emp_name VARCHAR(30),
    emp_phone CHAR(14));

SELECT * FROM employee;
  emp_id      emp_name           emp_phone
--------   -----------          --------------
       1         Beth           (619) 619-6190
       1         Beth           (619) 620-6200
       1         Beth           (619) 720-7200
       2         Greg           (858) 858-8580
       2         Greg           (858) 859-8590
       2         Greg           (858) 860-8600
       3         Louise         (421) 421-4210
       3         Louise         (421) 422-4220
```

```
         3         Louise         (421) 423-4230
```

The following target table contains one row per employee and stores all phone numbers associated with an employee in an ARRAY type.

```
CREATE SET TABLE employeePhoneInfo
   (emp_id INTEGER,
    emp_name VARCHAR(30),
    emp_phone emp_phone_array);

INSERT INTO employeePhoneInfo
   SELECT emp_id, emp_name,
      ARRAY_AGG(emp_phone, NEW emp_phone_array())
   FROM employee GROUP BY emp_id,emp_name
      WHERE emp_id < 100;

SELECT * FROM employeePhoneInfo;
```

```
   emp_id        emp_name     emp_phone
---------     -----------     ---------
        1            Beth     ( (619) 619-6190, (619) 620-6200, (619) 720-7200 )
        2            Greg     ( (858) 858-8580, (858) 859-8590, (858) 860-8600 )
        3          Louise     ( (421) 421-4210, (421) 422-4220, (421) 423-4230 )
```

UNNEST

Purpose

A table function for use with ARRAY data types that allows arrays to be converted into column tables.

Syntax

```
┬──────────────────┬─ UNNEST ─ ( ┬──────────────┬─ array_expr ) ┬──────────────────┬
└─ TD_SYSFNLIB. ─┘                └─ key_expr, ─┘               └─ WITH ORDINALITY ─┘
```

1101A866

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
key_expr	an optional expression that evaluates to a Teradata data type, except for ARRAY, UDT, Period data type, BLOB, or CLOB.
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression

ANSI Compliance

UNNEST for a one-dimensional (1-D) ARRAY data type is ANSI SQL:2011 compliant. However, the optional *key_expr* parameter is a Teradata extension to the ANSI SQL:2011 standard.

UNNEST for a multidimensional (n-D) ARRAY data type is a Teradata extension to the ANSI SQL:2011 standard.

Argument Types

UNNEST accepts either 1 or 2 arguments. The optional key value argument allows for grouping the rows produced in the result table. This is especially helpful when using UNNEST as a complement of the ARRAY_AGG function, because ARRAY_AGG may be invoked with the aggregate GROUP BY clause to produce multiple ARRAY values based on the grouping.

The *array_expr* argument is an expression that evaluates to an ARRAY data type that is currently defined in the system.

Result Type and Value

UNNEST returns a table that contains one, two, or three columns. The first output column corresponds to the optional key value, if it was specified. The next output column has the same data type as the element type of the ARRAY type that was passed in. If the optional WITH ORDINALITY clause was specified, then UNNEST generates an additional integer column that contains the position associated with each element.

The table is populated with one row for each element of the ARRAY value passed to UNNEST. For an n-D ARRAY input value, the column containing the element values from the input n-D ARRAY are output in row-major order.

Usage Notes

UNNEST is a table function; therefore, it can only appear in the FROM clause of an SQL SELECT statement.

If the ARRAY expression argument evaluates to NULL, then no rows will be returned in the result table. If the ARRAY expression argument is not NULL but has one or more elements that are not present, then the column value for the array element in the corresponding result row will be NULL.

Example

This example calls UNNEST, passing the optional key value argument and using the optional WITH ORDINALITY clause, using a column of 1-D ARRAY type myarray. The example returns 20 rows, one for each value stored in the myarray array for each row of the tt2 table. The result has these three columns:

- out_key is the optional key value passed to UNNEST. In this example, it corresponds to the value of the int_key column in the tt2 table, so it identifies the array values in the results that come from each row of the tt2 table.
- pos is the position of an element within the array. It is included in the results because the example uses the WITH ORDINALITY clause.
- val is the value of the element in the array.

```
CREATE TYPE myarray AS INTEGER ARRAY[10];

CREATE TABLE tt2(
   pkey INTEGER,
   int_key INTEGER,
   vc_key VARCHAR(20),
   myarr myarray);

INS INTO tt2 VALUES(0, 0, 'item 0', NEW myarray(10, 20, 30, 40, 50));
INS INTO tt2 VALUES(1, 1, 'item 1', NEW myarray(11, 21, 31, 41, 51));
INS INTO tt2 VALUES(2, 2, 'item 2', NEW myarray(12, 22, 32, 42, 52));
INS INTO tt2 VALUES(3, 3, 'item 3', NEW myarray(NULL, 23, 33, 43, NULL));
INS INTO tt2 VALUES(4, 4, 'item 4', NULL);
```

```
SELECT out_key, tf.pos, tf.val
FROM tt2,
     TABLE (UNNEST(tt2.int_key, tt2.myarr) WITH ORDINALITY) AS
tf(out_key, val, pos) WHERE tt2.int_key = tf.out_key ORDER BY 1,2;

 *** Query completed. 20 rows found. 3 columns returned.
 *** Total elapsed time was 1 second.

    out_key          pos          val
-----------  -----------  -----------
          0            1           10
          0            2           20
          0            3           30
          0            4           40
          0            5           50
          1            1           11
          1            2           21
          1            3           31
          1            4           41
          1            5           51
          2            1           12
          2            2           22
          2            3           32
          2            4           42
          2            5           52
          3            1            ?
          3            2           23
          3            3           33
          3            4           43
          3            5            ?
```

CARDINALITY

Purpose

Returns an integer representing the number of elements in an ARRAY data type that currently have assigned values, or the number of elements that are initialized within a specific scope reference.

Syntax

System function syntax:

```
── CARDINALITY ── ( array_expr ┬──────────────────┬ ) ──────────────────────────
                               └─ , scope_reference ─┘
```

1101A869

Method-style syntax:

```
── array_expr. CARDINALITY ── (┬──────────────────┬ ) ──────────────────────────
                               └─ scope_reference ─┘
```

1101A870

where:

Syntax element...	Specifies...
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200. An explicit scope that includes uninitialized elements is acceptable input when an ARRAY expression is specified in a CARDINALITY function.

ANSI Compliance

CARDINALITY for a one-dimensional (1-D) ARRAY data type is ANSI SQL:2011 compliant.

CARDINALITY for a multidimensional (n-D) ARRAY data type is a Teradata extension to the ANSI SQL:2011 standard.

Argument Type

The *array_expr* argument is an expression that evaluates to an ARRAY data type that is currently defined in the system.

Result Type and Value

CARDINALITY returns an integer value that represents the number of elements in the ARRAY that currently have assigned values. This count includes elements that are NULL. Since an ARRAY value may have fewer element values assigned than defined for its maximum size *n*, the CARDINALITY function may return a value that is smaller than *n*.

The function returns 0 if the array argument is empty (that is, it does not have any elements assigned). If the array argument is NULL, then CARDINALITY returns NULL as the result.

Even though an ARRAY function normally returns an error if any uninitialized elements fall within the range of the specified *scope_reference* in any of the ARRAY input arguments, an ARRAY expression specified within a CARDINALITY function behaves differently: an explicit scope that includes uninitialized elements is an acceptable input when an ARRAY expression is specified in a CARDINALITY function.

Examples

In the following example, an integer value is returned that represents the number of elements in the 1-D ARRAY that are currently filled in.

```
CREATE TYPE phonenumbers AS CHAR(10) ARRAY[20];

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);

SELECT CARDINALITY(phonelist)
FROM employee_info;
```

The CARDINALITY function can also be used to return the number of elements that are currently initialized within a specific scope reference. For example, the following query returns the number of initialized elements within the range from 5 to 10 of the phonelist array.

```
SELECT CARDINALITY(phonelist, 5, 10);
```

Consider the following 2-D ARRAY data type:

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_table(shots shot_ary);

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

In the following example, an integer value is returned that represents the number of elements in the 2-D ARRAY that are currently filled in.

```
SELECT CARDINALITY(shot1) FROM seismic_data;
```

The following shows the same query using method-style syntax:

```
SELECT shot1.CARDINALITY() FROM seismic_data;
```

The following query returns the number of initialized elements within the range from 5 to 10 on each dimension of the phonelist array.

```
SELECT CARDINALITY(phonelist, NEW ArrayVec(5,5), NEW ArrayVec(10,10));
```

ARRAY Concatenation Operator

Purpose

Concatenates one-dimensional ARRAY data types.

Syntax

```
──── array_expr1 ─ || ─ array_expr2 ──────────
                                   1145-046A
```

where:

Syntax element...	Specifies...
array_expr1	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
array_expr2	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
||	the concatenation operator

ANSI Compliance

The ARRAY concatenation operator for one-dimensional (1-D) ARRAY data types is partially ANSI SQL:2011 compliant.

The ARRAY concatenation operator requires that both operands are instances of the same ARRAY type and that the target type of the concatenation operation is also the same type. This is a deviation from the ANSI standard because it defines the target data type of a 1-D ARRAY concatenation to be a new 1-D ARRAY type with the length defined as the sum of the length of both operands.

Argument Type

array_expr1 and *array_expr2* must have the same 1-D ARRAY data type as defined by a CREATE TYPE statement.

Result Type and Value

The result of the concatenation operation is a new 1-D ARRAY value of the same type as the data type of the two arguments, where all of the elements present in *array_expr1* are followed by all the elements that are present in *array_expr2*. Elements that are present include NULL elements, but do not include elements that are in an uninitialized state.

Since the result value and the argument values are of the same 1-D ARRAY type, they have the same maximum size *n*. Therefore, if the number of elements that results from the concatenation of *array_expr1* and *array_expr2* is greater than the maximum size *n* of the defined 1-D ARRAY type, the operation will abort with an error.

If either argument is NULL, the result of the operation is NULL.

Restrictions

The ARRAY concatenation operator cannot be used with multidimensional ARRAY data types.

Example

This example takes two 1-D ARRAY values of the same data type and concatenates them together.

```
CREATE TYPE address AS CHAR(10) ARRAY[5];

CREATE TABLE employee_info (eno INTEGER, addressval address);

/* Assume one row in table employee_info contains the following value
for "addressval":
Addressval[1] = '123 Main St.'
Addressval[2] = 'San Diego'
Addressval[3] = 'CA'
*/

/* The following select statement concatenates the current value in
"addressval" with ZIP code and country information, which are stored as
additional elements of the 1-D array. */

SELECT addressval || NEW address('92101', 'USA');
FROM employee_info;

/* Result value is the following:
Addressval[1] = '123 Main St.'
Addressval[2] = 'San Diego'
Addressval[3] = 'CA'
Addressval[4] = '92101'
Addressval[5] = 'USA'
*/
```

ARRAY Concatenation Function

Purpose

Concatenates one-dimensional ARRAY data types and can be applied to a subset of the elements of the array.

Syntax

System function syntax:

```
── ARRAY_CONCAT ── ( expr1, expr2 ──┬──────────────────┬── ) ──────────────
                                    └─ , scope_reference ─┘
```

1101A868

Method-style syntax:

```
── expr1. ARRAY_CONCAT ── ( expr2 ──┬──────────────────┬── ) ──────────────
                                    └─ , scope_reference ─┘
```

1101A867

where:

Syntax element...	Specifies...
expr1	An ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
expr2	An ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Argument Type

expr1 and *expr2* must be ARRAY expressions, and they must have the same 1-D ARRAY data type as defined by a CREATE TYPE statement.

Result Type and Value

The result of the concatenation operation is a new 1-D ARRAY value of the same type as the data type of the two arguments, where all of the elements present in *expr1* are followed by all the elements that are present in *expr2*. Elements that are present include NULL elements, but do not include elements that are in an uninitialized state.

Since the result value and the argument values are of the same 1-D ARRAY type, they have the same maximum size *n*. Therefore, if the number of elements that results from the concatenation of *expr1* and *expr2* is greater than the maximum size *n* of the defined 1-D ARRAY type, the operation will abort with an error.

If either argument is NULL, the result of the operation is NULL.

Restrictions

The ARRAY concatenation function cannot be used with multidimensional ARRAY data types.

Example

This example takes two 1-D ARRAY values of the same data type and concatenates them together. PhoneNum and OldPhoneNum are columns in the employee_info table, and both columns have the same 1-D ARRAY data type.

```
/* The following select statement concatenates the current value in
"PhoneNum" with the value in "OldPhoneNum". */

/* Assume one row in table employee_info contains the following value for
"PhoneNum":
PhoneNum[1] = '6197211000'
PhoneNum[2] = '6197221000'
PhoneNum[3] = '6197231000'

Also, assume one row in table employee_info contains the following value
for "OldPhoneNum":
OldPhoneNum[1] = '8582001000'
OldPhoneNum[2] = '8582002000'
OldPhoneNum[3] = '8582003000'
*/

SELECT ARRAY_CONCAT(PhoneNum, OldPhoneNum)
FROM employee_info;

/* Result value is the following:
PhoneNum[1] = '6197211000'
PhoneNum[2] = '6197221000'
PhoneNum[3] = '6197231000'
PhoneNum[4] = '8582001000'
PhoneNum[5] = '8582002000'
```

```
PhoneNum[6] = '8582003000'
*/
```

The following shows the same query using method-style syntax:

```
SELECT PhoneNum.ARRAY_CONCAT(OldPhoneNum)
FROM employee_info;
```

ARRAY Comparison Functions

Purpose

Performs comparisons on the individual elements of an ARRAY value, whether referred to in its entirety or over a scope reference.

Syntax

System function syntax:

Method-style syntax:

where:

Syntax element...	Specifies...
expr1	an ARRAY expression or a scalar expression. An ARRAY expression is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression The element type of the array must be a numeric data type.

Syntax element…	Specifies…
expr2	an ARRAY expression or a scalar expression. An ARRAY expression is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression The element type of the array must be a numeric data type.
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Argument Type

Either *expr1* or *expr2* or both arguments must be an ARRAY expression. For details, see "Usage Notes".

Supported ARRAY Comparison Functions

The following table provides a description of the supported ARRAY comparison functions.

Relational Function	Description
ARRAY_GT	Greater than function. Compares two expressions and determines whether *expr1* is greater than *expr2*. If it is, the function returns a non-zero value in the corresponding result ARRAY element. If *expr1* is less than or equal to *expr2*, the function returns zero in the corresponding result ARRAY element.
ARRAY_GE	Equal or greater than function. Compares two expressions and determines whether *expr1* is greater than or equal to *expr2*. If it is, the function returns a non-zero value in the corresponding result ARRAY element. If *expr1* is less than *expr2*, the function returns zero in the corresponding result ARRAY element.
ARRAY_LT	Less than function. Compares two expressions and determines whether *expr1* is less than *expr2*. If it is, the function returns a non-zero value in the corresponding result ARRAY element. If *expr1* is greater than or equal to *expr2*, the function returns zero in the corresponding result ARRAY element.
ARRAY_LE	Equal or less than function. Compares two expressions and determines whether *expr1* is less than or equal to *expr2*. If it is, the function returns a non-zero value in the corresponding result ARRAY element. If *expr1* is greater than *expr2*, the function returns zero in the corresponding result ARRAY element.

Relational Function	Description
ARRAY_EQ	Equality function. Compares two expressions and determines whether *expr1* is equal to *expr2*. If it is, the function returns a non-zero value in the corresponding result ARRAY element. If *expr1* is not equal to *expr2*, the function returns zero in the corresponding result ARRAY element.
ARRAY_NE	Non-equality function. Compares two expressions and determines if *expr1* is not equal to *expr2*. If the two expressions are not equal, the function returns a non-zero value in the corresponding result ARRAY element. If *expr1* is equal to *expr2*, the function returns zero in the corresponding result ARRAY element.

Usage Notes

For the relational functions ARRAY_GT, ARRAY_GE, ARRAY_LT, ARRAY_LE, ARRAY_EQ, and ARRAY_NE, two forms are supported for ARRAY types.

In the first form, both left-hand side (LHS) and right-hand side (RHS) operands are two instances of the same ARRAY type.

The relational operation is performed in a pairwise fashion, one by one for each matching pair of elements in the two arguments. If the optional argument *scope_reference* is defined, then the operation is performed only over the elements within the scope. All elements outside the boundaries of *scope_reference* are set to NULL.

The result is an ARRAY of the same type as the input array arguments, with three possible values: 1 (true), 0 (false), or NULL for all numeric types except DECIMAL(m,n) where m=n. For DECIMAL(m,n) where m=n, the function cannot store a value of 1 because no digits are permitted to the LHS of the decimal point. In this case, the function returns .9[0]. Any higher level of precision above 1 is padded with zero to the right.

In the second form, one of either LHS or RHS is an ARRAY with a valid Teradata data type, and the other operand is a numeric value of the same data type as the one defined for the ARRAY in LHS.

The relational operation is performed by applying the numeric value argument to each element of the ARRAY argument. If the optional argument *scope_reference* is defined, then the operation is performed only over the elements within the scope. All elements outside the boundaries of *scope_reference* are set to NULL.

The result is an ARRAY of the same type as the input array argument, with three possible values: 1 (true), 0 (false), or NULL for all numeric types except DECIMAL(m,n) where m=n. For DECIMAL(m,n) where m=n, the function cannot store a value of 1 because no digits are permitted to the LHS of the decimal point. In this case, the function returns .9[0]. Any higher level of precision above 1 is padded with zero to the right. If RHS evaluates to NULL, then NULL is returned.

For both forms of the relational comparison functions, if a NULL element is encountered in an ARRAY argument, the resulting ARRAY will contain a NULL element in that position. Comparison involving arrays are not affected by the optional DEFAULT NULL clause in the

CREATE TYPE statement. However, the probability of getting an error is increased when comparing arrays that have been created without the DEFAULT NULL clause.

If an ARRAY argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the ARRAY that contains uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

Restrictions

You cannot make relational comparisons of ARRAY values. You cannot use any of the relational comparison operators on ARRAY data. You can only make relational comparisons on individual elements of an ARRAY.

Example 1

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[10];

CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

Assume the following element values for the sale_price and regular_price arrays:

```
sale_price[1:10] =  50, 100,  200,  90,  250, 550, 200, 200,  50,  75
regular_price[1:10] = 100, 300, 230, 110, 500, 550, 200, 400, 100, 150
```

The following query returns a 1-D ARRAY with scope reference [1:10] of BYTEINT element type. During evaluation, each element within the specified scope, [5:10], in the regular_price array is compared with the corresponding element of the sale_price array using the relational function ARRAY_GT. The resulting values in the output array are 0 or a non-zero number for the elements within the scope reference, and NULL for all other elements of the array.

```
SELECT ARRAY_GT(regular_price, sale_price, 5,10)
FROM inventory;
```

The output from the query is a 1-D ARRAY with the following values:

```
output_array[1:10] = [ NULL, NULL, NULL, NULL, 1, 0, 0, 1, 1, 1]
```

The following query returns a 1-D ARRAY with scope reference [1:10] of type item_price. During evaluation, each element within the specified scope [5:10] in the regular_price array is compared with the corresponding element of the sale_price array using the relational function ARRAY_GT. The resulting ARRAY of element type BYTEINT value (0 or non-zero number) is multiplied by the corresponding element value of the regular_price array.

```
SELECT ARRAY_MUL(regular_price, ARRAY_GT(sale_price, regular_price, 5,
10))
FROM inventory;
```

The output of the query is a 1-D ARRAY with the following values:

```
output_array[1:10] = [ NULL, NULL, NULL, NULL, 500, 0, 0, 400, 100, 150]
```

In the following query, the relational function ARRAY_LT compares each element within the scope of the 1-D ARRAY sale_price with a literal value of 100. If the element value is less than 100, the comparison function returns a non-zero value, otherwise it returns 0if the condition is not satisfied, or NULL if the element is not initialized. A 1-D ARRAY of item_price ARRAY type is returned with the corresponding values for each element of the input 1-D ARRAY.

```
SELECT ARRAY_LT(sale_price,100)
FROM inventory;
```

The output of the query is a 1-D ARRAY with the following values:

```
output_array[1:10] = [ 1, 0 , 0, 1, 0, 0, 0, 0, 1, 1]
```

Example 2

Consider the following 2-D ARRAY data type and table:

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER DEFAULT NULL;

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

The following query returns a 2-D ARRAY with an element type of INTEGER. The size of the output array is the same as that of the input array argument. During evaluation, the element in position [5][10] of the shot1 array is compared to a value of 5. If the element is greater than 5, the value for the corresponding element in the output array is set to a non-zero value, otherwise it is set to 0. All other elements in the output array are set to NULL.

```
SELECT ARRAY_GT(shot1, 5, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

The following is the same query using method-style syntax:

```
SELECT shot1.ARRAY_GT(5, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

In the following query, the relational function ARRAY_LT compares each element within the scope reference [3:5][8:10] of the 2-D ARRAY shot1 with a literal value of 0. If the element value is less than 0 the comparison function returns 0, otherwise it returns a non-zero value. The resulting array of shot_ary type is then multiplied by the shot1 array.

```
SELECT ARRAY_MUL(shot1, ARRAY_LT(shot1, 0, NEW arrayVec(3,8), NEW
arrayVec(5,10)))
FROM seismic_data;
```

ARRAY Arithmetic Functions

Purpose

Performs arithmetic operations (add, subtract, multiply, divide, and mod) on an ARRAY value, whether referred to in its entirety or over a scope reference.

Syntax

System function syntax:

Method-style syntax:

where:

Syntax element...	Specifies...
expr1	an ARRAY expression. The element type of the array must be a numeric data type. An ARRAY expression is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression

Syntax element...	Specifies...
expr2	an ARRAY expression or a numeric value. The element type of the array must be a numeric data type. An ARRAY expression is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Argument Types

Valid arguments for the ARRAY arithmetic functions are one of the following:

- *expr1* and *expr2* are each an instance of the same ARRAY type.
- *expr1* is an ARRAY value with a numeric argument type and *expr2* is a numeric value that can be assigned to the element type of *expr1*.

If either *expr1* or *expr2* is NULL, the function returns NULL.

Usage Notes

For the arithmetic functions ARRAY_ADD, ARRAY_SUB, ARRAY_MUL, ARRAY_DIV, and ARRAY_MOD, two forms are supported for ARRAY types.

In the first form, both left-hand side (LHS) and right-hand side (RHS) operands are two instances of the same ARRAY type.

The arithmetic operation is performed in a pairwise fashion, one by one for each matching pair of elements in the two arguments. If the optional argument *scope_reference* is defined, then the operation is performed only over the elements within the scope. All elements outside the boundaries of *scope_reference* are set to NULL. The result is an ARRAY of the same data type as the input ARRAY arguments, with cardinality (number of populated elements) the same as the first input ARRAY argument.

In the second form, one of either LHS or RHS is an ARRAY value with a numeric argument type, and RHS is a numeric value that can be assigned to the element type of LHS. The arithmetic operation is performed by applying the RHS value with the arithmetic operation to each element of the ARRAY argument on the LHS. The result is an ARRAY of the same type as the input ARRAY argument, with cardinality (number of populated elements) the same as the input ARRAY argument. If the optional argument *scope_reference* is defined, then the

operation is performed only over the elements within the scope. All elements outside the boundaries of *scope_reference* are set to NULL.

For both forms of the arithmetic functions, if a NULL element is encountered in an ARRAY argument, the resulting ARRAY will contain a NULL element in that position. Arithmetic operations involving arrays are not affected by the optional DEFAULT NULL clause in the CREATE TYPE statement. However, the probability of getting an error is increased when comparing arrays that have been created without the DEFAULT NULL clause.

If an ARRAY argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the ARRAY that contains uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

Result Type and Value

The result is an ARRAY of the same size as the maximum cardinality of the input array arguments, and the element type is the same as that of *expr1*. Note that the size of the result array type refers to its current cardinality.

Example 1

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];

CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price_diff item_price);
```

Assume the following element values for the sale_price_diff and regular_price arrays:

```
sale_price_diff[1:20] =  100, 200, 120, 140, 50, 160, 45, 10, 90, 100
                          50, 100, 200, 90, 250, 550, 200, 200, 50, 75
regular_price[1:20] =  50, 90, 80, 10, 45, 30, 20, 10, 90, 100,
                      100, 300, 230, 110, 500, 550, 200, 400, 100, 150
```

The following query returns a 1-D ARRAY of element type item_price. During evaluation, each element within the specified scope in the regular_price array is combined with the corresponding element of the sale_price_diff array using the arithmetic function ARRAY_SUB.

```
SELECT ARRAY_SUB(regular_price, sale_price_diff, 10, 20)
FROM inventory;
```

The query returns a 1-D ARRAY with the following values:

```
output_array[1:20] = [ NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, 50, 200, 30, 20, 250, 0, 0, 200, 50, 75]
```

In the following query, a literal value of 2 is added to all elements in the 1-D ARRAY sale_price_diff.

```
SELECT ARRAY_ADD(sale_price_diff, 2) FROM inventory;
```

This query returns a 1-D ARRAY with the following values:

```
output_array[1:20] = [ 102, 202, 122, 142, 52, 162, 47, 12, 92, 102
                       52, 102, 202, 92, 252, 552, 202, 202, 52, 77]
```

The following query shows the use of a filtering condition while performing arithmetic operations on an ARRAY. In this example, all elements within the scope [10:20] that have a value less than 1000 are multiplied by 2.

```
SELECT ARRAY_MUL(regular_price, 2, 10, 20) FROM inventory
WHERE ARRAY_COUNT_DISTINCT(ARRAY_LT(regular_price,1000,10,20),1)>1;
```

This query returns a 1-D ARRAY with the following values:

```
output_array[1:20] = [NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, 200, 600, 460, 220, 1000, 1100, 400, 800, 200, 300]
```

Example 2

Consider the following 2-D ARRAY data type and table:

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER DEFAULT NULL;

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

The following query returns an ARRAY of type shot_ary with elements in scope reference [10:20][10:20] modified as a result of the operation. During evaluation, each element within the specified scope reference in shot1 is combined with the corresponding element of shot2 using the arithmetic function ARRAY_ADD.

```
SELECT ARRAY_ADD(shot1, shot2, NEW arrayVec(10, 20),
   NEW arrayVec(10,20))
FROM seismic_data;
```

In the following query, a literal value of 9 is added to all elements within the specified scope reference of the shot1 array.

```
SELECT ARRAY_ADD(shot1, 9, NEW arrayVec(10,10), NEW arrayVec(20,20))
FROM seismic_data;
```

The following is the same query using method-style syntax:

```
SELECT shot1.ARRAY_ADD(9, NEW arrayVec(10,10), NEW arrayVec(20,20)) FROM
seismic_data;
```

The following query shows the use of a filtering condition while performing arithmetic operations on an n-D ARRAY. In this example, all elements within the scope reference [10:20][10:20] that have a negative value are multiplied by zero.

```
SELECT ARRAY_MUL(shot1, 0, NEW arrayVec(10,10), NEW arrayVec(20,20))
FROM seismic_data
WHERE ARRAY_COUNT_DISTINCT(ARRAY_LT(shot1, 0, NEW arrayVec(10,10),
   NEW arrayVec(20,20)),0)>1;
```

ARRAY_SUM

Purpose

Returns a value representing the total sum of adding the values of each element in *array_expr* or the sum of the elements within the specified scope.

Syntax

System function syntax:

```
── ARRAY_SUM ── ( array_expr ┬──────────────────┬ ) ──────────────────────────
                             └─ scope_reference ─┘
```

1101A980

Method-style syntax:

```
── array_expr.ARRAY_SUM ── ( ┬──────────────────┬ ) ──────────────────────────
                             └─ scope_reference ─┘
```

1101A981

where:

Syntax element...	Specifies...
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression The element type of the array must be a numeric data type.
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Usage Notes

ARRAY_SUM takes an array expression as an argument and returns a scalar value representing the total sum of adding the values of each element in the array argument. The array argument must have a numeric element type. If you specify a scope reference, the summation is applied only to the elements within the limits of the given scope.

The affected elements cannot be filtered through a conditional expression. Therefore, a SELECT statement involving ARRAY_SUM must not contain an array relational expression in the WHERE clause.

If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation. If the array argument contains any elements that is in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

If *array_expr* is NULL, the function returns NULL.

Result Type

The result type of ARRAY_SUM is NUMBER unless the element type of the array argument is FLOAT. In this case, the result type is FLOAT.

Example 1

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];

CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

When evaluating the following query, each element value of the regular_price array is added together. The resulting value is divided by 2.

```
SELECT ARRAY_SUM(regular_price) / 2 FROM inventory;
```

The following is the same query using method-style syntax.

```
SELECT regular_price.ARRAY_SUM() / 2 FROM inventory;
```

In the following query, ARRAY_SUM adds each element within the specified scope of the regular_price array. The query returns a value representing the total sum of adding the affected element values of the regular_price array.

```
SELECT ARRAY_SUM(regular_price, NEW arrayVec(5,10))FROM inventory;
```

The following shows an alternate way to specify the same query.

```
SELECT ARRAY_SUM(regular_price, 5, 10) FROM inventory;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

When evaluating the following query, each element value in the shot1 array is added together. The resulting value is divided by 2.

```
SELECT ARRAY_SUM(shot1) / 2 FROM seismic_data;
```

In the following query, ARRAY_SUM adds each element within the specified scope reference of the shot1 array. The query returns a value representing the total sum of adding the affected element values of the shot1 array.

```
SELECT ARRAY_SUM(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10)) FROM
seismic_data;
```

ARRAY_AVG

Purpose

Returns the average of all the element values in *array_expr* or the average of the elements within the specified scope.

Syntax

System function syntax:

```
── ARRAY_AVG ── ( array_expr ┬──────────────────┬ ) ──────────────────────────
                             └─ scope_reference ─┘
```

1101A978

Method-style syntax:

```
── array_expr.ARRAY_AVG ── ( ┬──────────────────┬ ) ──────────────────────────
                             └─ scope_reference ─┘
```

1101A979

where:

Syntax element...	Specifies...
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression The element type of the array must be a numeric data type.
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Usage Notes

ARRAY_AVG takes an array expression as an argument and returns a scalar value of type FLOAT. This value represents the result of adding the values of each element in the array

argument, and dividing this result by the total number of elements. The array argument must have a numeric element type. If you specify a scope reference, the function is applied only to the elements within the limits of the given scope.

The affected elements cannot be filtered through a conditional expression. Therefore, a SELECT statement involving ARRAY_AVG must not contain an array relational expression in the WHERE clause.

If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation. If the array argument contains any elements that is in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

If *array_expr* is NULL, the function returns NULL.

Result Type

The result data type of ARRAY_AVG is FLOAT.

Example 1

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];

CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

When evaluating the following query, each element value of the regular_price array is added. The total sum of all the element values is then divided by the number of elements in the regular_price array.

```
SELECT ARRAY_AVG(regular_price) FROM inventory;
```

In the following query, ARRAY_AVG adds each element within the specified scope of the regular_price array. The result is a scalar value representing the total sum of adding the affected element values of regular_price divided by the total number of elements composing the specified scope.

```
SELECT ARRAY_AVG(regular_price, 5, 10)FROM inventory;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

When evaluating the following query, each element value in the shot1 array is added. The total sum of all the element values is then divided by the number of elements in the shot1 array.

```
SELECT ARRAY_AVG(shot1) FROM seismic_data;
```

In the following query, ARRAY_AVG adds each element within the specified scope reference of the shot1 array. The result is a scalar value representing the total sum of adding the affected element values of shot1 divided by the total number of elements composing the specified scope reference.

```
SELECT ARRAY_AVG(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

The following is the same query using method-style syntax.

```
SELECT shot1.ARRAY_AVG(NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

ARRAY_MAX

Purpose

Returns the maximum of all element values in *array_expr* or the maximum of the elements within the specified scope.

Syntax

System function syntax:

```
── ARRAY_MAX ── ( array_expr ┬──────────────────┬ ) ──────────────────────────
                             └─, scope_reference─┘
```

1101A976

Method-style syntax:

```
── array_expr.ARRAY_MAX ── ( ┬──────────────────┬ ) ──────────────────────────
                             └─, scope_reference─┘
```

1101A977

where:

Syntax element...	Specifies...
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression The element type of the array must be a numeric data type.
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Usage Notes

ARRAY_MAX takes an array expression as an argument and returns an element value that represents the maximum value of all the elements composing the array. If you specify a scope reference, the function is applied only to the elements within the limits of the given scope.

The affected elements cannot be filtered through a conditional expression. Therefore, a SELECT statement involving ARRAY_MAX must not contain an array relational expression in the WHERE clause.

If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation. If the array argument contains any elements that is in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

If *array_expr* is NULL, the function returns NULL.

Result Type

The result type is the same as the element type of the array argument.

Example 1

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];

CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

The following query returns an element value, which is the maximum of all the element values in the regular_price array.

```
SELECT ARRAY_MAX(regular_price) FROM inventory;
```

The following is the same query using method-style syntax.

```
SELECT regular_price.ARRAY_MAX() FROM inventory;
```

The following query returns an element value, which is the maximum of the element values within the specified scope of the regular_price array.

```
SELECT ARRAY_MAX(regular_price, 5, 10)FROM inventory;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

The following query returns an element value, which is the maximum of all the element values in the shot1 array.

```
SELECT ARRAY_MAX(shot1) FROM seismic_data;
```

The following query returns an element value, which is the maximum of the element values within the specified scope of the shot1 array.

```
SELECT ARRAY_MAX(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

ARRAY_MIN

Purpose

Returns the minimum of all element values in *array_expr* or the minimum of the elements within the specified scope.

Syntax

System function syntax:

```
── ARRAY_MIN ── ( array_expr ┬──────────────────┬ ) ──────────────────────────
                             └─ , scope_reference ─┘
```

1101A974

Method-style syntax:

```
── array_expr.ARRAY_MIN ── ( ┬──────────────────┬ ) ──────────────────────────
                             └─ , scope_reference ─┘
```

1101A975

where:

Syntax element...	Specifies...
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression The element type of the array must be a numeric data type.
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Usage Notes

ARRAY_MIN takes an array expression as an argument and returns an element value that represents the minimum value of all the elements composing the array. If you specify a scope reference, the function is applied only to the elements within the limits of the given scope.

The affected elements cannot be filtered through a conditional expression. Therefore, a SELECT statement involving ARRAY_MIN must not contain an array relational expression in the WHERE clause.

If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation. If the array argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

If *array_expr* is NULL, the function returns NULL.

Result Type

The result type is the same as the element type of the array argument.

Example 1

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];

CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

The following query returns an element value, which is the minimum of all the element values in the regular_price array.

```
SELECT ARRAY_MIN(regular_price) FROM inventory;
```

The following query returns an element value, which is the minimum of the element values within the specified scope of the regular_price array.

```
SELECT ARRAY_MIN(regular_price, 5, 10)FROM inventory;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

The following query returns an element value, which is the minimum of all the element values in the shot1 array.

```
SELECT ARRAY_MIN(shot1) FROM seismic_data;
```

The following query returns an element value, which is the minimum of the element values within the specified scope of the shot1 array.

```
SELECT ARRAY_MIN(shot1, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

The following is the same query using method-style syntax.

```
SELECT shot1.ARRAY_MIN(NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

ARRAY_COUNT_DISTINCT

Purpose

Returns the number of distinct elements in *array_expr*, optionally matching a specific input value.

Syntax

System function syntax:

```
── ARRAY_COUNT_DISTINCT ── ( array_expr ┬──────────────────┬──────────────────┬ ) ──
                                        └, scope_reference┘ └, matching_expr   ┘
                                                                                 1101A972
```

Method-style syntax:

```
── array_expr.ARRAY_COUNT_DISTINCT ── ( ┬──────────────────┬──────────────────┬ ) ──
                                        └scope_reference   ┘ └, matching_expr   ┘
                                                                                 1101A973
```

where:

Syntax element...	Specifies...
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression The element type of the array must be a numeric or character data type.
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.
matching_expr	an optional expression to be matched by one or more elements in the array.

ANSI Compliance

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

Usage Notes

ARRAY_COUNT_DISTINCT takes an array expression as an argument and returns a value that represents one of two possible outputs:

- If you specify the optional argument *matching_expr*, then the function returns the number of elements in *array_expr* whose values are equal to *matching_expr*. If *matching_expr* is NULL, the function returns the number of elements with NULLs.
- If you do not specify *matching_expr*, then the function returns the number of distinct elements in *array_expr*. If a NULL element is encountered in the array argument, it is ignored and not considered when doing the calculation.

If you specify a scope reference, the function is applied only to the elements within the limits of the given scope. You can use ARRAY_COUNT_DISTINCT with arrays of numeric as well as character element types.

If the array argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the array with uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

If *array_expr* is NULL, the function returns NULL.

Result Type

The result type is an INTEGER.

Example 1

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS DECIMAL(7,2) ARRAY[20];

CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

In the following query, ARRAY_COUNT_DISTINCT returns the number of elements whose value is equal to 100 within the scope reference of the regular_price array.

```
SELECT ARRAY_COUNT_DISTINCT(regular_price, 5, 10, 100) FROM inventory;
```

The following is the same query using method-style syntax.

```
SELECT regular_price.ARRAY_COUNT_DISTINCT(5, 10, 100) FROM inventory;
```

The following query returns the number of distinct elements in the regular_price array.

```
SELECT ARRAY_COUNT_DISTINCT(regular_price)FROM inventory;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_data (
```

```
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

In the following query, ARRAY_COUNT_DISTINCT returns the number of elements whose value is equal to 100 within the specified scope of the shot1 array.

```
SELECT ARRAY_COUNT_DISTINCT(shot1, NEW arrayVec(5,5), NEW
arrayVec(10,10), 100) FROM seismic_data;
```

ARRAY_GET

Purpose

Returns the element value in *array_expr* that corresponds to the specified index position.

Syntax

System function syntax:

```
── ARRAY_GET ── ( array_expr, array_index ) ──────────────────────────────
```

1101A970

Method-style syntax:

```
── array_expr.ARRAY_GET ── ( array_index ) ──────────────────────────────
```

1101A971

where:

Syntax element...	Specifies...
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
array_index	one of the following: • an integer value. • an array instance of the predefined array type ArrayVec with a comma-separated list of integer values to define the bounds. For details, see "ARRAY Scope Reference" on page 200. The value of *array_index* must be within the limits of *array_expr*.

ANSI Compliance

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

Usage Notes

ARRAY_GET takes an array expression as an argument and returns the element value in *array_expr* that corresponds to the position specified by *array_index*.

If *array_expr* is a one-dimensional ARRAY type, the index of the element to be located is defined by an INTEGER or an ArrayVec type that must be within the defined boundaries of *array_expr*.

If *array_expr* is a multidimensional ARRAY type, the index of the element to be located is defined using the predefined array type ArrayVec. For details on the ArrayVec type, see "ARRAY Scope Reference" on page 200. The number of dimensions defined by ArrayVec must be between 2 and 5 (the maximum number of dimensions supported) and must correspond to the number of dimensions in *array_expr*. The values for each dimension are separated by a comma and they must be within the defined boundaries of *array_expr*.

If the value of *array_index* references an element of the ARRAY which is in an uninitialized state, an error is returned. To avoid referencing an element of the ARRAY that is uninitialized, set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

If *array_expr* is NULL, the function returns NULL.

Result Type

The result type is the same as the element type of the array argument.

Example 1

Consider the following 1-D ARRAY data type and table.

```
CREATE TYPE item_price AS VARRAY(50) OF DECIMAL(7,2);

CREATE TABLE inventory (itemkind INTEGER,
                        regular_price item_price,
                        sale_price item_price);
```

In the following query, ARRAY_GET returns the element value located in position 40 of the regular_price array.

```
SELECT ARRAY_GET(regular_price,40) FROM inventory;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

In the following query, ARRAY_GET returns the element value located in position [5][10] of the shot1 array.

```
SELECT ARRAY_GET(shot1, NEW arrayVec(5,10)) FROM seismic_data;
```

The following is the same query using method-style syntax.

```
SELECT shot1.ARRAY_GET(NEW arrayVec(5,10)) FROM seismic_data;
```

ARRAY_COMPARE

Purpose

Performs a pairwise comparison of the elements of two arrays and returns a value of 1, 0, or NULL to indicate whether or not the two array arguments contain the same element values.

Syntax

System function syntax:

```
── ARRAY_COMPARE ── ( expr1, expr2 ┬──────────────────────────────────┬ ) ──────────
                                   └─ , scope_reference, nulls_equal_flag ─┘
```

1101A969

Method-style syntax:

```
── expr1.ARRAY_COMPARE ── ( expr2 ┬──────────────────────────────────┬ ) ──────────
                                  └─ , scope_reference, nulls_equal_flag ─┘
```

1101A968

where:

Syntax element...	Specifies...
expr1	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
expr2	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.
nulls_equal_flag	an optional flag used to specify how NULL element values should be compared. Valid values for this parameter are 0 and 1.

ANSI Compliance

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

The function does a pairwise comparison of the elements of two arrays, following the rules for ARRAY comparison according to the ANSI standard.

Usage Notes

The ARRAY_COMPARE function does a pairwise comparison of the two array arguments, and returns a scalar value indicating whether or not the two array expressions contain all of the same element values.

ARRAY_COMPARE follows the rules for ANSI array comparison as follows:

1. If either *expr1* or *expr2* evaluates to NULL, the result is NULL.
2. *expr1* and *expr2* are equal if and only if *all* the element values of *expr1* are equal to the element values of *expr2*.
3. An element value of *expr1* is equal to an element value of *expr2* if and only if they both have the same value and both have the same position in the array.
4. If any non-NULL elements do not match for a given element of Array A and Array B, then this is considered a mismatch and ARRAY_COMPARE returns a value of 0.
5. If either or both values are NULL for a given element, and all other elements in the arrays match, this is considered as a mismatch according to the ANSI standard. The result in this case, according to the ANSI standard, is UNKNOWN. In Teradata, this evaluates to a NULL result.

If rule (1) or (5) above is satisfied, ARRAY_COMPARE returns NULL. If rules (2) and (3) are satisfied, ARRAY_COMPARE returns 1. Otherwise, ARRAY_COMPARE returns 0.

When the optional *nulls_equal_flag* is set to 1, then when both values are NULL for a given element, this is considered as a match, which is a deviation from the ANSI standard. If all other elements match, then rule (2) above applies.

If you specify the optional argument *scope_reference*, then the function is applied only to the elements within the limits of the given scope.

If an array argument contains any elements that are in an uninitialized state, an error is returned. Use a scope reference to avoid referencing a range of the ARRAY that contains uninitialized elements, or set any uninitialized elements to NULL. You can do this with the OEXTEND method. See "OEXTEND" on page 274.

Restrictions

You cannot make relational comparisons of ARRAY values. You cannot use any of the relational comparison operators on ARRAY data. You can only make relational comparisons on individual elements of an ARRAY.

Examples

In this example, the inventory table contains two ARRAY columns, sale_price_june and sale_price_july, which contain the following element values:

```
sale_price_june[1:10] = 50, 100, 200, 90, 250, 550, 200, 200, 50, 75
sale_price_july[1:10] = 50, 100, 200, 90, 250, 550, 200, 200, 50, 75
```

The output of the following query is 1 since all of the element values in the sale_price_june array are equal to the element values in the sale_price_july array.

```
SELECT ARRAY_COMPARE(sale_price_june, sale_price_july)
FROM inventory;
```

The following is the same query using method-style syntax.

```
SELECT sale_price_june.ARRAY_COMPARE(sale_price_july)
FROM inventory;
```

Consider the same arrays but with different element values as follows:

```
sale_price_june[1:10] = 50, 100, 200, 90, 250, 550, 200, 200, 50, 75
sale_price_july[1:10] = 50, 100, 200, 90, 300, 550, 200, 200, 50, 75
```

The output of the following query is 0 since the element value in position 5 differs in the two arrays.

```
SELECT ARRAY_COMPARE(sale_price_june, sale_price_july)
FROM inventory;
```

The following query compares the values of elements in positions 6 through 10. The output of this query is 1 since the element values in the given positions of both arrays are equal.

```
SELECT ARRAY_COMPARE(sale_price_june, sale_price_july, 6, 10)
FROM inventory;
```

ARRAY_UPDATE

Purpose

Updates all or a subset of the elements in *array_expr* to the specified new value.

Syntax

System function syntax:

```
── ARRAY_UPDATE ── ( array_expr, new_value ──┬──────────────────────────┬──── ) ──
                                             └── , ──┬── scope_reference ──┬──┘
                                                     └── array_index ──────┘
```

1101A956

Method-style syntax:

```
── array_expr.ARRAY_UPDATE ── ( new_value ──┬──────────────────────────┬──── ) ──
                                            └── , ──┬── scope_reference ──┬──┘
                                                    └── array_index ──────┘
```

1101A967

where:

Syntax element...	Specifies...
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
new_value	an expression that evaluates to a value with the same data type as the element type of *array_expr*.
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.
array_index	an optional index to the array, which is one of the following: • An integer value. • An array instance of the predefined array type ArrayVec with a comma-separated list of integer values to define the bounds. For details, see "ARRAY Scope Reference" on page 200. The value of *array_index* must be within the limits of *array_expr*.

ANSI Compliance

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

Usage Notes

ARRAY_UPDATE takes an array expression as an argument and updates all or a subset of the elements in *array_expr* with the value specified by *new_value*.

The following rules apply to the ARRAY_UPDATE function:

- If *array_expr* is NULL, then the result value is NULL.
- The value of *new_value* must evaluate to the same data type as the element type of *array_expr*, or it must be a data type that can be implicitly converted to the ARRAY element type.
- If you do not specify *scope_reference* or *array_index*, then all the elements in *array_expr* are updated to the value of *new_value*.
- If you specify *scope_reference*, then only the elements within the scope reference of *array_expr* are updated to the value of *new_value*. *scope_reference* must refer to consecutive elements in the array.
- If you specify *array_index*, then only the single element specified by *array_index* is updated to the value of *new_value* in *array_expr*.
- If an element to be updated contains a NULL, the NULL is replaced with the value of *new_value* in *array_expr*.
- When one or more elements are modified, and there are uninitialized elements in the ARRAY value prior to the elements to be updated, then ARRAY_UPDATE sets any preceding uninitialized elements to NULL. This behavior is the same as that of a regular SQL UPDATE statement that sets an ARRAY element value.

Result Type and Value

ARRAY_UPDATE returns a new copy of the array specified by *array_expr* with the specified elements updated to the new value.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS CHAR(10) ARRAY[20];
```

```
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The following query returns an updated copy of the phonelist array, with all the elements updated to the new value:

```
SELECT ARRAY_UPDATE(phonelist, '9095551234')
FROM employee_info;
```

The following query returns an updated copy of the phonelist array, with a subset of the elements updated to the new value, as specified by the scope reference. The result is that

elements 2, 3, and 4 are updated to the new value. The rest of the elements in the array retain their original values.

```
SELECT ARRAY_UPDATE(phonelist, '9095551234', 2, 4)
FROM employee_info;
```

The following is the same query using method-style syntax.

```
SELECT phonelist.ARRAY_UPDATE('9095551234', 2, 4)
FROM employee_info;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

The following query returns an updated copy of the shot1 array, with all the elements updated to the new value.

```
SELECT ARRAY_UPDATE(shot1, 0)
FROM seismic_data;
```

The following query returns an updated copy of the shot1 array, with a subset of the elements updated to the new value, as specified by the scope reference. The result is that the elements in the scope reference range [5:10][5:10] are updated to the new value. The rest of the elements in the array retain their original values.

```
SELECT ARRAY_UPDATE(shot1, 0, NEW arrayVec(5,5), NEW arrayVec(10,10))
FROM seismic_data;
```

ARRAY_UPDATE_STRIDE

Purpose

Updates all or a subset of the elements in *array_expr* to the specified new value. The *stride* argument indicates how many elements should be skipped between each updated element.

Syntax

System function syntax:

```
── ARRAY_UPDATE_STRIDE ── ( array_expr, new_value, stride ──┬──────────────────────┬── ) ──
                                                          └── , ──┬─ scope_reference ─┬──┘
                                                                  └─ array_index ─────┘
                                                                                  1101A954
```

Method-style syntax:

```
── array_expr.ARRAY_UPDATE_STRIDE ── ( new_value, stride ──┬──────────────────────┬── ) ──
                                                         └── , ──┬─ scope_reference ─┬──┘
                                                                 └─ array_index ─────┘
                                                                                 1101A955
```

where:

Syntax element...	Specifies...
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
new_value	an expression that evaluates to a value with the same data type as the element type of *array_expr*.
stride	an unsigned integer value indicating the number of elements that should be skipped between each updated element.
scope_reference	an optional ARRAY scope reference. See "ARRAY Scope Reference" on page 200.

Syntax element...	Specifies...
array_index	an optional index to the array, which is one of the following: • An integer value. • An array instance of the predefined array type ArrayVec with a comma-separated list of integer values to define the bounds. For details, see "ARRAY Scope Reference" on page 200. The value of *array_index* must be within the limits of *array_expr*.

ANSI Compliance

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

Usage Notes

ARRAY_UPDATE_STRIDE takes an array expression as an argument and updates all or a subset of the elements in *array_expr* with the value specified by *new_value*. The *stride* argument indicates how many elements should be skipped between each updated element.

The following rules apply to the ARRAY_UPDATE_STRIDE function:

- If *array_expr* is NULL, then the result value is NULL.
- The value of *new_value* must evaluate to the same data type as the element type of *array_expr*, or it must be a data type that can be implicitly converted to the ARRAY element type.
- If you do not specify *scope_reference* or *array_index*, then all the elements in *array_expr* are updated to the value of *new_value*.
- If you specify *scope_reference*, then only the elements within the scope reference of *array_expr* are updated to the value of *new_value*. *scope_reference* must refer to consecutive elements in the array.
- If you specify *array_index*, then only the single element specified by *array_index* is updated to the value of *new_value* in *array_expr*. In this case, *stride_value* has no effect.
- The *stride_value* argument specifies the number of elements to skip between each element to be updated. For example, if *stride_value* is 1, then every other element within the affected range of elements is updated. The affected range of elements depends on whether or not *scope_reference* is specified. For a multidimensional array, the elements are stored in row-major order so they are traversed and skipped according to *stride_value* in this order.
- If an element to be updated contains a NULL, the NULL is replaced with the value of *new_value* in *array_expr*.
- When one or more elements are modified, and there are uninitialized elements in the ARRAY value prior to the elements to be updated, then ARRAY_UPDATE_STRIDE sets any preceding uninitialized elements to NULL. This behavior is the same as that of a regular SQL UPDATE statement that sets an ARRAY element value.
- If an element is skipped that was previously uninitialized, it is updated to NULL.

Result Type and Value

ARRAY_UPDATE_STRIDE returns a new copy of the array specified by *array_expr* with the specified elements updated to the new value.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS CHAR(10) ARRAY[20];

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The following query returns an updated copy of the phonelist array with every other element updated to the new value:

```
SELECT ARRAY_UPDATE_STRIDE(phonelist, '9095551234', 1)
FROM employee_info;
```

The following query returns an updated copy of the phonelist array, with a subset of the elements updated to the new value, as specified by the scope reference and stride value. The result is that elements 2 and 4 are updated to the new value. The rest of the elements in the array retain their original values.

```
SELECT ARRAY_UPDATE_STRIDE(phonelist, '9095551234', 1, 2, 4)
FROM employee_info;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS INTEGER ARRAY[1:50][1:50];

CREATE TABLE seismic_data (
   id INTEGER,
   shot1 shot_ary,
   shot2 shot_ary);
```

The following query returns an updated copy of the shot1 array, with every other element updated to the new value.

```
SELECT ARRAY_UPDATE_STRIDE(shot1, 0, 1)
FROM seismic_data;
```

The following query returns an updated copy of the shot1 array, with a subset of the elements updated to the new value, as specified by the scope reference and stride value. The result is that every 5th element beginning with the first element in the scope reference range [1:50][1:50] is updated to the new value. The rest of the elements in the array retain their original values.

```
SELECT ARRAY_UPDATE_STRIDE(shot1, 0, 5, NEW arrayVec(1,1),
   NEW arrayVec(50,50))
FROM seismic_data;
```

The following is the same query using method-style syntax.

```
SELECT shot1.ARRAY_UPDATE_STRIDE (0, 5, NEW arrayVec(1,1),
   NEW arrayVec(50,50))
FROM seismic_data;
```

OEXISTS

Purpose

Returns an integer value indicating whether the element specified by the index in *array_expr* contains a value or is in an uninitialized state.

Syntax

System function syntax:

```
┬──────────────────┬─ OEXISTS ── ( array_expr, ┬─ index_value ─┬ ) ──────────────────
└─ TD_SYSFNLIB. ──┘                           └─ array_bound ─┘
```

1101A945

Method-style syntax:

```
──── array_expr.OEXISTS ── ( ┬─ index_value ─┬ ) ──────────────────────────────────
                            └─ array_bound ─┘
```

1101A966

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
index_value	an unsigned integer value.
array_bound	an array instance of the predefined array type ArrayVec with a comma-separated list of integer values to define the bounds. For details, see "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Usage Notes

OEXISTS takes an array expression as an argument and checks the element specified by the index to see if it contains a value or is in an uninitialized state. OEXISTS returns 1 if the specified element contains a non-NULL or NULL. If the element is in an uninitialized state, the method returns 0.

If the value of *index_value* or *array_bound* is out of bounds for *array_expr*, OEXISTS returns 0. If either *index_value* or *array_bound* is NULL, a 0 value is returned. If *array_expr* is NULL, then an error stating that the function does not exists is returned.

If *array_expr* refers to an empty array with all elements in an uninitialized state, then OEXISTS returns 0. The OEXISTS method with an integer *index_value* argument is compatible with the Oracle EXISTS method for one-dimensional ARRAY types.

Result Type and Value

OEXISTS returns an integer value of 1 or 0.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());

/* Update the empty ARRAY instance such that element 3 is set to a value;
then elements 1 and 2 are set to NULL, the rest are uninitialized. */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query checks to see whether element 2 of the phonelist array contains a value.

```
SELECT eno, phonelist.OEXISTS(2)
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OEXISTS(2)
---         --------------------
  1           1           (element 2 contains a non-NULL value)
  2           1           (element 2 contains a NULL)
```

The following query checks to see whether element 3 of the phonelist array contains a value.

```
SELECT eno, phonelist.OEXISTS(3)
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OEXISTS(3)
---         --------------------
  1           0           (element 3 is in an uninitialized state)
  2           1           (element 3 contains a non-NULL value)
```

The following is the same query using function-style syntax.

```
SELECT eno, OEXISTS(phonelist, 3)
FROM employee_info;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));

/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());

/* Update the empty ARRAY instance such that element [1][3] is set to a
value; then elements [1][1] and [1][2] are set to NULL, the rest are
uninitialized */

UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query checks to see whether element [1][2] of the shots array contains a value.

```
SELECT id, shots.OEXISTS(NEW arrayVec(1, 2))
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OEXISTS(new arrayVec(1, 2))
--          ---------------------------------
 1           1           (element [1][2] contains a non-NULL value)
 2           1           (element [1][2] contains a NULL)
```

The following query checks to see whether element [1][3] of the shots array contains a value.

```
SELECT id, shots.OEXISTS(NEW arrayVec(1, 3))
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OEXISTS(NEW arrayVec(1, 3))
--          ---------------------------------
 1           0           (element [1][3] is in an uninitialized state)
 2           1           (element [1][3] contains a non-NULL value)
```

OCOUNT

Purpose

Returns the number of elements in *array_expr* that contain either a NULL or non-NULL value.

Syntax

System function syntax:

```
┬──────────────────┬── OCOUNT ── ( array_expr ) ──────────────────────────────
└── TD_SYSFNLIB. ──┘
```

1101A946

Method-style syntax:

```
── array_expr.OCOUNT ── (  ) ──
```

1101A965

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression

ANSI Compliance

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

Usage Notes

OCOUNT takes an array expression as an argument and returns the number of elements that contain a non-NULL value or NULL. Elements that are in an uninitialized state are not counted. If the array is empty (all elements of the array are in an uninitialized state), then OCOUNT returns a value of 0.

If *array_expr* is NULL, an error is returned.

The OCOUNT method is compatible with the Oracle COUNT method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

Result Type

OCOUNT returns an unsigned integer value.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());

/* Update the empty ARRAY instance such that element 3 is set to a value;
then elements 1 and 2 are set to NULL, the rest are uninitialized. */
UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query returns the number of elements in the phonelist array that contain NULL or non-NULL values.

```
SELECT eno, phonelist.OCOUNT()
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OCOUNT()
---         ------------------
 1           2      (the first 2 elements have non-NULL values)
 2           3      (the first 3 elements have NULL or non-NULL values)
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
```

```
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());

/* Update the empty ARRAY instance such that element [1][3] is set to a
value; then elements [1][1] and [1][2] are set to NULL, the rest are
uninitialized */

UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query returns the number of elements in the shots array that contain NULL or non-NULL values.

```
SELECT id, shots.OCOUNT()
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OCOUNT()
--          --------------
 1           2    (the first 2 elements contain non-NULL values)
 2           3    (the first 3 elements contain NULL or non-NULL values)
```

The following is the same query using function-style syntax.

```
SELECT id, OCOUNT(shots)
FROM seismic_table;
```

OLIMIT

Purpose

Returns the highest possible subscript value in *array_expr* as either an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

Syntax

System function syntax:

```
┌──────────────────┐
│                  │
└── TD_SYSFNLIB. ──┘── OLIMIT ── ( array_expr ) ──────────────────────────────
```

1101A947

Method-style syntax:

```
── array_expr.OLIMIT ── (  ) ──
```

1101A964

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression

ANSI Compliance

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

Usage Notes

OLIMIT takes an array expression as an argument and returns the highest possible subscript value in the ARRAY type. If *array_expr* is a one-dimensional ARRAY type, OLIMIT returns an unsigned INTEGER value. If *array_expr* is a multidimensional ARRAY type, OLIMIT returns a new instance of the predefined ARRAY type ArrayVec, containing the subscript information.

If *array_expr* is NULL, an error is returned.

The OLIMIT method is compatible with the Oracle LIMIT method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

Result Type

OLIMIT returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());
```

The following query returns the highest possible subscript value in the phonelist array.

```
SELECT eno, phonelist.OLIMIT()
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OLIMIT()
---         ------------------
 1           20
 2           20
```

The following is the same query using function-style syntax.

```
SELECT eno, OLIMIT(phonelist)
FROM employee_info;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));

/* Empty ARRAY instance */
```

```
INSERT INTO seismic_table VALUES (2, shot_ary());
```

The following query returns the highest possible subscript value in the shots array.

```
SELECT id, shots.OLIMIT()
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OLIMIT()
--          --------------
 1           NEW arrayVec(50,50)
 2           NEW arrayVec(50,50)
```

OFIRST

Purpose

Returns the lowest subscript value in *array_expr* as either an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

Syntax

System function syntax:

```
┬──────────────────┬─ OFIRST ── ( array_expr ) ──────────────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A948

Method-style syntax:

```
── array_expr.OFIRST ── (  ) ──
```

1101A963

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression

ANSI Compliance

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

Usage Notes

OFIRST takes an array expression as an argument and returns the lowest subscript value in the ARRAY type. If *array_expr* is a one-dimensional ARRAY type, OFIRST returns an unsigned INTEGER value. If *array_expr* is a multidimensional ARRAY type, OFIRST returns a new instance of the predefined ARRAY type ArrayVec, containing the subscript information for the

first element in the array. If the array is empty (all elements of the array are in an uninitialized state), then OFIRST returns NULL.

If *array_expr* is NULL, an error is returned.

The OFIRST method is compatible with the Oracle FIRST method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

Result Type

OFIRST returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());
```

The following query returns the lowest subscript value in the phonelist array.

```
SELECT eno, phonelist.OFIRST()
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OFIRST()
---         ------------------
 1           1
 2           1
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));
```

```
/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());
```

The following query returns the lowest subscript value in the shots array.

```
SELECT id, shots.OFIRST()
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OFIRST()
--          --------------
 1           NEW arrayVec(1,1)
 2           NEW arrayVec(1,1)
```

The following is the same query using function-style syntax.

```
SELECT id, OFIRST(shots)
FROM seismic_table;
```

OLAST

Purpose

Returns the highest subscript value with a populated element in *array_expr*.

Syntax

System function syntax:

```
┬──────────────────┬─ OLAST ── ( array_expr ) ──────────────────────────────
└─ TD_SYSFNLIB. ──┘
```

1101A949

Method-style syntax:

```
── array_expr.OLAST ── ( ) ──
```

1101A962

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression

ANSI Compliance

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

Usage Notes

OLAST takes an array expression as an argument and returns the highest subscript value with a populated element in the ARRAY type. This is different from the highest possible subscript value, as returned by the OLIMIT method. If *array_expr* is a one-dimensional ARRAY type, OLAST returns an unsigned INTEGER value. If *array_expr* is a multidimensional ARRAY type, OLAST returns a new instance of the predefined ARRAY type ArrayVec, containing the

subscript information. If the array is empty (all elements of the array are in an uninitialized state), then OLAST returns NULL.

If *array_expr* is NULL, an error is returned.

The OLAST method is compatible with the Oracle LAST method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

Result Type

OLAST returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());

/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */

UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query returns the highest subscript value with a populated element in the phonelist array.

```
SELECT eno, phonelist.OLAST()
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OLAST()
---         -----------------
 1           2
 2           3
```

The following is the same query using function-style syntax.

```
SELECT eno, OLAST(phonelist)
FROM employee_info;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));

/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());

/* Update empty ARRAY instance such that element [1][3] is set to a
value; Then elements [1][1] and [1][2] are set to NULL, the rest are
uninitialized */

UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query returns the highest subscript value with a populated element in the shots array.

```
SELECT id, shots.OLAST()
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OLAST()
--          ------------------
 1           NEW arrayVec(1,2)
 2           NEW arrayVec(1,3)
```

OPRIOR

Purpose

Returns the subscript of the element prior to the element specified for *array_expr*, if that element is currently populated.

Syntax

System function syntax:

```
┬──────────────────┬─ OPRIOR ── ( array_expr, ┬─ integer_value ─┬ ) ──────────────────
└─ TD_SYSFNLIB. ──┘                          └─ array_bound ────┘
```

1101A950

Method-style syntax:

```
──── array_expr.OPRIOR ── ( ┬─ integer_value ─┬ ) ──────────────────────────────
                            └─ array_bound ────┘
```

1101A961

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
integer_value	an unsigned INTEGER value.
array_bound	an array instance of the predefined array type ArrayVec with a comma-separated list of integer values to define the bounds. For details, see "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Usage Notes

OPRIOR takes an array expression as an argument and returns the subscript of the element prior to the element specified by *integer_value* or *array_bound*, if that element is currently populated. If you specify an element subscript that is less than or equal to the value of OFIRST for the array, a NULL is returned. Also, if *integer_value* or *array_bound* is NULL, an error is returned.

If *array_expr* is a one-dimensional ARRAY type, OPRIOR returns an unsigned INTEGER value. If *array_expr* is a multidimensional ARRAY type, OPRIOR returns a new instance of the predefined ARRAY type ArrayVec, containing the subscript information. If the array is empty (all elements of the array are in an uninitialized state), then OPRIOR returns NULL.

If *array_expr* is NULL, an error is returned.

The OPRIOR method with one INTEGER argument is compatible with the Oracle PRIOR method for one-dimensional ARRAY types.

Result Type

OPRIOR returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());

/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */

UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query returns the subscript of the element prior to element 2 in the phonelist array.

```
SELECT eno, phonelist.OPRIOR(2)
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OPRIOR(2)
---         -------------------
```

```
 1           1
 2           1
```

The following is the same query using function-style syntax.

```
SELECT eno, OPRIOR(phonelist, 2)
FROM employee_info;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));

/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());

/* Update empty ARRAY instance such that element [1][3] is set to a
value; Then elements [1][1] and [1][2] are set to NULL, the rest are
uninitialized */

UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query returns the subscript of the element prior to element [1][2] in the shots array.

```
SELECT id, shots.OPRIOR(NEW arrayVec(1, 2))
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OPRIOR(NEW arrayVec(1, 2))
--          --------------------------------
 1           NEW arrayVec(1,1)
 2           NEW arrayVec(1,1)
```

ONEXT

Purpose

Returns the subscript of the element subsequent to the element specified for *array_expr*, if that element is currently populated.

Syntax

System function syntax:

```
┬──────────────────┬─ ONEXT ─ ( array_expr, ┬─ integer_value ─┬ ) ──────────────────
└─ TD_SYSFNLIB. ─┘                         └─ array_bound ───┘
```

1101A951

Method-style syntax:

```
──── array_expr.ONEXT ─ ( ┬─ integer_value ─┬ ) ──────────────────────────────────
                          └─ array_bound ───┘
```

1101A960

where:

Syntax element…	Specifies…
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
integer_value	an unsigned INTEGER value.
array_bound	an array instance of the predefined array type ArrayVec with a comma-separated list of integer values to define the bounds. For details, see "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Usage Notes

ONEXT takes an array expression as an argument and returns the subscript of the element subsequent to the element specified by *integer_value* or *array_bound*, if that element is currently populated. If you specify an element subscript that is greater than or equal to the value of OLAST for the array, a NULL is returned. Also, if *integer_value* or *array_bound* is NULL, an error is returned.

If *array_expr* is a one-dimensional ARRAY type, ONEXT returns an unsigned INTEGER value. If *array_expr* is a multidimensional ARRAY type, ONEXT returns a new instance of the predefined ARRAY type ArrayVec, containing the subscript information. If the array is empty (all elements of the array are in an uninitialized state), then ONEXT returns NULL.

If *array_expr* is NULL, an error is returned.

The ONEXT method with one INTEGER argument is compatible with the Oracle NEXT method for one-dimensional ARRAY types.

Result Type

ONEXT returns an unsigned INTEGER value or a new instance of the predefined ARRAY type ArrayVec.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());

/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */

UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query returns the subscript of the element subsequent to element 2 in the phonelist array.

```
SELECT eno, phonelist.ONEXT(2)
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.ONEXT(2)
---         ------------------
```

```
 1           ?         (the element following element 2 is unpopulated)
 2           3
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));

/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());

/* Update empty ARRAY instance such that element [1][3] is set to a
value; Then elements [1][1] and [1][2] are set to NULL, the rest are
uninitialized */

UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query returns the subscript of the element subsequent to element [1][2] in the shots array.

```
SELECT id, shots.ONEXT(NEW arrayVec(1, 2))
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.ONEXT(NEW arrayVec(1, 2))
--          -------------------------------
 1           ?     (The element following element [1][2] is unpopulated)
 2           NEW arrayVec(1,3)
```

The following is the same query using function-style syntax.

```
SELECT id, ONEXT(shots, NEW arrayVec(1, 2))
FROM seismic_table;
```

OEXTEND

Purpose

Allocates space for one or more new elements in *array_expr*.

Syntax

System function syntax:

Method-style syntax:

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
num_spaces	an optional unsigned integer value representing the number of NULL elements to append to *array_expr*.
index_value	an optional unsigned integer value used as an index to an element in *array_expr*.
array_bound	an optional array instance of the predefined array type ArrayVec with a comma-separated list of integer values to define the bounds. For details, see "ARRAY Scope Reference" on page 200.

ANSI Compliance

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

Usage Notes

OEXTEND takes an array expression as an argument and allocates space for one or more new elements in the array.

IF you specify...	THEN the method...
OEXTEND()	appends a single NULL element to the ARRAY value. For a multidimensional (n-D) ARRAY, this is done in row-major order.
OEXTEND(*num_spaces*)	appends *n* NULL elements to the ARRAY value, where *n* = *num_spaces*. For an n-D ARRAY, this is done in row-major order.
OEXTEND(*num_spaces*, *index_value*) or OEXTEND(*num_spaces*, *array_bound*)	appends *n* copies of the element indexed by either *index_value* or *array_bound* to the ARRAY value, where *n* = *num_spaces*. For an n-D ARRAY, this is done in row-major order.

The OEXTEND method with zero, one, or two arguments of INTEGER type is compatible with the Oracle EXTEND method, for one-dimensional ARRAY types. However, the empty set of parentheses required when OEXTEND is called with no arguments is a deviation from Oracle syntax.

OEXTEND returns an error in the following cases:

- If *array_expr* is NULL.
- If either *index _value* or *array_bound* is NULL.
- If the number of elements appended by OEXTEND overflows the maximum size of the array.

If *num_spaces* is NULL, no error occurs, but no action is taken by OEXTEND.

Result Type

OEXTEND returns a new modified copy of the array argument.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);
```

```
CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));
```

```
/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());

/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */

UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query extends the phonelist array with a single NULL element.

```
SELECT eno, phonelist.OEXTEND()
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OEXTEND()
---         -------------------
 1           ('1112223333', '6195551234',NULL)
 2           (NULL,NULL,'8584850000',NULL)
```

The following query extends the phonelist array with three NULL elements.

```
SELECT eno, phonelist.OEXTEND(3)
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OEXTEND(3)
---         --------------------
 1           ('1112223333', '6195551234',NULL,NULL,NULL)
 2           (NULL,NULL,'8584850000',NULL,NULL,NULL)
```

The following query extends the phonelist array with two copies of element 1.

```
SELECT eno, phonelist.OEXTEND(2,1)
FROM employee_info
WHERE eno = 1;
```

The following is the result of the query.

```
ENO         phonelist.OEXTEND(2,1)
---         ----------------------
 1           ('1112223333', '6195551234', '1112223333', '1112223333')
```

The following is the same query using function-style syntax.

```
SELECT eno, OEXTEND(phonelist,2,1)
FROM employee_info
WHERE eno = 1;
```

Example 2

The following example shows how you can use the OEXTEND method to fill the end of a constructed ARRAY with NULL elements so that these elements are no longer in an uninitialized state. This can be helpful when the ARRAY may be used as an argument to another system function or operator.

```
CREATE TYPE myarray AS VARRAY(5) OF INTEGER;
```

```
CREATE TABLE mytab (id INTEGER, ary myarray);

/* Populate the first 3 elements. The last 2 elements are uninitialized.
*/
INSERT INTO mytab VALUES (1, NEW myarray(1,2,3));

/* Fill the last 2 elements with NULLs. */
UPDATE mytab
SET ary = ary.OEXTEND(2)
WHERE id = 1;

SELECT ary FROM mytab;
```

The following is the result of the query.

```
(1,2,3,NULL,NULL)
```

Example 3

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));

/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());

/* Update empty ARRAY instance such that element [1][3] is set to a
value; Then elements [1][1] and [1][2] are set to NULL, the rest are
uninitialized */

UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query extends the shots array with a single NULL element.

```
SELECT id, shots.OEXTEND()
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OEXTEND()
--          ---------------
 1           (11,12,NULL)
 2           (NULL,NULL,1133,NULL)
```

The following query extends the shots array with three NULL elements.

```
SELECT id, shots.OEXTEND(3)
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OEXTEND(3)
--          ----------------
 1           (11,12,NULL,NULL,NULL)
 2           (NULL,NULL,1133,NULL,NULL,NULL)
```

The following query extends the shots array with two copies of element [1][1].

```
SELECT id, shots.OEXTEND(2, NEW arrayVec(1,1))
FROM seismic_table
WHERE id = 1;
```

The following is the result of the query.

```
ID          shots.OEXTEND(2, NEW arrayVec(1,1))
--          -----------------------------------
 1           (11,12,11,11)
```

OTRIM

Purpose

Removes one or more populated elements from the end of *array_expr*.

Syntax

System function syntax:

```
┌──────────────────┐
│                  │
└─ TD_SYSFNLIB. ───┘── OTRIM ── ( array_expr ┌──────────────────┐ ) ──────────────────
                                             │                  │
                                             └─ , integer_value ─┘
```

1101A952

Method-style syntax:

```
── array_expr.OTRIM ── ( ┌──────────────┐ ) ──────────────────────────────────────
                         │              │
                         └─ integer_value ─┘
```

1101A958

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression
integer_value	an optional unsigned INTEGER value indicating the number of populated elements to remove from the array. The default value is 1.

ANSI Compliance

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

Usage Notes

OTRIM takes an array expression as an argument and removes one or more populated elements from the end of the array value. If you do not specify *integer_value*, then OTRIM removes a single populated element from the end of the array value; otherwise, OTRIM

removes the number of populated elements specified by *integer_value* from the end of the array value. For a multidimensional ARRAY type, this is done in row-major order. OTRIM returns a new modified copy of the array value as the result.

If *array_expr* is NULL, an error is returned. If *integer_value* is NULL, no error occurs, but no action is taken by OTRIM. If the number of elements to be removed by OTRIM is greater than the current value of OCOUNT for the array, then an error is returned.

The OTRIM method is compatible with the Oracle TRIM method for one-dimensional ARRAY types. If *integer_value* is not specified, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

Result Type

OTRIM returns a new modified copy of the array argument.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());

/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */

UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query removes the last populated element from the phonelist array.

```
SELECT eno, phonelist.OTRIM()
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OTRIM()
---         -----------------
 1           ('1112223333')
 2           (NULL,NULL)
```

The following query removes the last two populated elements from the phonelist array.

```
SELECT eno, phonelist.OTRIM(2)
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.OTRIM(2)
---         ------------------
 1           ()       (the only 2 populated elements were removed so we
                       now have an empty array value)
 2           (NULL)
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));

/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());

/* Update empty ARRAY instance such that element [1][3] is set to a
value; Then elements [1][1] and [1][2] are set to NULL, the rest are
uninitialized */

UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query removes the last populated element from the shots array.

```
SELECT id, shots.OTRIM()
FROM seismic_table
WHERE id = 1;
```

The following is the result of the query.

```
ID          shots.OTRIM()
--          -------------
 1           (11)
```

The following query removes the last two populated elements from the shots array.

```
SELECT id, shots.OTRIM(2)
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.OTRIM(2)
--          --------------
 1           ()   (the only 2 populated elements were removed so we now
                  have an empty array value)
 2           (NULL)
```

The following is the same query using function-style syntax.

```
SELECT id, OTRIM(shots, 2)
FROM seismic_table;
```

ODELETE

Purpose

Removes all elements from *array_expr*.

Syntax

System function syntax:

```
┬──────────────────┬─ ODELETE ── ( array_expr ) ──────────────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A944

Method-style syntax:

```
── array_expr.ODELETE ── (  ) ──
```

1101A957

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the method is located.
array_expr	an ARRAY expression, which is one of the following: • A reference to an ARRAY column • An ARRAY constructor expression. See "ARRAY Constructor Expression" on page 197. • A UDF expression • A UDM expression

ANSI Compliance

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

Usage Notes

ODELETE takes an array expression as an argument and removes all populated elements from the array value. ODELETE returns a new modified copy of the array value as the result.

If *array_expr* is NULL, an error is returned.

The ODELETE method is compatible with the Oracle DELETE method for one-dimensional ARRAY types. However, the empty set of parentheses required by Teradata syntax is a deviation from Oracle syntax.

Result Type

ODELETE returns a new modified copy of the array argument.

Example 1

Consider the following one-dimensional ARRAY data type and table.

```
CREATE TYPE phonenumbers AS VARRAY(20) OF CHAR(10);

CREATE TABLE employee_info (eno INTEGER, phonelist phonenumbers);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO employee_info VALUES (1,
   phonenumbers('1112223333', '6195551234'));

/* Empty ARRAY instance */
INSERT INTO employee_info VALUES (2,
   phonenumbers());

/* Update empty ARRAY instance such that element 3 is set to a value;
   Then elements 1 and 2 are set to NULL, the rest are uninitialized */

UPDATE employee_info
SET phonelist[3] = '8584850000'
WHERE id = 2;
```

The following query removes all populated elements from the phonelist array, leaving all of the elements in an uninitialized state.

```
SELECT eno, phonelist.ODELETE()
FROM employee_info;
```

The following is the result of the query.

```
ENO         phonelist.ODELETE()
---         -------------------
 1           ()
 2           ()
```

The following is the same query using function-style syntax.

```
SELECT eno, ODELETE(phonelist)
FROM employee_info;
```

Example 2

Consider the following 2-D ARRAY data type and table.

```
CREATE TYPE shot_ary AS VARRAY(1:50)(1:50) OF INTEGER;

CREATE TABLE seismic_table (
   id INTEGER,
```

```
   shots shot_ary);
```

The table is populated with the following values:

```
/* The first 2 elements are populated; the rest are uninitialized. */
INSERT INTO seismic_table VALUES (1, shot_ary(11, 12));

/* Empty ARRAY instance */
INSERT INTO seismic_table VALUES (2, shot_ary());

/* Update empty ARRAY instance such that element [1][3] is set to a
value; Then elements [1][1] and [1][2] are set to NULL, the rest are
uninitialized */

UPDATE seismic_table
SET shots[1][3] = 1133
WHERE id = 2;
```

The following query removes all populated elements from the shots array, leaving all elements of the array in an uninitialized state.

```
SELECT id, shots.ODELETE()
FROM seismic_table;
```

The following is the result of the query.

```
ID          shots.ODELETE()
--          ---------------
 1           ()
 2           ()
```

CHAPTER 5 Attribute Functions

This chapter describes SQL attribute functions.

Attribute Function Overview

Attribute functions return descriptive information about their operand. Except for the DEFAULT function, the operand need not be a column reference; it can be a general expression that is not evaluated mathematically.

When an attribute function is used in a request, the response returns one row for every data row that meets the search condition.

Some of these functions are extensions to ANSI SQL.

For a list of data type attributes, see "Data Type Phrases" in *SQL Data Types and Literals*.

Each attribute function is described individually in the following topics.

ANSI Equivalence of Teradata Attribute Functions

Several of the Teradata attribute functions are extensions to the ANSI SQL:2011 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

```
┬─ BYTE ─┬─ ( byte_expression ) ──
└─ BYTES ┘
```

1101F174

where:

Syntax element ...	Specifies ...
byte_expression	the byte string for which the number of bytes is to be returned.

ANSI Compliance

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

Argument Types

The data types of *byte_expression* are restricted to the following:

- BYTE, VARBYTE and BLOB
- UDT that has an implicit cast to a predefined byte type

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

  For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Length Includes Trailing Zeros

Because trailing double zero bytes are considered bytes, the length of the value in a *fixed* length column is always equal to the length defined for the column.

The length of the value in a *variable* length column is always equal to the number of bytes, including any trailing double zero bytes, contained in that value.

If you do not want trailing blanks included in the byte count for a data value, use the TRIM function on the argument to BYTES. For example:

```
SELECT BYTES( TRIM( TRAILING FROM byte_col ) ) FROM table1;
```

For more information on TRIM, see "TRIM" on page 1248.

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

```
— CHARACTER_LENGTH ┬ (string_expression) ———
— CHAR_LENGTH ———┘
```

FF07D088

where:

Syntax element ...	Specifies ...
string_expression	the string expression for which the length is to be returned.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Usage Notes

CHARACTER_LENGTH is the ANSI form of the Teradata CHARACTERS function. Use CHARACTER_LENGTH instead of CHARACTERS for ANSI SQL:2011 conformance.

Use CHARACTER_LENGTH in place of MCHARACTERS. (MCHARACTERS no longer appears in this book because its use is deprecated and it will not be supported after support for KANJI1 is dropped.)

Argument Types

The type of *string_expression* must be CHARACTER, VARCHAR, or CLOB. For non-character data types, the function returns an error.

By default, Teradata Database performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and a predefined character type.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Result

For all server character sets except KANJI1, CHARACTER_LENGTH returns the length of *string_expression* in characters.

For KANJI1, the following results are obtained.

FOR this client character set ...	CHARACTER_LENGTH returns ...
KanjiEBCDIC	the length of *string_expression* as the number of bytes. A mix of single and multibyte characters is expected. If any Shift-Out/Shift-In characters are present, they are included in the result count.
KanjiEUC KanjiShift-JIS	the length of *string_expression* as the number of logical characters, based on the client session character set. A mix of single and multibyte characters is expected.
ASCII EBCDIC	the length of *string_expression* as the number of bytes.

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see "KANJI1 Character Set" in *International Character Set Support*.

Because trailing pad characters are considered characters, the length of the value in a CHARACTER column is always equal to the length defined for the column.

The length of the value in a VARCHAR or CLOB column is always equal to the number of characters, including any trailing pad characters, contained in that value.

Suppressing Trailing Pad Characters

To suppress trailing pad characters from the character count for a data value, use the TRIM function on the argument to CHARACTER_LENGTH. For example:

```
SELECT CHARACTER_LENGTH( TRIM( TRAILING FROM Name ) )
FROM Employee;
```

Example

The following statement applies the CHARACTER_LENGTH function to the Name column, which is type VARCHAR(30) CHARACTER SET LATIN, to obtain the number of characters in each employee name:

```
SELECT Name, CHARACTER_LENGTH(Name)
FROM Employee;
```

The result is as follows (note that separator blanks are considered characters):

```
Name      Character_Length(Name)
--------  ----------------------
Smith T            7
Newman P           8
Omura H            7
      .            .
```

Example Set 1: KanjiEBCDIC

FOR this server character set ...	AND example ...	CHARACTER_LENGTH returns ...
GRAPHIC	ABC	3
KANJI1	De<**MNP**>	10
	<><>	4

Example Set 2: KanjiShift-JIS

FOR this server character set ...	AND example ...	CHARACTER_LENGTH returns ...
KANJI1	<><>	10
	DeF	3
UNICODE	ABC	3
GRAPHIC	ABC	3

Example Set 3: KanjiEUC

FOR this server character set ...	AND example ...	CHARACTER_LENGTH returns ...
KANJI1	ss_3**C**ss_3**D**	2
GRAPHIC		2
UNICODE	<><>	0
	d**A** ss_2**B** ss_3**E**	4
LATIN	ABC	3

CHARACTERS

Purpose

Returns an integer value representing the number of logical characters or bytes contained in the specified operand string.

Syntax

```
— CHARACTERS —┬( string_expression ) —
— CHARS ———— │
— CHAR ———— ┘
```

1101A488

where:

Syntax element ...	Specifies ...
string_expression	a character (single byte, multibyte, mixed single byte and multibyte) string for which the number of characters is to be returned. The data types for *string_expression* are restricted to CHARACTER, VARCHAR, and CLOB.

ANSI Compliance

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

Value Returned by CHARACTERS and Server Character Set

Because CHARACTERS returns the number of logical characters or bytes in *string_expression*, the value differs depending on the server character set of *string_expression*. The following table illustrates the differences among the various character sets for a CHARACTER(12) column.

FOR this server character set ...	The length of *string_expression* ...
• UNICODE • LATIN • GRAPHIC	is always 12. Unicode, Latin, and Graphic are fixed width character types.
• KANJISJIS • KANJI1	varies depending on the mix of characters (multibyte and single byte) in the string. KanjiSJIS and KANJI1 are variable width character sets.

CHARACTER_LENGTH versus CHARACTERS

Use of the CHARACTERS function is deprecated. Instead, use the ANSI-equivalent "CHARACTER_LENGTH."

DEFAULT

Purpose

Returns the current default value for the specified or derived column.

Syntax

```
── DEFAULT ──┬──────────────────┬──
          └─ ( column_name ) ─┘
```

1101A394

where:

Syntax element ...	Specifies ...
column_name	the name of a column in a base table, view, queue table, or derived table. The column name can be qualified or unqualified.

ANSI Compliance

DEFAULT is partially ANSI SQL:2011 compliant.

The form of DEFAULT that specifies a column name is a Teradata extension. Using DEFAULT in a predicate is also a Teradata extension.

Result Type and Attributes

The result type, format, and title for DEFAULT(*x*) appear in the following table.

Data Type	Format	Title
Data type of the specified column	Format of the specified column	Default(x)

For information on data type default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Result Value

The DEFAULT function returns the default value of the specified column or derived column (if the column name is omitted).

If the specified or derived column is a view column or derived table column, the DEFAULT function returns the default value of the underlying table column.

If the default value of a column evaluates to a system variable, for example when the default value is CURRENT_TIME or USER, the DEFAULT function returns the value of the system variable at the time the statement is executed.

DEFAULT returns null when any of the following conditions are true:

- The specified or derived column was defined with a DEFAULT NULL phrase
- The specified or derived column has no explicit default value
- The data type of the specified or derived column is UDT
- The specified or derived column is the name of a view column that is derived from a single underlying table column that has no explicit default value

  For an example, see "Example 3: Specifying a View Column Name" on page 296.
- The specified or derived column is the name of a view column that is not derived from a single underlying table column, for example, the view column is derived from a literal expression

Omitting the Column Name

You can use the form of DEFAULT that omits the column name under certain conditions in an INSERT, UPDATE, or MERGE statement or in a predicate clause that involves a comparison operation. The form of DEFAULT that omits the column name cannot be part of an expression.

When the DEFAULT function does not specify a column name, Teradata Database derives the column based on context. For example, consider the following table definition:

```
CREATE TABLE Manager
   (Emp_ID      INTEGER
   ,Dept_No     INTEGER DEFAULT 99
);
```

The following INSERT statement uses DEFAULT without a column name to insert the default value into the Dept_No column:

```
INSERT INTO Manager VALUES (103499, DEFAULT);
```

Using the DEFAULT function without specifying a column name can produce an error if Teradata Database cannot derive the column context.

For an example that omits the column name when using the DEFAULT function in a predicate clause that involves a comparison operation, see "Example 2: Using DEFAULT in a Predicate" on page 295.

For details on using the DEFAULT function in INSERT, UPDATE, and MERGE statements, see *SQL Data Manipulation Language*.

Using a Qualified Column Name

If you specify a qualified column name that includes the name of the table, you can use DEFAULT in a SELECT statement that has no FROM clause. For example, you can use the following statement to get the default value of the Dept_No column in the Manager table:

```
SELECT DEFAULT(Manager.Dept_No);
```

Restrictions

The DEFAULT function cannot be used as a partitioning expression for defining PPIs.

Error Conditions

Using the DEFAULT function can result in an error when any of the following conditions are true:

- The column name is omitted and Teradata Database cannot derive the column context
- The DEFAULT function appears in a partitioning expression for defining PPIs
- The column name is omitted and the DEFAULT function appears in an expression that does not support the DEFAULT function without a column name
- The DEFAULT function appears in an expression for which the result type is incompatible

  For example, consider the following table definition:

  ```
  CREATE TABLE Parts_Table
     (Part_Code   INTEGER DEFAULT 9999
     ,Part_Name   CHAR(20)
  );
  ```

  The following statement results in an error because the result type of the DEFAULT function is not compatible with the column to which the result is being compared:

  ```
  SELECT * FROM Parts_Table WHERE Part_Name = DEFAULT(Part_Code);
  ```

Example 1: Inserting the Default Value Under Certain Conditions

Consider the following Employee table definition:

```
CREATE TABLE Employee
   (Emp_ID      INTEGER
   ,Last_Name   VARCHAR(30)
   ,First_Name  VARCHAR(30)
   ,Dept_No     INTEGER DEFAULT 99
);
```

The following statement uses DEFAULT to insert the default value of the Dept_No column when the supplied value is negative.

```
USING (id INTEGER, n1 VARCHAR(30), n2 VARCHAR(30), dept INTEGER)
INSERT INTO Employee VALUES
   (:id
   ,:n1
   ,:n2
   ,CASE WHEN (:dept < 0) THEN DEFAULT(Dept_No) ELSE :dept END
);
```

Example 2: Using DEFAULT in a Predicate

The following statement uses DEFAULT to compare the values of the Dept_No column with the default value of the Dept_No column. Because the comparison operation involves a single column reference, Teradata Database can derive the column context of the DEFAULT function even though the column name is omitted.

```
SELECT * FROM Employee WHERE Dept_No < DEFAULT;
```

Note that if the DEFAULT function evaluates to null, the predicate is unknown and the WHERE condition is false.

Example 3: Specifying a View Column Name

Consider the DBC.HostsInfo system view, which has the following definition:

```
REPLACE VIEW DBC.HostsInfo (LogicalHostId, HostName, DefaultCharSet)
AS SELECT
    LogicalHostId
   ,HostName
   ,DefaultCharSet
FROM DBC.Hosts WITH CHECK OPTION;
```

The underlying table, DBC.Hosts, has the following definition:

```
CREATE SET TABLE DBC.Hosts, FALLBACK, NO BEFORE JOURNAL,
NO AFTER JOURNAL, CHECKSUM = DEFAULT
   (LogicalHostId SMALLINT FORMAT 'ZZZ9' NOT NULL
   ,HostName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT
NULL
   ,DefaultCharSet VARCHAR(128) CHARACTER SET UNICODE NOT
CASESPECIFIC
      NOT NULL)
UNIQUE PRIMARY INDEX (LogicalHostId)
UNIQUE INDEX (HostName);
```

The following statement uses the DEFAULT function with the DBC.HostsInfo.HostName view column name:

```
SELECT DISTINCT DEFAULT(HostName) FROM DBC.HostsInfo;
```

The result of the DEFAULT function is null because the HostName view column is derived from a table column that has no explicit default value.

Related Topics

For information on ...	See ...
using predicates	Chapter 19: “Logical Predicates.”
comparison operations in predicates	Chapter 10: “Comparison Operators and Functions.”
the DEFAULT value control phrase	*SQL Data Types and Literals*.
INSERT, UPDATE, and MERGE statements	*SQL Data Manipulation Language*.

FORMAT

Purpose

Returns the declared format for the named expression.

Syntax

```
– FORMAT – ( column_name ) –
```

1101A489

where:

Syntax element ...	Specifies ...
expression	the expression for which the FORMAT is to be reported.

ANSI Compliance

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

Result Type and Attributes

FORMAT returns a CHAR(*n*) character string of up to 30 characters. The result type, character set, format, and title for FORMAT appear in the following table.

Data Type	Format	Title
CHAR(*n*) CHARACTER SET UNICODE	X(30)	Format(*named_expression*)

For information on data type default formats, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Example

The following statement requests the format of the Salary column in the Employee table.

```
SELECT FORMAT(Employee.Salary);
```

The result is the following.

```
Format(Salary)
------------------------------
ZZZ,ZZ9.99
```

OCTET_LENGTH

Purpose

Returns the length of *string_expression* in octets when it is converted to the named character set (taking the export width value into consideration).

Syntax

```
— OCTET_LENGTH — ( — string_expression ┬──────────────────────┬ ) —
                                       └, character_set_name  ┘
```

1101A513

where:

Syntax element ...	Specifies ...
string_expression	the character string for which the number of octets is required.
character_set_name	the character set in which the result is to be returned. If *character_set_name* is not provided, the session character set is assumed. See the list of Teradata-provided character sets in the table on "Usage Notes" on page 299.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Argument Types

The data type of *string_expression* must be one of the following:

- CHARACTER or VARCHAR
- UDT that has an implicit cast to a predefined character type

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

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

  For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Usage Notes

Any Shift-Out/Shift-In and trailing GRAPHIC pad characters are included in the result count.

OCTET_LENGTH operates in the same manner in both Teradata and ANSI modes.

IF *string_expression* is ...	THEN ...
of type KANJI1	the result is independent of *character_set_name*.
not CHARACTER data	an error is generated.

OCTET_LENGTH takes the export width value into consideration. For information about export width, see "CREATE USER" in *SQL Data Definition Language*.

The following table lists the client character sets shipped with Teradata. Although these character sets are shipped with the system, your system administrator must install them individually to become available for use.

Your site might also have site-defined character sets. Check with your system administrator for a complete list of character sets available at your site.

Character Sets		Where Found
• ASCII • EBCDIC	• UTF8 • UTF16	Built-in
• ARABIC1256_6A0 [a] • CYRILLIC1251_2A0 [a] • EBCDIC037_0E • EBCDIC273_0E • EBCDIC277_0E • HANGUL949_7R0 [a] • HANGULEBCDIC933_1II • HANGULKSC5601_2R4 • HEBREW1255_5A0 [a] • KANJI932_1S0 [a] • KANJIEBCDIC5026_0I • KANJIEBCDIC5035_0I • KANJIEUC_0U • KANJISJIS_0S • KATAKANAEBCDIC	• LATIN1250_1A0 [a] • LATIN1252_0A • LATIN1252_3A0 [a] • LATIN1254_7A0 [a] • LATIN1258_8A0 [a] • LATIN1_0A • LATIN9_0A • SCHEBCDIC935_2IJ • SCHGB2312_1T0 • SCHINESE936_6R0 [a] • TCHBIG5_1R0 • TCHEBCDIC937_3IB • TCHINESE950_8R0 [a] • THAI874_4A0 [a]	DBC.CharTranslationsV

a. Windows code page compatible session character set

Examples

Examples of output from OCTET_LENGTH appear in the following table.

Client Character Set	Server Character Set	string_expression	Result
EBCDIC	LATIN	abcdefgh	8
ASCII	KANJI1	abcdefgh	8
KanjiEBCDIC	KANJI1	AB<**CDE**>P	11
KanjiEBCDIC	GRAPHIC	MNOP	8 (record mode)
			10 (field mode)
KanjiEUC	KANJISJIS	d**A** ss_2**B** ss_3**E**	8
KanjiShift-JIS	KANJISJIS	**DeF**	5
KanjiShift-JIS	UNICODE	ABC	6

TITLE

Purpose

Returns the title of an expression as it would appear in the heading for displayed or printed results.

Syntax

```
── TITLE ─ ( expression ) ─
```

1101B039

where:

Syntax element ...	Specifies ...
expression	the expression for which the title is to be returned.

ANSI Compliance

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

Result Type and Attributes

TITLE returns a CHAR(*n*) character string of up to 60 characters. The result type, character set, format, and title for TITLE appear in the following table.

Data Type	Format	Title
CHAR(*n*) CHARACTER SET UNICODE	X(60)	Title(*named_expression*)

For information on data type default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Usage Notes

Use the TITLE phrase to change the heading for displayed or printed results that is different from the column name, which is the default heading.

For more information, see *SQL Data Types and Literals*.

Example

The following statement requests the title of the Salary column in the Employee table.

```
SELECT TITLE(Employee.Salary);
```

The result is the following.

```
Title(Salary)
------------------------------------------------------------
Salary
```

TYPE

Purpose

Returns the data type defined for an expression.

Syntax

```
— TYPE —( expression )—
```

1101A491

where:

Syntax element ...	Specifies ...
expression	the expression for which the data type is to be returned.

ANSI Compliance

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

Result Type and Attributes

TYPE returns a CHAR(*n*) character string that contains the name of the data type of the expression.

When the argument is a function or operation, TYPE returns a character string that contains the result type of the function or operation. For rules on the result type for an operation or function, refer to the documentation for the specific function or operation.

The result type, character set, format, and title for TYPE appear in the following table.

Data Type	Format	Title
CHAR(*n*) CHARACTER SET LATIN	X(39)	Type(*named_expression*)

For a list of the supported data types, see *SQL Data Types and Literals*. For information on geospatial types, see *SQL Geospatial Types*.

Character Type Arguments

If the server character set for a character type argument is different from the user default server character set, then the resulting character string also contains the CHARACTER SET phrase and the name of the server character set for the argument.

For examples, see "Example 1" and "Example 2" on page 304.

Example 1

Consider the Name column in the following table definition:

```
CREATE TABLE Employee
  (EmployeeID INTEGER
  ,Name       CHARACTER(30) CHARACTER SET LATIN
  ,Salary     DECIMAL(8,2));
```

If the user default server character set is LATIN, then the character string that TYPE returns for the Name column does not contain the CHARACTER SET phrase.

```
SELECT TYPE(Employee.Name);

Type(Name)
----------
CHAR(30)
```

Example 2

If the user default server character set is LATIN, but the server character set for the Name column is UNICODE, then the result string contains the CHARACTER SET phrase.

```
CREATE TABLE Employee
  (EmployeeID INTEGER
  ,Name VARCHAR(30) CHARACTER SET UNICODE
  ,Salary     DECIMAL(8,2));

SELECT TYPE(Employee.Name);

Type(Name)
---------------------------------
VARCHAR(30) CHARACTER SET UNICODE
```

Example 3

The following statement returns the types of the Name and Salary columns:

```
SELECT TYPE(Employee.Name), TYPE(Employee.Salary);

Type(Name)   Type(Salary)
-----------  ------------
VARCHAR(30)  DECIMAL(8,2)
```

Example 4

If TYPE is used to request the data type of two columns, defined as GRAPHIC and LONG VARGRAPHIC, respectively, the result is as follows.

```
TYPE(GColName)                 TYPE(LVGColName)
-----------------------------  -------------------------------------
CHAR(4) CHARACTER SET GRAPHIC  VARCHAR(32000) CHARACTER SET GRAPHIC
```

In the case of a LONG VARGRAPHIC column, the length returned is the maximum length of 32000.

Example 5

Consider the following TYPE function.

```
SELECT TYPE(SUBSTR(Employee.Name,3,2));
```

The result type of SUBSTR depends on the session mode.

If the session is set to ANSI mode, the returned result is as follows:

```
Type(Substr(Name,3,2))
----------------------
VARCHAR(30)
```

If the session is set to Teradata mode, the returned result is as follows:

```
Type(Substr(Name,3,2))
----------------------
VARCHAR(2)
```

Example 6

Consider the following table definition:

```
CREATE TABLE images
  (imageid INTEGER
  ,imagedesc VARCHAR(50)
  ,image BLOB(2K))
UNIQUE PRIMARY INDEX (imageid);
```

The following statement applies the TYPE function to the BLOB column:

```
SELECT TYPE(images.image) FROM images;
```

The result is:

```
Type(image)
-----------
BLOB(2048)
```

Note that the result is a normal integer length, and does not use the K option that was used to define the BLOB column the CREATE TABLE statement.

CHAPTER 6 Bit/Byte Manipulation Functions

This chapter describes the functions that provide support for byte/bit manipulation operations.

Prerequisites

The byte/bit manipulation functions in this chapter are embedded services system functions. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Bit and Byte Numbering Model

The following diagrams show the logical bit and byte numbering model employed by the byte/bit manipulation functions described in this chapter.

The model is big endian or little endian independent. Note that the numbering system used for numeric data types is consistent with that used for byte strings. This simplifies the development of appropriate bit masks.

Users of the byte/bit manipulation functions should mentally visualize the numeric and byte data types as shown below when contemplating what masks (*bit_mask_arg*) need to be applied to the target data (*target_arg*).

BYTEINT

Example

A BYTEINT value of 40 with a binary representation of 00101000:

SMALLINT

Example

A SMALLINT value of 10,280 with a binary representation of 0010100000101000:

INTEGER

Example

An INTEGER value of 673,720,360 with a binary representation of 00101000 00101000 00101000 00101000:

BIGINT

Example

A BIGINT value of 2,893,606,913,523,066,920 with a binary representation of 00101000 00101000 00101000 00101000 00101000 00101000 00101000 00101000:

BYTE and VARBYTE

Example 1

A VARBYTE(8) with 8 bytes:

Example 2

A VARBYTE(8) with 3 bytes:

Example 3

Example of BYTE(4):

HEXADECIMAL BYTE LITERALS

With respect to byte-bit system functions, hexadecimal byte literals are interpreted as follows:

A 2-byte hexadecimal byte literal: '00FF'XB

A 4-byte hexadecimal byte literal: '01020304'XB

Note that hexadecimal byte literals are represented by an even number of hexadecimal digits. Hexadecimal literals are extended on the right with zeros when required. For example:

A 3-byte hexadecimal byte literal, '112233'XB, becomes a 4-byte hexadecimal byte literal: '11223300'XB

For more information, see “Hexadecimal Byte Literals” in *SQL Data Types and Literals*.

Performing Bit-Byte Operations against Arguments with Non-Equal Lengths

This topic applies only to the BITOR, BITXOR, and BITAND functions.

If the *target_arg* and *bit_mask_arg* arguments passed to these functions differ in length:

- The *target_arg* and *bit_mask_arg* arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

  Teradata Database pads to the left (instead of to the right) so that the hexadecimal byte literals, serving as bit masks, will not have to be changed every time the size of a byte string is changed.

Example

The following query performs the BITAND operation on an INTEGER and a single-byte hexadecimal byte literal.

```
SELECT BITAND(287454020, 'FFFF'XB);
```

The INTEGER value 287,454,020 has a hexadecimal value of 0x11223344 and a bit numbering representation of:

The hexadecimal byte literal 0xFFFF has a bit numbering representation of:

To process the BITAND operation, the two arguments are aligned on their least significant byte/bit as follows:

The shorter-length hexadecimal byte literal 0xFFFF is padded with zeros to the left until it is the same length as the INTEGER value 287,454,020.

When both operands are the same size, the BITAND operation is performed, producing the following result:

BITAND

Purpose

Performs the logical AND operation on the corresponding bits from the two input arguments.

Syntax

```
┬──────────────┬── BITAND ── ( target_arg, bit_mask_arg ) ──────────────
└─ TD_SYSFNLIB. ┘
                                                          1101A671
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable byte expression.
bit_mask_arg	a fixed byte value, a variable byte value, or a numeric expression.

ANSI Compliance

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

Description

This function takes two bit patterns of equal length and performs the logical AND operation on each pair of corresponding bits. If the bits at the same position are both 1, then the result is 1; otherwise, the result is 0. If either input argument is NULL, the function returns NULL.

If the *target_arg* and *bit_mask_arg* arguments differ in length, the arguments are processed as follows:

- The *target_arg* and *bit_mask_arg* arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

For more information, see "Performing Bit-Byte Operations against Arguments with Non-Equal Lengths" on page 311.

Argument Types and Rules

BITAND is an overloaded scalar function. The data type of the *target_arg* parameter can be one of the following:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- VARBYTE(*n*)

The data type of the *bit_mask_arg* parameter varies depending upon the data type of the *target_arg* parameter. The following (*target_arg*, *bit_mask_arg*) input combinations are permitted:

target_arg type	*bit_mask_arg* type
BYTEINT	BYTE(1)
BYTEINT	BYTEINT
SMALLINT	BYTE(2)
SMALLINT	SMALLINT
INTEGER	BYTE(4)
INTEGER	INTEGER
BIGINT	BYTE(8)
BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs. For example, BITAND(BYTEINT, INTEGER) is allowed because it can be implicitly converted to BITAND(INTEGER,INTEGER).

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for BITAND is: BITAND(*target_arg, bit_mask_arg*).

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

Example

In the following query, the input argument 23 has a data type of BYTEINT and a binary representation of 00010111. The input argument 20 has a data type of BYTEINT and a binary representation of 00010100. The bitwise AND product of the two arguments results in a BYTEINT value of 20, or binary 00010100, which is returned by the query.

```
SELECT BITAND(23,20);
```

BITNOT

Purpose

Performs a bitwise complement on the binary representation of the input argument.

Syntax

```
┬──────────────┬── BITNOT ── ( target_arg ) ──────────────────────────
└─ TD_SYSFNLIB.┘
```

1101A669

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable byte expression.

ANSI Compliance

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

Description

The bitwise NOT, or complement, is a unary operation which performs logical negation on each bit, forming the ones' complement of the specified binary value. The digits in the argument which were 0 become 1, and vice versa. BITNOT returns NULL if *target_arg* is NULL.

Argument Types and Rules

BITNOT is an overloaded scalar function. It is defined with the following parameter data types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *SQL External Routine Programming*.

If the argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see "Function Name Overloading" in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for BITNOT is: BITNOT(*target_arg*).

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

Example

In the following query, the input argument 2 has a data type of BYTEINT and a binary representation of 00000010. Performing a BITNOT operation on this value results in a BYTEINT value of -3, or binary 11111101.

```
SELECT BITNOT(2);
```

BITOR

Purpose

Performs the logical OR operation on the corresponding bits from the two input arguments.

Syntax

```
┌──────────────────────────┐
│                          │
└─ TD_SYSFNLIB. ───────────┘── BITOR ── ( target_arg, bit_mask_arg ) ──────────────────────────
                                                                                   1101A668
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable byte expression.
bit_mask_arg	a fixed byte value, a variable byte value, or a numeric expression.

ANSI Compliance

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

Description

This function takes two bit patterns of equal length and performs the logical OR operation on each pair of corresponding bits as follows:

IF...	THEN the result is...
either of the bits from the input arguments is 1	1
both of the bits from the input arguments are 0	0
any of the input arguments is NULL	NULL

If the *target_arg* and *bit_mask_arg* arguments differ in length, the arguments are processed as follows:

- The *target_arg* and *bit_mask_arg* arguments are aligned on their least significant byte/bit.

- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

For more information, see "Performing Bit-Byte Operations against Arguments with Non-Equal Lengths" on page 311.

Argument Types and Rules

BITOR is an overloaded scalar function. The data type of the *target_arg* parameter can be one of the following:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- VARBYTE(*n*)

The data type of the *bit_mask_arg* parameter varies depending upon the data type of the *target_arg* parameter. The following (*target_arg*, *bit_mask_arg*) input combinations are permitted:

target_arg type	*bit_mask_arg* type
BYTEINT	BYTE(1)
BYTEINT	BYTEINT
SMALLINT	BYTE(2)
SMALLINT	SMALLINT
INTEGER	BYTE(4)
INTEGER	INTEGER
BIGINT	BYTE(8)
BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs. For example, BITOR(BYTEINT, INTEGER) is allowed because it can be implicitly converted to BITOR(INTEGER,INTEGER).

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly

cast. For details, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for BITOR is: BITOR(*target_arg*, *bit_mask_arg*).

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

Example

In the following query, the input argument 23 has a data type of BYTEINT and a binary representation of 00010111. The input argument 45 has a data type of BYTEINT and a binary representation of 00101101. The bitwise OR product of the two arguments results in a BYTEINT value of 63, or binary 00111111, which is returned by the query.

```
SELECT BITOR(23,45);
```

BITXOR

Purpose

Performs a bitwise XOR operation on the binary representation of the two input arguments.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── BITXOR ── ( target_arg, bit_mask_arg ) ──────────────
                                                                   1101A670
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable byte expression.
bit_mask_arg	a fixed byte value, a variable byte value, or a numeric expression.

ANSI Compliance

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

Description

The bitwise exclusive OR takes two bit patterns of equal length and performs the logical XOR operation on each pair of corresponding bits. The result in each position is 1 if the two bits are different, and 0 if they are the same. If either input argument is NULL, the function returns NULL.

If the *target_arg* and *bit_mask_arg* arguments differ in length, the arguments are processed as follows:

- The *target_arg* and *bit_mask_arg* arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

For more information, see "Performing Bit-Byte Operations against Arguments with Non-Equal Lengths" on page 311.

Argument Types and Rules

BITXOR is an overloaded scalar function. The data type of the *target_arg* parameter can be one of the following:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- VARBYTE(*n*)

The data type of the *bit_mask_arg* parameter varies depending upon the data type of the *target_arg* parameter. The following (*target_arg*, *bit_mask_arg*) input combinations are permitted:

target_arg type	*bit_mask_arg* type
BYTEINT	BYTE(1)
BYTEINT	BYTEINT
SMALLINT	BYTE(2)
SMALLINT	SMALLINT
INTEGER	BYTE(4)
INTEGER	INTEGER
BIGINT	BYTE(8)
BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs. For example, BITXOR(BYTEINT, INTEGER) is allowed because it can be implicitly converted to BITXOR(INTEGER,INTEGER).

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for BITXOR is: BITXOR(*target_arg*, *bit_mask_arg*).

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

Example

In the following query, the input argument 12 has a data type of BYTEINT and a binary representation of 00001100. The input argument 45 has a data type of BYTEINT and a binary representation of 00101101. The bitwise XOR product of the two arguments results in a BYTEINT value of 33, or binary 00100001, which is returned by the query.

```
SELECT BITXOR(12,45);
```

COUNTSET

Purpose

Returns the count of the binary bits within the *target_arg* expression that are either set to 1 or set to 0 depending on the *target_value_arg* value.

Syntax

```
┬──────────────────┬── COUNTSET ── ( target_arg ┬──────────────────┬ ) ──
└ TD_SYSFNLIB. ────┘                           └ , target_value_arg ┘
                                                                    1101A676
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable byte expression.
target_value_arg	an optional integer value. Only a value of 0 or 1 is allowed. If *target_value_arg* is not specified, the default is 1.

ANSI Compliance

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

Description

COUNTSET takes the *target_arg* input expression and counts the number of bits within the expression that are either set to 1 or set to 0, depending on the value of *target_value_arg*.

The *target_value_arg* parameter only accepts a value of 0 or 1. If a value for *target_value_arg* is not specified, the default value of 1 is used, and COUNTSET counts the bit values that are set to 1.

If any of the input arguments is NULL, the function returns NULL.

Argument Types and Rules

COUNTSET is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*[, *target_value_arg*]) input combinations:

target_arg type	*target_value_arg* type (optional)
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see "Function Name Overloading" in *SQL External Routine Programming*.

Result Type and Attributes

The result data type is INTEGER.

The result format is the default format for INTEGER.

The default title for COUNTSET is: COUNTSET(*target_arg*[, *target_value_arg*]).

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

Example

The following query takes the input argument 23, which has a data type of BYTEINT and a binary representation of 00010111. Since *target_value_arg* is not specified, the default value of 1 is used. Therefore, the function counts the number of bit values that are set to 1. The query result is an INTEGER value of 4.

```
SELECT COUNTSET(23);
```

GETBIT

Purpose

Returns the value of the bit specified by *target_bit_arg* from the *target_arg* byte expression.

Syntax

```
┬──────────────────┬── GETBIT ── ( target_arg, target_bit_arg ) ──────────────────
└─ TD_SYSFNLIB. ─┘
                                                                  1101A672
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable byte expression.
target_bit_arg	an integer expression.

ANSI Compliance

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

Description

GETBIT gets the bit specified by *target_bit_arg* from the *target_arg* byte expression and returns either 0 or 1 to indicate the value of that bit.

The range of input values for *target_bit_arg* can vary from 0 (bit 0 is the least significant bit) to the (sizeof(*target_arg*) - 1).

If *target_bit_arg* is negative or out-of-range (meaning that it exceeds the size of *target_arg*), an error is returned.

If either input argument is NULL, the function returns NULL.

Argument Types and Rules

GETBIT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *target_bit_arg*) input combinations:

target_arg type	*target_bit_arg* type
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see "Function Name Overloading" in *SQL External Routine Programming*.

Result Type and Attributes

GETBIT returns a BYTEINT value of either 0 or 1, reflecting the value of the bit residing at the *target_bit_arg* position of the *target_arg* byte expression.

The result format is the default format for BYTEINT.

The default title for GETBIT is: GETBIT(*target_arg*, *target_bit_arg*).

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

Example

The following query gets the value of the third bit of the input argument 23, which has a data type of BYTEINT and a binary representation of 00010111. The query result is a BYTEINT value of 1 or binary 00000001.

```
SELECT GETBIT(23,2);
```

ROTATELEFT

Purpose

Returns the expression *target_arg* rotated by the specified number of bits (*num_bits_arg*) to the right, with the most significant bits wrapping around to the left.

Syntax

```
┌──────────────────────────┐
│                          │
└─ TD_SYSFNLIB. ───────────┘── ROTATELEFT ── ( target_arg, num_bits_arg ) ──────────────────────
                                                                                  1101A664
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable expression.
num_bits_arg	an integer expression indicating the number of bit positions to rotate.

ANSI Compliance

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

Description

IF...	THEN the function...
num_bits_arg is equal to zero	returns *target_arg* unchanged.
num_bits_arg is negative	rotates the bits to the right instead of the left.
target_arg and/or *num_bits_arg* are NULL	returns NULL.
num_bits_arg is larger than the size of *target_arg*	rotates (*num_bits_arg* MOD sizeof(*target_arg*)) bits. The scope of the rotation operation is bounded by the size of the *target_arg* expression.

Note: When operating against an integer value (BYTEINT, SMALLINT, INTEGER, or BIGINT), rotating a bit into the most significant position will result in the integer becoming negative. This is because all integers in Teradata Database are signed integers.

Argument Types and Rules

ROTATELEFT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *num_bits_arg*) input combinations:

target_arg type	*num_bits_arg* type
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see "Function Name Overloading" in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for ROTATELEFT is: ROTATELEFT(*target_arg*, *num_bits_arg*).

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

Example 1

In the following query, the input argument 16 has a data type of BYTEINT and a binary representation of 00010000. When this value is rotated left by two bits, the result in binary is 01000000. This value translates to a BYTEINT value of 64, which is the result returned by the query.

```
SELECT ROTATELEFT(16,2);
```

Example 2

In the following query, the input argument 64 has a data type of BYTEINT and a binary representation of 01000000. When this value is rotated left by three bits, the result in binary is 00000010. This value translates to a BYTEINT value of 2, which is the result returned by the query.

```
SELECT ROTATELEFT(64,3);
```

ROTATERIGHT

Purpose

Returns the expression *target_arg* rotated by the specified number of bits (*num_bits_arg*) to the right, with the least significant bits wrapping around to the left.

Syntax

```
┬──────────────────┬── ROTATERIGHT ── ( target_arg, num_bits_arg ) ──────────────────
└─ TD_SYSFNLIB. ─┘
                                                                          1101A665
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable expression.
num_bits_arg	an integer expression indicating the number of bit positions to rotate.

ANSI Compliance

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

Description

IF...	THEN the function...
num_bits_arg is equal to zero	returns *target_arg* unchanged.
num_bits_arg is negative	rotates the bits to the left instead of the right.
target_arg and/or *num_bits_arg* are NULL	returns NULL.
num_bits_arg is larger than the size of *target_arg*	rotates (*num_bits_arg* MOD sizeof(*target_arg*)) bits. The scope of the rotation operation is bounded by the size of the *target_arg* expression.

Note: When operating against an integer value (BYTEINT, SMALLINT, INTEGER, or BIGINT), rotating a bit into the most significant position will result in the integer becoming negative. This is because all integers in Teradata Database are signed integers.

Argument Types and Rules

ROTATERIGHT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *num_bits_arg*) input combinations:

target_arg type	*num_bits_arg* type
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see "Function Name Overloading" in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for ROTATERIGHT is: ROTATERIGHT(*target_arg*, *num_bits_arg*).

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

Example 1

In the following query, the input argument 32 has a data type of BYTEINT and a binary representation of 00100000. When this value is rotated right by two bits, the result in binary is 00001000. This value translates to a BYTEINT value of 8, which is the result returned by the query.

```
SELECT ROTATERIGHT(32,2);
```

Example 2

In the following query, the input argument 4 has a data type of BYTEINT and a binary representation of 00000100. When this value is rotated right by four bits, the result in binary is 01000000. This value translates to a BYTEINT value of 64, which is the result returned by the query.

```
SELECT ROTATERIGHT(4,4);
```

SETBIT

Purpose

Sets the value of the bit specified by *target_bit_arg* to the value of *target_value_arg* in the *target_arg* byte expression.

Syntax

```
┬──────────────┬── SETBIT ── ( target_arg, target_bit_arg ┬──────────────────┬ ) ──
└ TD_SYSFNLIB. ┘                                          └ , target_value_arg ┘
                                                                                1101A673
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable byte expression.
target_bit_arg	an integer expression.
target_value_arg	an optional integer value. Only a value of 0 or 1 is allowed. If *target_value_arg* is not specified, the default is 1.

ANSI Compliance

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

Description

SETBIT takes the *target_arg* input and sets the bit specified by *target_bit_arg* to the value, 0 or 1, as provided by the *target_value_arg* argument.

The *target_value_arg* parameter only accepts a value of 0 or 1. If a value for *target_value_arg* is not specified, the default value of 1 is used.

The range of input values for *target_bit_arg* can vary from 0 (bit 0 is the least significant bit) to the (sizeof(*target_arg*) - 1).

If *target_bit_arg* is negative or out-of-range (meaning that it exceeds the size of *target_arg*), an error is returned.

If any of the input arguments is NULL, the function returns NULL.

Argument Types and Rules

SETBIT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *target_bit_arg*[,*target_value_arg*]) input combinations:

target_arg type	*target_bit_arg* type	*target_value_arg* type (optional)
BYTEINT	INTEGER	INTEGER
SMALLINT	INTEGER	INTEGER
INTEGER	INTEGER	INTEGER
BIGINT	INTEGER	INTEGER
VARBYTE(*n*)	INTEGER	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for SETBIT is: SETBIT(*target_arg*, *target_bit_arg*[,*target_value_arg*]).

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

Example 1

The following query takes the input argument 23, which has a data type of BYTEINT and a binary representation of 00010111, and sets the value of the third bit to 1. The query result is a BYTEINT value of 23 or binary 00010111.

```
SELECT SETBIT(23,2);
```

Example 2

The following query takes the input argument 23, which has a data type of BYTEINT and a binary representation of 00010111, and sets the value of the third bit to 0. The query result is a BYTEINT value of 19 or binary 00010011.

```
SELECT SETBIT(23,2,0);
```

SHIFTLEFT

Purpose

Returns the expression *target_arg* shifted by the specified number of bits (*num_bits_arg*) to the left. The bits in the most significant positions are lost, and the bits in the least significant positions are filled with zeros.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── SHIFTLEFT ── ( target_arg, num_bits_arg ) ──────────────────────
```

1101A667

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable expression.
num_bits_arg	an integer expression indicating the number of bit positions to shift.

ANSI Compliance

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

Description

IF...	THEN the function...
num_bits_arg is equal to zero	returns *target_arg* unchanged.
num_bits_arg is negative	shifts the bits to the right instead of the left.
target_arg and/or *num_bits_arg* are NULL	returns NULL.
num_bits_arg is larger than the size of *target_arg*	returns an error. The scope of the shift operation is bounded by the size of the *target_arg* expression. Specifying a shift that is outside the range of *target_arg* results in an SQL error.

Note: When operating against an integer value (BYTEINT, SMALLINT, INTEGER, or BIGINT), shifting a bit into the most significant position will result in the integer becoming negative. This is because all integers in Teradata Database are signed integers.

Argument Types and Rules

SHIFTLEFT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *num_bits_arg*) input combinations:

target_arg type	*num_bits_arg* type
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see "Function Name Overloading" in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for SHIFTLEFT is: SHIFTLEFT(*target_arg*, *num_bits_arg*).

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

Example

In the following query, the input argument 3 has a data type of BYTEINT and a binary representation of 00000011. When this value is shifted left by two bits, the result in binary is 00001100. This value translates to a BYTEINT value of 12, which is the result returned by the query.

```
SELECT SHIFTLEFT(3,2);
```

SHIFTRIGHT

Purpose

Returns the expression *target_arg* shifted by the specified number of bits (*num_bits_arg*) to the right. The bits in the least significant positions are lost, and the bits in the most significant positions are filled with zeros.

Syntax

```
┬──────────────┬── SHIFTRIGHT ── ( target_arg, num_bits_arg ) ──────────────
└ TD_SYSFNLIB. ┘
```

1101A666

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable expression.
num_bits_arg	an integer expression indicating the number of bit positions to shift.

ANSI Compliance

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

Description

IF...	THEN the function...
num_bits_arg is equal to zero	returns *target_arg* unchanged.
num_bits_arg is negative	shifts the bits to the left instead of the right.
target_arg and/or *num_bits_arg* are NULL	returns NULL.
num_bits_arg is larger than the size of *target_arg*	returns an error. The scope of the shift operation is bounded by the size of the *target_arg* expression. Specifying a shift that is outside the range of *target_arg* results in an SQL error.

Note: When operating against an integer value (BYTEINT, SMALLINT, INTEGER, or BIGINT), shifting a bit out of the most significant position will result in the integer becoming negative. This is because all integers in Teradata Database are signed integers.

Argument Types and Rules

SHIFTRIGHT is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *num_bits_arg*) input combinations:

target_arg type	*num_bits_arg* type
BYTEINT	INTEGER
SMALLINT	INTEGER
INTEGER	INTEGER
BIGINT	INTEGER
VARBYTE(*n*)	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see “Compatible Types” and “Parameter Types in Overloaded Functions” in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result Type and Attributes

The result data type depends on the data type of the *target_arg* input argument that is passed to the function as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTEINT	BYTEINT
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BIGINT	BIGINT	BIGINT
VARBYTE(*n*)	VARBYTE(*n*)	VARBYTE(*n*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for SHIFTRIGHT is: SHIFTRIGHT(*target_arg*, *num_bits_arg*).

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

Example

In the following query, the input argument 3 has a data type of BYTEINT and a binary representation of 00000011. When this value is shifted right by two bits, the result in binary is 00000000. This value translates to a BYTEINT value of 0, which is the result returned by the query.

```
SELECT SHIFTRIGHT(3,2);
```

SUBBITSTR

Purpose

Extracts a bit substring from the *target_arg* input expression based on the specified bit position.

Syntax

```
┌──────────────────────────┐
│                          │
└─ TD_SYSFNLIB. ───────────┘── SUBBITSTR ── ( target_arg, position_arg, num_bits_arg ) ──
                                                                                  1101A674
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric or variable byte expression.
position_arg	an integer expression indicating the starting position of the bit substring to be extracted.
num_bits_arg	an integer expression indicating the length of the bit substring to be extracted. This specifies the number of bits for the function to return.

ANSI Compliance

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

Description

SUBBITSTR extracts a bit substring from the *target_arg* string expression starting at the bit position specified by *position_arg*. See "Bit and Byte Numbering Model" on page 307 for the range of bit positions for each data type.

The *num_bits_arg* value specifies the length of the bit substring to be extracted and indicates the number of bits that the function should return. Note that since the return value of the function is a VARBYTE string, the number of bits returned will be rounded to the byte boundary greater than the number of bits requested.

The bits returned will be right-justified, and the excess bits (those exceeding the requested number of bits) will be filled with zeroes.

If *position_arg* is negative or out-of-range (meaning that it exceeds the size of *target_arg*), an error is returned.

If *num_bits_arg* is negative, or is greater than the number of bits remaining once the starting *position_arg* is taken into account, an error is returned.

If any of the input arguments is NULL, the function returns NULL.

Argument Types and Rules

SUBBITSTR is an overloaded scalar function. It is defined with the following parameter data types for the following (*target_arg*, *position_arg*, *num_bits_arg*) input combinations:

target_arg type	*position_arg* type	*num_bits_arg* type
BYTEINT	INTEGER	INTEGER
SMALLINT	INTEGER	INTEGER
INTEGER	INTEGER	INTEGER
BIGINT	INTEGER	INTEGER
VARBYTE(*n*)	INTEGER	INTEGER

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If any argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *SQL External Routine Programming*.

If any argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see "Function Name Overloading" in *SQL External Routine Programming*.

Result Type and Attributes

The result data type is a VARBYTE string. The size (number of bytes) of the VARBYTE string depends on the data type of the *target_arg* input argument and the number of bits requested.

For example:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	VARBYTE(1)	VARBYTE(1)
SMALLINT	VARBYTE(2)	VARBYTE(2)

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
INTEGER	VARBYTE(4)	VARBYTE(4)
BIGINT	VARBYTE(8)	VARBYTE(8)
VARBYTE(*n*)	VARBYTE(*m*) where *m* is the smallest number of bytes to accommodate the requested number of bits.	VARBYTE(*m*)

The maximum supported size (*n*) for VARBYTE is 8192 bytes.

The default title for SUBBITSTR is: SUBBITSTR(*target_arg*, *position_arg*, *num_bits_arg*).

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

Example

The following query takes the input argument 20, which has a data type of BYTEINT and a binary representation of 00010100, and requests that 3 bits be returned starting at the third bit. The 3 bits returned are 101, which are placed into a right-justified zero-filled byte. The result from the query is a value of 5, or binary 00000101, with the result data type being VARBYTE(1).

```
SELECT SUBBITSTR(20,2,3);
```

TO_BYTE

Purpose

Converts a numeric data type to the Teradata Database server byte representation (byte value) of the input value.

Syntax

```
┬──────────────────┬── TO_BYTE ── ( target_arg ) ──────────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A675

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
target_arg	a numeric expression.

ANSI Compliance

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

Description

The number of bytes returned by the function varies according to the data type of the *target_arg* value.

For information on the server representation of integral values, see *SQL Data Types and Literals*.

If *target_arg* is NULL, the function returns NULL.

Argument Types and Rules

TO_BYTE is an overloaded scalar function. It is defined with the following parameter data types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT

All expressions passed to this function must either match these declared data types or can be converted to these types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to one of the declared data types by following UDF implicit conversion rules, it must be explicitly cast. For details, see "Compatible Types" and "Parameter Types in Overloaded Functions" in *SQL External Routine Programming*.

If the argument cannot be converted to one of the declared data types, an error is returned indicating that no function exists that matches the DML UDF expression submitted.

For more information on overloaded functions, see "Function Name Overloading" in *SQL External Routine Programming*.

Result Type and Attributes

The result data type is a BYTE value (a fixed byte data type). The size of the byte string returned varies according to the data type of the *target_arg* input argument as shown in the following table:

IF the data type of *target_arg* is...	THEN the result type is...	AND the result format is the default format for...
BYTEINT	BYTE(1)	BYTE(1)
SMALLINT	BYTE(2)	BYTE(2)
INTEGER	BYTE(4)	BYTE(4)
BIGINT	BYTE(8)	BYTE(8)

The default title for TO_BYTE is: TO_BYTE(*target_arg*).

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

Example

In the following query, the input argument 23 has a data type of BYTEINT and a binary representation of 00010111. Performing a TO_BYTE operation on this value results in the value 00010111 being returned with the data type of BYTE(17).

```
SELECT TO_BYTE(23);
```

CHAPTER 7 Built-In Functions

Built-in functions, which are niladic (have no arguments), return various information about the system. Built-in functions are sometimes referred to as special registers.

The built-in functions can be used anywhere that a literal can appear.

If a SELECT statement that contains a built-in function references a table name, then the result of the query contains one row for every row of the table that satisfies the search condition.

ACCOUNT

Purpose

Returns the account string for the current user.

Syntax

```
─── ACCOUNT ───
```

FF07R001

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for ACCOUNT are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Account

Usage Notes

If a SET SESSION ACCOUNT statement has changed the current account string, then the ACCOUNT function returns the new account string based on the request level: whether for an entire session or for an individual request.

Example

The following statement requests the account string for the current user:

```
SELECT ACCOUNT;
```

The system responds with something like the following:

```
Account
------------------------------
$M_D2102
```

CURRENT_DATE

Purpose

Returns the current date.

Syntax

where:

Syntax element ...	Specifies ...
AT LOCAL	that the value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled. If TimeDateWZControl is disabled, the value returned is constructed from the time value local to the Teradata Database server and the session time zone.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

CURRENT_DATE and the AT clause are ANSI SQL:2011 compliant.

As an extension to ANSI, you can specify an AT clause after the CURRENT_DATE function, and you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Usage Notes

CURRENT_DATE returns the current date at the time when the request started. If CURRENT_DATE is invoked more than once during the request, the same date is returned. The date returned does not change during the duration of the request.

If you specify CURRENT_DATE without the AT clause or CURRENT_DATE AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, CURRENT_DATE returns a date constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, CURRENT_DATE returns a date constructed from the time value local to the Teradata Database server and the session time zone.

For more information, see "DBS Control (dbscontrol)" in *Utilities*.

CURRENT_DATE returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- CURRENT_DATE is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- CURRENT_DATE is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

For more information about time zone strings, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

Result Type and Attributes

The result data type, format, and title for CURRENT_DATE are:

Data Type	Format	Title
DATE	Default format for the DATE data type when the Dateform mode is set to IntegerDate. For more information on the default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.	Date

To convert CURRENT_DATE, use Teradata explicit conversion syntax or ANSI CAST syntax. For an example that uses Teradata explicit conversion syntax to change the default output format, see "Example 3: Changing the Default Output Format" on page 360.

CURRENT_DATE versus DATE

CURRENT_DATE provides similar functionality to the Teradata function DATE using ANSI-compliant syntax. For information on the Teradata DATE function, see "DATE" on page 367.

Example 1

This example assumes that the default format for DATE values is 'yy/mm/dd'. Consider the following statements:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CURRENT_DATE AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;
SELECT CURRENT_DATE AT INTERVAL -'08:00' HOUR TO MINUTE;
SELECT CURRENT_DATE AT TIME ZONE INTERVAL -'08' HOUR;
SELECT CURRENT_DATE AT INTERVAL -'08' HOUR;
SELECT CURRENT_DATE AT TIME ZONE '-08:00';
SELECT CURRENT_DATE AT '-08:00';
SELECT CURRENT_DATE AT TIME ZONE '-8';
SELECT CURRENT_DATE AT '-8';
SELECT CURRENT_DATE AT TIME ZONE -8;
SELECT CURRENT_DATE AT -8;
SELECT CURRENT_DATE AT -8.0;
```

The above SELECT statements return the current date based on the time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. If the current timestamp at UTC is TIMESTAMP '2008-06-01 06:30:00.000000+00:00', these SELECT statements would return '08/05/31' as the date.

If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, and the DBS Control flag TimeDateWZControl is enabled, the statement would return '08/06/01' as the current date based on the current session time and time zone displacement, INTERVAL '01:00' HOUR TO MINUTE. For example:

```
SELECT CURRENT_DATE;
SELECT CURRENT_DATE AT LOCAL;
```

The date returned is not adjusted to account for the start or end of daylight saving time.

Example 2

This example assumes that the default format for DATE values is 'yy/mm/dd'. Consider the following statements:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;
SELECT CURRENT_DATE AT INTERVAL '09:00' HOUR TO MINUTE;
```

The above SELECT statement returns the current date based on the time zone displacement, INTERVAL '09:00' HOUR TO MINUTE. If the current timestamp at UTC is TIMESTAMP '2008-06-01 19:30:00.000000+00:00', the SELECT statement would return '08/06/02' as the date.

If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, and the DBS Control flag TimeDateWZControl is enabled, the statement would return '08/06/01' as the current date based on the current session time and time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The date returned is not adjusted to account for the start or end of daylight saving time.

Example 3

This example assumes that the default format for DATE values is 'yy/mm/dd'. Consider the following statements:

```
SET TIME ZONE INTERVAL '10:00' HOUR TO MINUTE;

SELECT CURRENT_DATE AT '05:45';
SELECT CURRENT_DATE AT 5.75;
```

The above SELECT statements return the current date based on the time zone displacement, INTERVAL '05:45' HOUR TO MINUTE. If the current timestamp at UTC is TIMESTAMP '2008-06-01 17:30:00.000000+00:00', the SELECT statements would return '08/06/01' as the date.

If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, and the DBS Control flag TimeDateWZControl is enabled, the statement would return '08/06/02' as the current date based on the current session time and time zone displacement, INTERVAL '10:00' HOUR TO MINUTE.

The date returned is not adjusted to account for the start or end of daylight saving time.

Example 4

The following queries return the current date at the time zone displacement based on the time zone string, 'America Pacific'. Teradata Database determines the time zone displacement based on the time zone string and the CURRENT_TIMESTAMP AT '00:00' (that is, at UTC). The date returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT CURRENT_DATE AT TIME ZONE 'America Pacific';
SELECT CURRENT_DATE AT 'America Pacific';
```

Example 5: Changing the Default Output Format

To change the default output format of the CURRENT_DATE result, use Teradata explicit conversion syntax and specify the FORMAT phrase. For example, the following statement requests the current date and specifies a format that is different from the default:

```
SELECT CURRENT_DATE (FORMAT 'MMMBDD,BYYYY');
```

The result is similar to:

```
        Date
------------
May 31, 2007
```

For more information on Teradata explicit conversion syntax, see “Teradata Conversion Syntax in Explicit Data Type Conversions” on page 589. For more information on default data type formats and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

CURRENT_ROLE

Purpose

Returns the current role of the current authorized user.

Syntax

```
── CURRENT_ROLE ──
```

1101A565

ANSI Compliance

This is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for CURRENT_ROLE are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Current_Role

Result Value

If you are not accessing the Teradata Database through a proxy connection, CURRENT_ROLE functions exactly like the ROLE built-in function and returns the session current role, which is the current role of the session user. For details, see "ROLE" on page 372.

If you are accessing the Teradata Database through a proxy connection, then CURRENT_ROLE returns the current role of the proxy user as shown in the following table.

IF the current role for the session is ...	THEN the result value is ...
a role set by PROXYROLE	the name of the role.
the default	If there is one proxy role in the CONNECT THROUGH privilege of the proxy user, the result value is the name of the role. If there are multiple proxy roles in the CONNECT THROUGH privilege of the proxy user, the result value is ALL.
PROXYROLE=ALL	ALL
PROXYROLE=NONE or NULL	NULL

Usage Notes

CURRENT_ROLE is not supported in the FastLoad and MultiLoad utilities.

Example

You can identify the current role for the current authorized user with the following statement:

```
SELECT CURRENT_ROLE;
```

The system displays a response similar to the following:

```
Current_Role
------------------------------
Buyers_role
```

CURRENT_TIME

Purpose

Returns the current time.

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.
AT LOCAL	that the value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled. If TimeDateWZControl is disabled, the value returned is constructed from the time value local to the Teradata Database server and the session time zone.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

CURRENT_TIME and the AT clause are ANSI SQL:2011 compliant.

As an extension to ANSI, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Usage Notes

CURRENT_TIME returns the current time when the request started. If CURRENT_TIME is invoked more than once during the request, the same time is returned. The time returned does not change during the duration of the request.

If you specify CURRENT_TIME without the AT clause or CURRENT_TIME AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, CURRENT_TIME returns a time constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, CURRENT_TIME returns a time constructed from the time value local to the Teradata Database server and the session time zone.

For more information, see "DBS Control (dbscontrol)" in *Utilities*.

CURRENT_TIME returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- CURRENT_TIME is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- CURRENT_TIME is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

For more information about time zone strings, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

Note: If CURRENT_TIME is used in a stored procedure, the procedure must be recompiled whenever the DBS Control fields System TimeZone Hour or System TimeZone Minute are changed. Recompiling stored procedures is not necessary if a time zone string is set using the tdlocaledef utility. For details, see "CREATE PROCEDURE (SQL Form)" in *SQL Data Definition Language* and "DBS Control (dbscontrol)" in *Utilities*.

Result Type and Attributes

The result data type, format, and title for CURRENT_TIME are:

Data Type	Format	Title
TIME WITH TIME ZONE	Default format for the TIME WITH TIME ZONE data type. For more information on the default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.	Current Time (*fractional_precision*)

To convert CURRENT_TIME, use Teradata explicit conversion syntax or ANSI CAST syntax. For an example that uses Teradata explicit conversion syntax to change the default output format, see "Example 3: Changing the Default Output Format" on page 360.

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.

CURRENT_TIME Fields

The fields in CURRENT_TIME are:

- HOUR
- MINUTE
- SECOND
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

CURRENT_TIME versus TIME

CURRENT_TIME provides similar functionality to the Teradata function TIME using ANSI-compliant syntax. For information on the Teradata TIME function, see "TIME" on page 378.

Example 1: Requesting the Current Time

If the DBS Control flag TimeDateWZControl is enabled, the following statements request the current time based on the current session time and time zone.

```
SELECT CURRENT_TIME;
SELECT CURRENT_TIME AT LOCAL;
```

The result is similar to:

```
Current Time(0)
---------------
 15:53:34+00:00
```

If the session time zone was defined with a time zone string that follows different DST and standard time zone displacements, then the time returned is automatically adjusted to account for the start and end of daylight saving time. Otherwise, no adjustment for daylight saving time is done.

Example 2: Requesting the Current Time with a Time Zone String

The following queries return the current time at the time zone displacement based on the time zone string, 'America Pacific'. The time returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT CURRENT_TIME AT TIME ZONE 'America Pacific';
SELECT CURRENT_TIME AT 'America Pacific';
```

Example 3: Changing the Default Output Format

To change the default output format of the CURRENT_TIME result, use Teradata explicit conversion syntax and specify the FORMAT phrase. For example, the following statement requests the current time and specifies a format that is different from the default:

```
SELECT CURRENT_TIME (FORMAT 'HH:MIBT');
```

The result looks like this:

```
Current Time(0)
---------------
       02:29 PM
```

For more information on Teradata explicit conversion syntax, see "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 589. For more information on default data type formats and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

CURRENT_TIMESTAMP

Purpose

Returns the current timestamp.

Syntax

1101A715

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.
AT LOCAL	that the value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled. If TimeDateWZControl is disabled, the value returned is constructed from the time value local to the Teradata Database server and the session time zone.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

CURRENT_TIMESTAMP and the AT clause are ANSI SQL:2011 compliant.

As an extension to ANSI, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Usage Notes

CURRENT_TIMESTAMP returns the current timestamp when the request started. If CURRENT_TIMESTAMP is invoked more than once during the request, the same timestamp is returned. The timestamp returned does not change during the duration of the request.

If you specify CURRENT_TIMESTAMP without the AT clause or CURRENT_TIMESTAMP AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, CURRENT_TIMESTAMP returns a timestamp constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, CURRENT_TIMESTAMP returns a timestamp constructed from the time value local to the Teradata Database server and the session time zone.

For more information, see "DBS Control (dbscontrol)" in *Utilities*.

CURRENT_TIMESTAMP returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- CURRENT_TIMESTAMP is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- CURRENT_TIMESTAMP is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

For more information about time zone strings, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

Note: If CURRENT_TIMESTAMP is used in a stored procedure, the procedure must be recompiled whenever the DBS Control fields System TimeZone Hour or System TimeZone Minute are changed. Recompiling stored procedures is not necessary if a time zone string is set using the tdlocaledef utility. For details, see "CREATE PROCEDURE (SQL Form)" in *SQL Data Definition Language* and "DBS Control (dbscontrol)" in *Utilities*.

Result Type and Attributes

The result data type, format, and title for CURRENT_TIMESTAMP are:

Data Type	Format	Title
TIMESTAMP WITH TIME ZONE	Default format for the TIMESTAMP WITH TIME ZONE data type. For more information on the default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.	Current TimeStamp (*fractional_precision*)

To convert CURRENT_TIMESTAMP, use Teradata explicit conversion syntax or ANSI CAST syntax. For an example that uses Teradata explicit conversion syntax to change the default output format, see "Example 4: Changing the Default Output Format" on page 364.

Precision

The seconds precision of the result of CURRENT_TIMESTAMP is limited to hundredths of a second. CURRENT_TIMESTAMP returns zeros for any digits to the right of the two most significant digits in the fractional portion of seconds.

CURRENT_TIMESTAMP Fields

The fields in CURRENT_TIMESTAMP are:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

Example 1: Requesting the Current Timestamp

If the DBS Control flag TimeDateWZControl is enabled, the following statements request the current timestamp based on the current session time and time zone.

```
SELECT CURRENT_TIMESTAMP;
SELECT CURRENT_TIMESTAMP AT LOCAL;
```

The result is similar to:

```
             Current TimeStamp(6)
--------------------------------
2001-11-27 15:53:34.910000+00:00
```

If the session time zone was defined with a time zone string that follows different DST and standard time zone displacements, then the timestamp returned is automatically adjusted to account for the start and end of daylight saving time. Otherwise, no adjustment for daylight saving time is done.

Example 2: CURRENT_TIMESTAMP and the TimeDateWZControl Flag

This example shows the effect of the DBS Control flag TimeDateWZControl on the results returned by CURRENT_TIMESTAMP when the function is specified without an AT clause or with an AT LOCAL clause.

Assume the following:

- The time local to the Teradata Database server is 11:59:00 Coordinated Universal Time (UTC), January 31, 2010.
- User TK lives in Tokyo, and has a time zone defined as +9 hours offset from UTC.
- User LA lives in Los Angeles, and has a time zone defined as -8 hours offset from UTC.
- User TK and User LA run the CURRENT_TIMESTAMP function at exactly the same time.

If the TimeDateWZControl flag is enabled:

For User TK, the CURRENT_TIMESTAMP function returns:

```
2010-02-01 10:59:00.000000+09:00
```

For User LA, the CURRENT_TIMESTAMP function returns:

```
2010-01-31 16:59:00.000000-08:00
```

If the TimeDateWZControl flag is disabled:

For User TK, the CURRENT_TIMESTAMP function returns:

```
2010-01-31 11:59:00.000000+09:00
```

For User LA, the CURRENT_TIMESTAMP function returns:

```
2010-01-31 11:59:00.000000-08:00
```

Example 3: Requesting the Current Timestamp with a Time Zone String

The following queries return the current timestamp at the time zone displacement based on the time zone string, 'America Pacific'. The timestamp returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT CURRENT_TIMESTAMP AT TIME ZONE 'America Pacific';
SELECT CURRENT_TIMESTAMP AT 'America Pacific';
```

Example 4: Changing the Default Output Format

To change the default output format of the CURRENT_TIMESTAMP result, use Teradata explicit conversion syntax and specify the FORMAT phrase. For example, the following statement requests the current timestamp and specifies a format that is different from the default:

```
SELECT CURRENT_TIMESTAMP (FORMAT 'MMMBDD,BYYYYBHH:MIBT');
```

The result looks like this:

```
 Current TimeStamp(6)
---------------------
Feb 19, 2002 07:45 am
```

For more information on Teradata explicit conversion syntax, see "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 589. For more information on default data type formats and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

CURRENT_USER

Purpose

Provides the user name of the current authorized user.

Syntax

```
── CURRENT_USER ──
```

1101A564

ANSI Compliance

This is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for CURRENT_USER are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Current_User

Result Value

If you are accessing the Teradata Database through a proxy connection, CURRENT_USER returns the proxy user name. Otherwise, it functions exactly like the USER built-in function and returns the session user name. For details, see "USER" on page 381.

Example 1

You can identify the current authorized user with the following statement:

```
SELECT CURRENT_USER;
```

The system responds with something like the following:

```
Current_User
------------------------------
BO-JSMITH
```

Example 2

The following example selects the job title for the current authorized user:

```
SELECT JobTitle FROM Employee WHERE Name = CURRENT_USER;
```

DATABASE

Purpose

Returns the name of the default database for the current user.

Syntax

```
─── DATABASE ───
```

FF07R002

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for DATABASE are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Database

Usage Notes

If a DATABASE request has changed the current default database, then the DATABASE function returns the new name of the default.

Example

The following statement requests the name of the default database:

```
SELECT DATABASE;
```

The system responds with something like the following:

```
Database
------------------------------
Customer_Service
```

DATE

Purpose

Returns the current date.

Syntax

where:

Syntax element ...	Specifies ...
AT LOCAL	that the value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled. If TimeDateWZControl is disabled, the value returned is constructed from the time value local to the Teradata Database server and the session time zone.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

For the ANSI-compliant syntax and behavior for the equivalent function, see "CURRENT_DATE" on page 351.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, you can specify an AT clause after the DATE function, and you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Usage Notes

DATE returns the current date at the time when the request started. If DATE is invoked more than once during the request, the same date is returned. The date returned does not change during the duration of the request.

If you specify DATE without the AT clause or DATE AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, DATE returns a date constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, DATE returns a date constructed from the time value local to the Teradata Database server and the session time zone.

For more information, see "DBS Control (dbscontrol)" in *Utilities*.

DATE returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- DATE is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- DATE is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

For more information about time zone strings, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

DATE cannot appear as the first argument in a user-defined method invocation.

Result Type and Attributes

The default format of DATE depends on the value of the Dateform mode. For more information on default data type formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*. The data type, format, and title for DATE are as follows:

Data Type	Format		Title
	Dateform Mode	DATE Function Format	
DATE	INTEGERDATE	the default format for DATE data types as specified in the SDF.	Date
	ANSIDATE	'YYYY-MM-DD'	

DATE versus CURRENT_DATE

DATE is deprecated. Use the ANSI SQL:2011 compliant CURRENT_DATE function instead. See "CURRENT_DATE" on page 351.

Example 1

This example assumes that the default format for DATE values is 'yy/mm/dd'. Consider the following statements:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT DATE AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;
SELECT DATE AT INTERVAL -'08:00' HOUR TO MINUTE;
SELECT DATE AT TIME ZONE INTERVAL -'08' HOUR;
SELECT DATE AT INTERVAL -'08' HOUR;
SELECT DATE AT TIME ZONE '-08:00';
SELECT DATE AT '-08:00';
SELECT DATE AT TIME ZONE '-8';
SELECT DATE AT '-8';
SELECT DATE AT TIME ZONE -8;
SELECT DATE AT -8;
SELECT DATE AT -8.0;
```

The above SELECT statements return the current date based on the time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. If the current timestamp at UTC is TIMESTAMP '2008-06-01 06:30:00.000000+00:00', these SELECT statements would return '08/05/31' as the date.

If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, and the DBS Control flag TimeDateWZControl is enabled, the statement would return '08/06/01' as the current date based on the current session time and time zone displacement, INTERVAL '01:00' HOUR TO MINUTE. For example:

```
SELECT DATE;
SELECT DATE AT LOCAL;
```

The date returned is not adjusted to account for the start or end of daylight saving time.

Example 2

The following queries return the current date at the time zone displacement based on the time zone string, 'America Pacific'. Teradata Database determines the time zone displacement based on the time zone string and the CURRENT_TIMESTAMP AT '00:00' (that is, at UTC). The date returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT DATE AT TIME ZONE 'America Pacific';
SELECT DATE AT 'America Pacific';
```

Example 3

Use the FORMAT phrase to change the presentation:

```
SELECT DATE (FORMAT 'mm-dd-yy');
    Date
--------
03-30-96
```

Example 4

Another form gives:

```
SELECT DATE (FORMAT 'mmmbdd,byyyy');
        Date
------------
Mar 30, 1996
```

PROFILE

Purpose

Returns the current profile for the session or NULL if none.

Syntax

```
── PROFILE ──
```

KZ01A006

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for PROFILE are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Profile

Example

You can identify the current profile for the session with the following statement:

```
SELECT PROFILE ;
```

ROLE

Purpose

Returns the session current role.

Syntax

```
── ROLE ──
```

KZ01A007

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for ROLE are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	Role

Result Value

The session logon can be not directory-based or directory-based.

If you are accessing the Teradata Database through a proxy connection, and you want to get the current role of the proxy user, use the CURRENT_ROLE built-in function. For details, see "CURRENT_ROLE" on page 355.

Session logon is not directory-based

If the session logon is not directory-based, refer to the following table.

Current Role for the Session	Result Value
Existing role	Name of the role
ALL	'ALL'
NONE or NULL	NULL

Session logon is directory-based

If the session logon is directory-based, refer to the following table.

Session	Result Value
Assigned a set of directory-managed roles and does not change the current role	'EXTERNAL'
Uses a SET ROLE EXTERNAL statement	
• does *not* have an assigned set of directory-managed roles, • maps to a permanent user that has a default database-managed role, and • does not change the current role	Name of the default role of the permanent user
Uses a SET ROLE *role_name* statement, where *role_name* is either a directory-managed or database-managed role	Name of the specified role
Uses a SET ROLE ALL statement	'ALL'

• *Not* assigned a set of directory-managed roles,
• Does not change the current role, and one of the following condition is true"
  • Directory-based logon does not map to a permanent user
  • Permanent user that the directory-based logon maps to does not have a default database-managed role
	NULL

Uses a SET ROLE NONE statement	
Uses a SET ROLE NULL statement	

Usage Notes

ROLE is not supported in the FastLoad and MultiLoad utilities.

Example

You can identify the session current role with the following statement:

```
SELECT ROLE;
```

The system responds with something like the following:

```
Role
------------------------------
EXTERNAL
```

SESSION

Purpose

Returns the number of the session for the current user.

Syntax

```
─── SESSION ───
```

FF07R003

ANSI Compliance

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

Result Type and Attributes

The data type, format, and title for SESSION are as follows:

Data Type	Format	Session
INTEGER	Default format for the INTEGER data type. For more information on the default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.	Session

Example

The following statement identifies the number of the session for the current user:

```
SELECT SESSION;
```

The system responds with something like the following:

```
    Session
-----------
       1048
```

TEMPORAL_DATE

Purpose

Returns the current transaction date where the evaluation is based on the session time zone.

Syntax

```
TEMPORAL_DATE
```

1182A008

Result Type and Attributes

The result data type, format, and title for TEMPORAL_DATE are as follows:

Data Type	Format	Title
DATE	Default format for the DATE data type when the Dateform mode is set to IntegerDate. For details on default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.	Date

Usage Notes

The value of TEMPORAL_DATE is the same for all requests submitted in a single transaction.

The system uses the session time zone to evaluate TEMPORAL_DATE.

When TEMPORAL_DATE appears in a CHECK constraint or DEFAULT clause, the result value is evaluated when the request applies the CHECK constraint (during an insert or update) or when the request uses the DEFAULT value for a given column.

For information on using TEMPORAL_DATE with temporal tables, see *Temporal Table Support*.

Restrictions

TEMPORAL_DATE is not supported in a partitioning expression for the PARTITION BY clause that defines a partitioned primary index.

TEMPORAL_TIMESTAMP

Purpose

Returns the current transaction timestamp where the evaluation is based on the session time zone.

Syntax

```
TEMPORAL_TIMESTAMP ──┬──────────────┬──
                     └─( precision )─┘
```

1182A009

where:

Syntax element ...	Specifies ...
precision	an optional precision range for the returned timestamp value. The valid range is 0 through 6, inclusive. The default is 6.

Result Type and Attributes

The result data type, format, and title for TEMPORAL_TIMESTAMP are as follows:

Data Type	Format	Title
TIMESTAMP(*n*) WITH TIME ZONE, where *n* is the same as the *precision* argument or 6 if omitted	Default format for the TIMESTAMP WITH TIME ZONE type. For details on default formats, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.	Timestamp

Usage Notes

The value of TEMPORAL_TIMESTAMP is the same for all requests submitted in a single transaction.

The system uses the session time zone to evaluate TEMPORAL_TIMESTAMP.

When TEMPORAL_TIMESTAMP appears in a CHECK constraint or DEFAULT clause, the result value is evaluated when the request applies the CHECK constraint (during an insert or update) or when the request uses the DEFAULT value for a given column.

For information on using TEMPORAL_TIMESTAMP with temporal tables, see *Temporal Table Support*.

Precision

The seconds precision of the result of TEMPORAL_TIMESTAMP is limited to hundredths of a second. TEMPORAL_TIMESTAMP returns zeros for any digits to the right of the two most significant digits in the fractional portion of seconds.

TIME

Purpose

Returns the current time.

Syntax

where:

Syntax element ...	Specifies ...
AT LOCAL	that the value returned is constructed from the session time and session time zone if the DBS Control flag TimeDateWZControl is enabled. If TimeDateWZControl is disabled, the value returned is constructed from the time value local to the Teradata Database server and the session time zone.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

For the ANSI-compliant syntax and behavior for the equivalent function, see "CURRENT_TIME" on page 357.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Usage Notes

TIME returns the current time when the request started. If TIME is invoked more than once during the request, the same time is returned. The time returned does not change during the duration of the request.

If you specify TIME without the AT clause or TIME AT LOCAL, then the value returned depends on the setting of the DBS Control flag TimeDateWZControl as follows:

- If the TimeDateWZControl flag is enabled, TIME returns a time constructed from the session time and session time zone.
- If the TimeDateWZControl flag is disabled, TIME returns a time constructed from the time value local to the Teradata Database server and the session time zone.

For more information, see "DBS Control (dbscontrol)" in *Utilities*.

TIME data is stored internally in UTC, which can affect how TIME result values sort. For more information, see "ORDER BY Clause" in *SQL Data Manipulation Language*.

TIME returns a value that is adjusted to account for the start and end of daylight saving time (DST) only in the following cases:

- TIME is specified with AT [TIME ZONE] *time_zone_string*, where *time_zone_string* follows different DST and standard time zone displacements.
- TIME is specified with AT LOCAL or without an AT clause and the session time zone was defined with a time zone string that follows different DST and standard time zone displacements.

For more information about time zone strings, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

TIME cannot appear as the first argument in a user-defined method invocation.

Result Type and Attributes

The data type, format, and title for TIME are as follows:

Data Type	Format	Title
FLOAT	HHMMSS.CC (hours, minutes, seconds, hundredths of a second)	Time

TIME versus CURRENT_TIME

TIME is deprecated. Use the ANSI SQL:2011 compliant CURRENT_TIME function instead. See "CURRENT_TIME" on page 357.

Example 1

If the DBS Control flag TimeDateWZControl is enabled, the following statements request the current time based on the current session time and time zone.

```
SELECT TIME;
SELECT TIME AT LOCAL;
```

The result is similar to:

```
    Time
--------
16:20:20
```

If the session time zone was defined with a time zone string that follows different DST and standard time zone displacements, then the time returned is automatically adjusted to account for the start and end of daylight saving time. Otherwise, no adjustment for daylight saving time is done.

Example 2

The following queries return the current time at the time zone displacement based on the time zone string, 'America Pacific'. The time returned is automatically adjusted to account for the start and end of daylight saving time.

```
SELECT TIME AT TIME ZONE 'America Pacific';
SELECT TIME AT 'America Pacific';
```

Example 3

The hundredths of a second are not displayed by the default format, but you can use the FORMAT phrase to display it:

```
SELECT TIME (FORMAT '99:99:99.99');
```

The system responds with something like the following:

```
       Time
-----------
16:26:30.19
```

Example 4

The following example inserts a row in a hypothetical table in which the column InsertTime has data type FLOAT and records the time that the row was inserted:

```
INSERT INTO HypoTable (ColumnA, ColumnB, InsertTime)
VALUES ('Abcde', 12345, TIME);
```

USER

Purpose

Provides the session user name.

Syntax

```
──USER──
```

FF07D272

ANSI Compliance

This is ANSI SQL:2011 compliant.

Result Type and Attributes

The data type, format, and title for USER are as follows:

Data Type	Format	Title
VARCHAR(30) CHARACTER SET UNICODE	X(30)	User

Result Value

IF the session logon is ...	THEN ...
not directory-based	the result value is the session user name.
directory-based	If the session maps to a permanent table, the result value is the name of the permanent user. If the session does not map to a permanent user, the result value is the authcid of the external user.

If you are accessing the Teradata Database through a proxy connection, and you want to get the name of the proxy user, use the CURRENT_USER built-in function. For details, see "CURRENT_USER" on page 365.

Example 1

You can identify the session user name with the following statement:

```
SELECT USER;
```

The system responds with something like the following.

```
User
------------------------------
JJ43901
```

Example 2

The following example selects the job title for the session user.

```
SELECT JobTitle FROM Employee WHERE Name = USER;
```

CHAPTER 8 Business Calendars

This chapter describes the system-defined business calendars, the macros you can use to manipulate them, the business calendar tables and views, and the embedded services system functions that support the system-defined business calendars.

About the Business Calendars

A business calendar defines business and non-business days. The significance of a day being a business day or a non-business day is user-determined. For example, a business day could be a work day, and a non-business day could be either a non-working day, a weekend day, a holiday, or a vacation day. You can define different week patterns (weekdays and weekends) and exceptions (holidays and business open and closed days) for the system-defined calendars.

There are three Teradata system-defined business calendars that you can set for your session:

- Teradata
- ISO
- COMPATIBLE

All three calendars are based on the de facto international standard, the Gregorian calendar. The Gregorian calendar has 365 days in most years and 366 days in a leap year. The calendars differ in how they define weeks and whether they allow partial weeks. You can use macros to specify weekday/weekend patterns and exceptions to the patterns.

The calendars support January 1, 1900, to December 31, 2100. The default session calendar is Teradata. Each calendar defaults to all business days. You can change that pattern using a macro. See "About Business Calendar Macros" on page 386.

Calendar Differences

For This Calendar...	The First Full Week Begins...
Teradata	on Sunday. The days of the year before Sunday belong to Week 0. For example, if the year starts on January 1, 2004 (a Thursday), then Week 0 is from January 1 to January 3. Week 1 begins on Sunday, January 4.

For This Calendar...	The First Full Week Begins...
ISO	on Monday. The first week of the year is the first week that has at least 4 days. If a week has fewer than 4 days, it belongs to the last week of the previous year. There are no partial weeks. This calendar follows the ISO and European standard. For example, if the year starts on January 1, 2008 (a Tuesday) and the week start is Monday, week 1 of 2008 is from December 31, 2007, to January 6, 2008.
COMPATIBLE	on January 1. regardless of what day of the week that is. There can be partial weeks with 1 day (for most years) or 2 days (for leap years) at the end of the year. The day the week begins can change from year to year. This calendar is Oracle-compatible. For example, if January 1, 2011, is a Saturday, the first week of the year is from Saturday, January 1, 2011, through Friday, January 7, 2011.

About ISO Computation

The way in which a week, month, quarter, and year are defined in the ISO calendar is different from the Teradata and COMPATIBLE calendars. ISO has only complete weeks. All the business calendar functions and views use the following rules for ISO computation.

Week

An ISO week always has 7 days. There are no partial weeks. The ISO week always starts on Monday and ends on Sunday.

Year

Each year has 52 or 53 complete weeks. The year start or end is determined by applying the ISO Thursday rule to the week at the year border. If Thursday falls in the old year in the Gregorian calendar, the border week becomes part of the old year in the ISO calendar. If Thursday falls in the new year in the Gregorian calendar, the border week becomes part of the new year in the ISO calendar. For example, December 31, 2009, is a Thursday. Therefore, in the ISO calendar the week of Monday, December 28, 2009, through Sunday, January 3, 2010, is the last week in 2009. The first week of 2010 in the ISO calendar begins on Monday, January 4, 2010. In contrast, the Teradata calendar for 2010 begins on Friday, January 1.

2009 Dec last week ISO

Dec 2009 Calendar						
S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

2010 Jan first week ISO

Jan 2010 Calendar						
S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Month

Each month has 4 or 5 complete weeks. The month start or end is determined by applying the ISO Thursday rule to the week at the month border. For example, in the ISO calendar, the month of October 2012 ends on Sunday, October 28, because Thursday in the border week falls on November 1. The new week that begins on Monday, October 29, becomes part of November.

2012 Oct last week ISO

Oct 2012 Calendar						
S	M	T	W	Th	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3

2012 Nov first week ISO

Nov 2012 Calendar						
S	M	T	W	Th	F	S
	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Quarter

The ISO year is divided into 4 quarters with 13 weeks per quarter. There are 14 weeks in the last quarter if the year has 53 weeks. Quarters are divided into 3 months with a pattern of 4,4,5 or 4,5,4 or 5,4,4 weeks. If the year has 53 weeks, the last quarter has 3 months with a pattern of 4,5,5 or 5,4,5 or 5,5,4 weeks. The quarter start or end is determined by applying the ISO Thursday rule to the week at the month border in the last month of the quarter. The week at the month border belongs to the month that has Thursday of that week. For example, Friday, January 1, 2016, is part of a week in which Thursday falls in the last week and last quarter of 2015. Therefore, Friday, January 1, 2016, belongs to last quarter of 2015 in the ISO calendar and the first quarter of 2016 in the Teradata calendar.

2015 Dec last week of quarter ISO

Dec 2015 Calendar						
S	M	T	W	Th	F	S
		1	2	33	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3						

2016 Jan first week of quarter ISO

Jan 2016 Calendar						
S	M	T	W	Th	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

About Business Calendar Macros

System business macros enable administrators and users to configure the system business calendars (Teradata, ISO, and COMPATIBLE) by:

- Creating the pattern for the calendar.
- Creating exceptions for the calendar.
- Deleting existing exceptions from the calendar.

These macros are created when the DIP script is executed and reside in the DBC database.

Note: To view any of the macros, use the DIP script.

Macro Name	Purpose
CreateBusinessCalendarPattern	Creates the pattern for a business calendar.
CreateException	Defines an exception list for a specific business calendar. The list identifies all of the days of the calendar that are exceptions to the pattern defined for the calendar.
DeleteException	Deletes a particular exception from a specific business calendar.
DeleteAllExceptions	Deletes all of the exceptions from a specific business calendar.

CreateBusinessCalendarPattern

You can use this macro to create the pattern for a business calendar. The pattern is the template for business days and non-business days for the entire calendar and the basic structural unit on which the calendar is based (for example, week, month, quarter or year).

Note: Currently, only the weekly pattern is supported. You can define each day of the week as a business day or non-business day. To make particular business days non-business days (for example, a holiday), you can use the CreateException macro (see "CreateException" on page 388).

When you execute the macro, the pattern for the calendar is created and the information about the pattern is stored in the DBC.BusinessCalendarPattern table.

Required Parameters

Parameter	Data Type	Format	Description
CalendarName	VARCHAR(256) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL	X(128)	Name of the business calendar. For the system business calendars, it must be Teradata, ISO, or COMPATIBLE.
DayName	CHAR(10)	X(10)	Name of the calendar day. Must be SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, or SATURDAY.
Pattern	CHAR(5)	X(5)	Pattern for the calendar day. Must be ON (working day) or OFF (non-working day).
PatternComment	VARCHAR(2048)	X(1024)	Reason or rationale for the pattern for the calendar day (ON or OFF).

Example 1

In the following, Saturday and Sunday of each week of the Teradata business calendar are defined as non-working days (non-working days have the OFF status) and this information is added to the DBC.BusinessCalendarPattern table.

Note: By default, each day of week for the Teradata business calendar is a working day.

```
EXEC DBC.CreateBusinessCalendarPattern
   ('TERADATA', ‘SATURDAY’,’OFF’, ‘Non-work Day’);

EXEC DBC.CreateBusinessCalendarPattern
   ('TERADATA', ‘SUNDAY’,’OFF’, ‘Non-work Day’);
```

Example 2

The following modifies the ISO business calendar by designating SUNDAY and MONDAY as working days:

```
EXEC DBC.CreateBusinessCalendarPattern
   ('ISO', 'SUNDAY','ON', 'Work Day');

EXEC DBC.CreateBusinessCalendarPattern
  ('ISO', 'MONDAY', 'ON', 'Work Day');
```

CreateException

You can use this macro to define an exception list for a specific business calendar. The list identifies all of the days (by date) of the calendar that are exceptions to the pattern defined for the calendar.

When you execute the macro, the exceptions in the list are inserted into the calendar and into the DBC.BusinessCalendarException table.

If you try to define an exception for a particular date that is already an exception, an error is returned. To replace the existing exception, you must first delete the existing exception then redefine a new one for the date.

Required Parameters

Parameter	Format	Data Type	Description
CalendarName	VARCHAR(256) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL	X(128)	Name of the business calendar. For the system business calendars, it must be Teradata, ISO, or COMPATIBLE.
ExceptionIndicator	CHAR(5)	X(5)	Type of day for the exception. Must be ON (business day) or OFF (non-business day).
ExceptionDate	DATE	YYYY-MM-DD	Date of the exception (not the day it was created).
PatternComment	VARCHAR(2048)	X(1024)	Reason or rationale for the pattern for the calendar day (ON or OFF).

Usage Notes

If you try to insert exceptions that are beyond the CalendarPeriod boundaries, the insertion is aborted.

If you try to insert an exception for a day that is already defined as an exception, you must delete the existing exception for that day and replace it with the new exception.

Example

In this example, an OFF and an ON exception are defined for the Teradata business calendar and the information about them is added to the DBC.BusinessCalendarException table. Days with the OFF status are non-business days. Days with the ON status are business days.

```
EXEC DBC.CreateException
   ('Teradata', ’OFF’, DATE '2008-03-03', 'Holiday');

EXEC DBC.CreateException
  ('Teradata', ’ON’, DATE '2008-06-07', 'Holiday make up day');
```

The two exceptions are inserted into the Teradata business calendar. This first exception indicates that March 3, 2008 is now a non-business day (noted in the comment as a holiday). The second exception indicates that June 7, 2008 is now a business day (noted in the comment as a day to make up for a previous holiday).

The CalendarPeriod of the Teradata calendar is from January 1, 1900 to December 31, 2100. If you try to insert holidays beyond the CalendarPeriod boundary, the insertion is aborted.

DeleteException

You can use this macro to delete a particular exception from a specific business calendar. When you delete an exception, the status for that exception day automatically reverts to the status defined for it in the pattern for the calendar.

When you execute the macro, the exception is removed from the calendar and from the DBC.BusinessCalendarException table.

If you need to delete all of the exceptions from a calendar, use the DeleteAllExceptions macro (see "DeleteAllExceptions" on page 389).

Required Parameters

Parameter	Data Type	Format	Description
CalendarName	VARCHAR(256) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL	X(128)	The name of the business calendar. For the system business calendars, it must be Teradata, ISO, or COMPATIBLE.
ExceptionDate	DATE	YY/MM/DD	The date of the exception (not the day it was created).

Example

In this example, the exception with the date of January 1, 2009 is deleted from the ISO business calendar and from the DBC.BusinessCalendarException table.

```
EXEC DBC.DeleteException('ISO', DATE '2009-01-01');
```

DeleteAllExceptions

You can use this macro to delete all of the exceptions from a specific business calendar. When you delete all exceptions, the status for all exception days automatically reverts to the status defined for the days in the pattern for the calendar.

When you execute the macro, all exceptions are removed from the calendar and from the DBC.BusinessCalendarException table.

If you need to delete a particular exception from a calendar, use the DeleteException macro (see "DeleteException" on page 389).

Required Parameter

Parameter	Data Type	Format	Description
CalendarName	VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL	X(128)	The name of the business calendar. For the system business calendars, it must be Teradata, ISO, or COMPATIBLE.

Example

In this example, all of the exceptions are removed from the ISO business calendar and from the DBC.BusinessCalendarException table.

```
EXEC DBC.DeleteAllExceptions('ISO');
```

About the Business Calendar Tables

You can query business calendar tables to see calendar information relevant to your business.

DBC.BusinessCalendarPattern Table

This table records calendar information about patterns of business and non-business days.

Column Name	Column Description
CalendarName	Business calendar name.
PatternType	Identifies the pattern as weekly, monthly, quarterly, or yearly. Currently, only the weekly pattern is supported. Pattern “W” indicates a weekly pattern.
DayNumber	The days in the week are numbered from 1 to 7, starting with Sunday as 1 and ending with Saturday as 7. There is one row in the table for each day, and the ON/OFF status for the day is stored in the Pattern column.
Pattern	A weekly pattern describes 7 rows, each set to 1 or 0. For each DayNumber, this column stores the status, whether it is a business day (1) or a non-business day (0).
PatternComment	A description of the pattern.
CreatorName	The name of the user who created or modified the pattern. For system-defined business calendars, this is DBC.
LastModified	The timestamp of the last pattern change.

DBC.BusinessCalendarException Table

This table contains a list of days that are exceptions to the pattern of working and non-working days.

Column Name	Column Description
CalendarName	Business calendar name to which the exception applies.
ExceptionIndicator	A value of 0 indicates that an ExceptionDate is a non-business day, and a value of 1 indicates that an ExceptionDate is a business day.
ExceptionDate	An ExceptionDate must fall within the Teradata calendar (January 1, 1900, to December 31, 2100). If the ExceptionDate is outside that period, it is not accepted.
ExceptionReason	The reason for the exception. For example, December 25, 2012 (a Tuesday) is normally a working day, but it is also the Christmas holiday, so the ExceptionReason is "Christmas." The ExceptionReason may be NULL.
CreatorName	The name of the user who created the exception.
CreationTime	The timestamp of the exception creation.

There are two types of exceptions:

- An OFF exception: A typical working day is a day off (for example, Monday, September 5, 2011, is the Labor Day holiday).
- An ON exception: A typical day off is a working day (for example, everyone at your job is required to work on Saturday, July 30, 2011).

You do not need to validate whether an OFF-exception falls on a business day and an ON-exception falls on a non-business day. Just set the exception, and specify an ExceptionReason. For example, Monday, July 4, is a holiday, but the Barbeque Bonanza store is open and your ExceptionReason is Independence Day Sale.

About the Business Calendar Views

You can query business calendar views to retrieve calendar information relevant to your business, such as the first business day of a week.

Sys_Calendar.BusinessCalendarExceptions

This view provides information about exceptions defined for a calendar, such as a holiday on Monday, which is normally a working day. You can query the view to check the exceptions.

View Column Name	Description	Data Type	Format
CalendarName	The name of the calendar.	VARCHAR(128)	X(128)
ExceptionIndicator	Returns ON for a working day and OFF for a nonworking day.	VARCHAR(3)	X(3)
ExceptionDate	The exception date set for the calendar.	DATE	YY/MM/DD
ExceptionReason	The reason for the exception.	VARCHAR(1024)	X(1024)
CreatorName	The user who created the exception.	VARCHAR(128)	X(128)
CreationTime	The timestamp of exception creation.	TIMESTAMP(0)	YYYY-MM-DDBHH:MI:SS

Sys_Calendar.BusinessCalendarPatterns

This view provides information about patterns defined for a calendar, such as a weekday/weekend pattern.

View Column Name	Description	Data Type	Format
CalendarName	The name of the calendar.	VARCHAR(128)	X(128)
DayName	The day name, for example, Friday	VARCHAR(9)	X(9)
Pattern	The ON or OFF pattern settings.	VARCHAR(3)	X(3)
PatternComment	The comment for each day in the pattern.	VARCHAR(1024)	X(1024)
CreatorName	The user who created the pattern.	VARCHAR(128)	X(128)
LastModified	The timestamp of last pattern change.	TIMESTAMP(0)	YYYY-MM-DD:HH:MI:SS

Sys_Calendar.BusinessCalendar

This Sys_Calendar view provides business functionality for the three system-defined business calendars.

View Column Name	Description	Data Type	Format
Calendar_Date	The date.	DATE	YY/MM/DD
Day_of_Week	An integer value that ranges from 1 to 7.	INTEGER	-(10)9
Day_of_Month	A integer value that ranges from 1 to 31.	INTEGER	-(10)9
Day_of_Year	An integer value that ranges from 1 to 366.	INTEGER	-(10)9
Day_of_Calendar	The number of days since the beginning of the calendar.	INTEGER	-(10)9
Weekday_of_Month	The *n*th occurrence of the weekday in the month (1-5).	INTEGER	-(10)9

View Column Name	Description	Data Type	Format
Week_of_Month	The week number of the month, ranging from 0 to 5.	INTEGER	-(10)9
Week_of_Quarter	The week number of the quarter, ranging from 0 to 14.	INTEGER	-(10)9
Week_of_Year	The week number of the year, ranging from 0 to 53.	INTEGER	-(10)9
Week_of_Calendar	For a given date, the number of the week in the calendar in which it occurs.	INTEGER	-(10)9
Month_of_Quarter	For a given date, the number of the month in the quarter in which it occurs.	INTEGER	-(10)9
Month_of_Year	For a given date, the number of the month in the year in which it occurs.	INTEGER	-(10)9
Month_of_Calendar	For a given date, the number of the month in the calendar in which it occurs.	INTEGER	-(10)9
Quarter_of_Year	For a given date, the quarter of the year in which it occurs.	INTEGER	-(10)9
Quarter_of_Calendar	For a given date, the quarter number of the calendar in which it occurs.	INTEGER	-(10)9
Year_of_Calendar	For a given date, the year number of the calendar in which it occurs.	INTEGER	-(10)9
WeekEnd	The end of the week for the given date.	DATE	YY/MM/DD
WeekBegin	The beginning of the week for the given date.	DATE	YY/MM/DD
MonthBegin	The beginning of the month for the given date.	DATE	YY/MM/DD
MonthEnd	The end of the month for the given date.	DATE	YY/MM/DD
QuarterBegin	The beginning of the quarter for the given date.	DATE	YY/MM/DD
QuarterEnd	The end of the quarter for the given date.	DATE	YY/MM/DD
YearBegin	The beginning of the year for the given date.	DATE	YY/MM/DD
YearEnd	The end of the year for the given date.	DATE	YY/MM/DD
IsBusinessDay	Whether or not the given day is a business day.	BYNET	-(3)9
BusinessWeekBegin	The first working day of the week in which the given date occurs.	DATE	YY/MM/DD
BusinessWeekEnd	The last working day of the week in which the given date occurs.	DATE	YY/MM/DD

View Column Name	Description	Data Type	Format
BusinessMonthBegin	The first working day of the month in which the given date occurs.	DATE	YY/MM/DD
BusinessMonthEnd	The last working day of the month in which the given date occurs.	DATE	YY/MM/DD
BusinessQuarterBegin	The first working day of the quarter in which the given date occurs.	DATE	YY/MM/DD
BusinessQuarterEnd	The last working day of the quarter in which the given date occurs.	DATE	YY/MM/DD
BusinessYearBegin	The first working day of the year in which the given date occurs.	DATE	YY/MM/DD
BusinessYearEnd	The last working day of the year in which the given date occurs.	DATE	YY/MM/DD

Example 1

This query returns the day of the week using the ISO calendar:

```
Set session calendar = iso;
Sel day_of_week from Sys_Calendar.Calendar where calendar_date = date
'2011-01-01';

day_of_week
-----------
6
```

Example 2

This query returns the day of the week using the COMPATIBLE calendar:

```
Set session calendar = compatible;

Sel day_of_week from Sys_Calendar.Calendar where calendar_date = date
'2011-01-01';

day_of_week
-----------
1
```

Example 3

This query returns the beginning of the week using the ISO calendar:

```
Set session calendar = iso;

Sel weekBegin from Sys_Calendar.BusinessCalendar where calendar_date =
date '2011-01-01';

WeekBegin
-----------
10/12/27
```

Example 4

This query returns the beginning of the week using the COMPATIBLE calendar:

```
Set session calendar = compatible;
Sel weekBegin from Sys_Calendar.BusinessCalendar where calendar_date =
date '2011-01-01';
WeekBegin
-----------
11/01/01
```

Example 5

By default, Monday is a working day in the ISO calendar. The following query adds an OFF exception for January 1, 2007, which is a Monday.

```
Exec DBC.CreateException('ISO', 'OFF', date '2007-01-01',  'Newyear
Day');
```

The following query returns the next working day after January 1, 2007:

```
Sel BusinessWeekBegin from Sys_Calendar.BusinessCalendar where
calendar_date = date '2007-01-01';
BusinessWeekBegin
-----------
07/01/02
```

Example 6

This query returns the beginning of the week and the week number for 2009, assuming that no calendar is set for the session. The session uses the default business calendar "Teradata."

```
SEL WeekBegin, week_of_year FROM Sys_Calendar.BusinessCalendar WHERE
calendar_date = DATE '2009-03-15';
Week_begin         week_of_year
-----------        -------------------
09/03/15           11
```

About the Business Calendar Functions

You can specify business calendar functions in an SQL statement wherever you specify UDFs, for example, in an INSERT, UPDATE, DELETE, MERGE, or SELECT statement.

The following facts apply to the business calendar functions:

- If you do not specify a calendar name, the calendar defaults to the session calendar.
- All functions related to a year, for example MonthNumber_Of_Year, are calculated relative to January 1 of that year.
- All functions related to a calendar, for example, DayNumber_Of_Calendar, are calculated relative to the beginning of the calendar, 1900-01-01.
- You can specify these functions anywhere in a DML statement, in CHECK CONSTRAINTS in the DDL statements, and all other places in an SQL statement where a UDF can be specified.

- All the definitions of the business calendar functions are stored in the TD_SYSFNLIB database. The format and title of the business calendar functions used in a SELECT statement follow the regular UDF style.
- These functions are available to all Teradata users and do not require any privileges.
- All the computations inside the functions are in UTC if the input is of type TIMESTAMP or TIMESTAMP WITH TIME ZONE.

Prerequisites for Using the Business Calendar Functions

Before you can use these functions, you must run the Database Initialization Program (DIP) utility and execute the DIPSYSFNC script. The DIPALL or DIPSYSFNC script will create the calendar functions in the TD_SYSFNLIB database. For more information about the DIP utility, see *Utilities*.

If you have a UDF with the same name as a business calendar function, you must remove that UDF from the normal UDF search path before you can invoke the business calendar function. If the business calendar function is not found in the current database, Teradata Database searches for the function in the TD_SYSFNLIB database. Alternatively, you may invoke the calendar function by using the fully qualified syntax, TD_SYSFNLIB.*calendar_function_name*.

Example 1: Cumulative Sales During First Week of All Quarters

This example finds cumulative sales of an item during first week of all quarters. Results are shown for each calendar setting.

```
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
GROUP BY Item_Code;
```

SET SESSION calendar = Teradata;

Item_Code	SUM(Sale_Amt)
101	225
102	120

SET SESSION calendar = ISO;

Item_Code	SUM(Sale_Amt)
101	135
102	30

SET SESSION calendar = Compatible

Item_Code	SUM(Sale_Amt)
101	245
102	100

Sales_Tbl		
Item_Code	**Sale_Amt**	**Sale_Date**
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example 2: Cumulative Sales During First Week of All Quarters - Teradata Calendar

This example shows how cumulative sales for the first week of all quarters are totaled using the Teradata calendar.

```
SET SESSION calendar = Teradata;
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
GROUP BY Item_Code;
```

SET SESSION calendar = Teradata;

Item_Code	SUM(Sale_Amt)
101	225
102	120

Sales_Tbl		
Item_Code	**Sale_Amt**	**Sale_Date**
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example 3: Cumulative Sales During First Week of All Quarters - ISO Calendar

This example shows how cumulative sales for the first week of all quarters are totaled using the ISO calendar.

```
SET SESSION calendar = ISO;
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
GROUP BY Item_Code;
```

SET SESSION calendar = ISO;

Item_Code	SUM(Sale_Amt)
101	135
102	30

Sales_Tbl		
Item_Code	**Sale_Amt**	**Sale_Date**
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example 4: Cumulative Sales During First Week of All Quarters - Compatible Calendar

This example shows how cumulative sales for the first week of all quarters are totaled using the Compatible calendar.

```
SET SESSION calendar = Compatible;
SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
GROUP BY Item_Code;
```

SET SESSION calendar = Compatible;

Item_Code	SUM(Sale_Amt)
101	245
102	100

Sales_Tbl		
Item_Code	Sale_Amt	Sale_Date
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example 5: Cumulative Weekday Sales During First Week of All Quarters - Teradata Calendar

This example finds cumulative sales of an item during weekdays of the first week of all quarters.

```
SET SESSION calendar = Teradata;
/* Setting SUNDAY and SATURDAY as weekend days. */
exec dbc.createbusinesscalendarpattern('teradata','SUNDAY','OFF',);
exec dbc.createbusinesscalendarpattern('teradata','SATURDAY','OFF',);

SEL Item_Code, SUM(Sale_Amt) FROM Sales_Tbl,
       Sys_Calendar.BusinessCalendar
WHERE WeekNumber_Of_Quarter(Sale_Date) = 1
AND calendar_date = Sale_Date
AND IsBusinessDay  = 1
GROUP BY Item_Code
ORDER BY 1;
```

SET SESSION calendar = Teradata;

Item_Code	SUM(Sale_Amt)
101	190
102	120

Sales_Tbl		
Item_Code	Sale_Amt	Sale_Date
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Example 6: Show Week 1 of September 2011

This example shows week 1 of September, 2011, for each business calendar setting.

```
SEL calendar_date, week_of_month FROM Sys_Calendar.BusinessCalendar
WHERE week_of_month = 1
AND month_of_year = 9
AND year_of_calendar = 2011
ORDER BY 1;
SET SESSION calendar = ISO;
```

```
 calendar_date  week_of_month
 -------------  -------------
      11/08/29              1
      11/08/30              1
      11/08/31              1
      11/09/01              1
      11/09/02              1
      11/09/03              1
      11/09/04              1
```

Sep-2011 Calendar

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

SET SESSION calendar = Teradata;

```
calendar_date  week_of_month
-------------  -------------
     11/09/04              1
     11/09/05              1
     11/09/06              1
     11/09/07              1
     11/09/08              1
     11/09/09              1
     11/09/10              1
```

Sep-2011 Calendar

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

```
SET SESSION calendar = Compatible;
```

```
calendar_date  week_of_month
-------------  -------------
     11/09/01              1
     11/09/02              1
     11/09/03              1
     11/09/04              1
     11/09/05              1
     11/09/06              1
     11/09/07              1
```

Sep-2011 Calendar

S	M	T	W	T	F	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Example 7: Inventory Status at the Beginning of Each Week in a Month

The following example shows the inventory status at the beginning of each week for the month of September, 2011 for each business calendar setting.

```
SEL Item, BEGIN(week_duration) AS WeekBegin, Inventory
FROM Inventory_Tbl
EXPAND ON Duration AS week_duration
BY ANCHOR WEEK_BEGIN;
```

SET SESSION CALENDAR = **Teradata**;		
Item	WeekBegin	Inventory
101	**2011-09-04**	12
101	**2011-09-11**	18
101	**2011-09-18**	11

SET SESSION CALENDAR = **ISO**;		
Item	WeekBegin	Inventory
101	2011-09-05	18
101	2011-09-12	15
101	2011-09-19	11

SET SESSION CALENDAR = **Compatible**;		
Item	WeekBegin	Inventory
101	**2011-09-03**	12
101	**2011-09-10**	18
101	**2011-09-17**	15

Inventory_Tbl

Item	Duration	Inventory
101	2011-09-01, 2011-09-05	12
101	2011-09-05, 2011-09-12	18
101	2011-09-12, 2011-09-18	15
101	2011-09-18, 2011-09-21	11

Sep-2011 Calendar

S	M	T	W	T	F	S
				1	2	**3**
4	**5**	6	7	8	9	**10**
11	**12**	13	14	15	16	**17**
18	**19**	20	21	22	23	24
25	26	27	28	29	30	

Example 8: Number of Weeks in Each Quarter in 2004

The following example shows the number of weeks in each quarter for 2004.

```
SEL quarter_of_year, MAX(week_of_quarter)
FROM  Sys_Calendar.BusinessCalendar
WHERE year_of_calendar=2004
GROUP BY 1 ORDER BY 1;
```

SET SESSION CALENDAR = **Teradata**;	
Quarter_of_Year	Maximum(Week_of_Quarter)
1	13
2	13
3	13
4	13

SET SESSION CALENDAR = **COMPATIBLE**;	
Quarter_of_Year	Maximum(Week_of_Quarter)
1	13
2	13
3	14
4	14

SET SESSION CALENDAR = **ISO**;	
Quarter_of_Year	Maximum(Week_of_Quarter)
1	13
2	13
3	13

Example 9: Sales by Quarter

The following example shows how sales are calculated by quarter for each business calendar setting.

```
SEL QuarterNumber_Of_Year(Sale_Date) Quarter_No,
Item_Code, SUM(Sale_Amt) FROM Sales_Tbl
GROUP BY 1,2
ORDER BY 1,2;
```

```
SET SESSION calendar = ISO;
```

Quarter_No	Item_Code	SUM(Sale_Amt)
1	101	325
1	102	150

```
SET SESSION calendar = Teradata;
```

Quarter_No	Item_Code	SUM(Sale_Amt)
1	101	300
1	102	150
4	101	25

```
SET SESSION calendar = COMPATIBLE;
```

Quarter_No	Item_Code	SUM(Sale_Amt)
1	101	300
1	102	150
4	101	25

Sales_Tbl		
Item_Code	**Sale_Amt**	**Sale_Date**
101	10	2008-12-30
101	15	2008-12-31
101	20	2009-01-01
101	25	2009-01-02
101	30	2009-01-03
101	35	2009-01-04
101	40	2009-01-05
101	45	2009-01-06
101	50	2009-01-07
101	55	2009-01-08
102	10	2009-01-01
102	20	2009-01-03
102	30	2009-01-05
102	40	2009-01-07
102	50	2009-01-09

Jan 2009 Calendar						
S	M	T	W	T	F	S
	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

td_week_begin

Purpose

Returns the week that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬─ td_week_begin ─ ( expression_1 ┬──────────────────────────┬┬──────────────────┬─ ) ─
└─ TD_SYSFNLIB. ───┘                                └─ , ┬─ calendar_name ─┬┘└─ , ─ expression_2 ─┘
                                                         └─ NULL ──────────┘
```

1145-011A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_WEEK_BEGIN(TIMESTAMP'2012-01-15
12:00:00+02:00','ISO',TIME'05:00:00');
```

returns `2012-01-09 07:00:00+02:00`.

td_week_end

Purpose

Returns the week end that falls immediately after the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬─ td_week_end ── ( expression_1 ┬──────────────────────────┬┬──────────────────┬─ ) ──
└─ TD_SYSFNLIB. ──┘                                │                          │└─ , ─ expression_2 ─┘
                                                   └─ , ┬─ calendar_name ─┬─┘
                                                        └─ NULL ──────────┘
```

1145-012A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 23:59:59+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Example

The following SQL statement:

```
SELECT TD_WEEK_END(timestamp'2012-05-08 10:00:00');
```

returns `2012-05-12 23:59:59`.

where no calendar specified indicates the session calendar, which in this example is `TERADATA`.

td_sunday

Purpose

Returns the Sunday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_SUNDAY(timestamp'2012-05-08 10:00:00');
```

returns `2012-05-06 00:00:00`.

where no calendar specified indicates the session calendar, which in this case is `TERADATA`.

td_monday

Purpose

Returns the Monday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬── td_monday ── ( expression_1 ┬──────────────────────────┬┬──────────────────────┬── ) ──
└─ TD_SYSFNLIB. ──┘                                └─ , ┬─ calendar_name ─┬─┘└─ , ─ expression_2 ─┘
                                                        └─ NULL ──────────┘
```

1145-020A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
TD_MONDAY(Date,'TERADATA');
```

returns `12/05/07`.

td_tuesday

Purpose

Returns the Tuesday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── td_tuesday ── ( expression_1 ┌──────────────────────────────┐┌──────────────────────┐ ) ──
                                                       │                              │└─ , ─ expression_2 ─┘
                                                       └─ , ─ calendar_name ──┐
                                                              └─ NULL ────────┘
                                                                                                  1145-021A
```

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_TUESDAY(TIMESTAMP'2012-05-08 10:00:00');
```

returns `2012-05-08 00:00:00`.

where no calendar specified indicates the session calendar, which in this case is `TERADATA`.

td_wednesday

Purpose

Returns the Wednesday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┌──────────────────┐
└─ TD_SYSFNLIB. ───┘ td_wednesday ── ( expression_1 ┌──────────────────────────────┐┌──────────────────────┐ ) ──
                                                    └─ , ┌─ calendar_name ─┐┘└─ , ─ expression_2 ─┘
                                                         └─ NULL ──────────┘
                                                                                                  1145-022A
```

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_WEDNESDAY(TIMESTAMP'2012-05-08 10:00:00+01:00','ISO');
```

returns `2012-05-02 01:00:00+01:00`.

td_thursday

Purpose

Returns the Thursday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── td_thursday ── ( expression_1 ──┬──────────────────────────┬──┬──────────────────────┬── ) ──
                                          │               │                          │  │                      │
                                          └─ , ─┬─ calendar_name ─┬─┘  └─ , ─ expression_2 ─┘
                                                └─ NULL ──────────┘
```

1145-023A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_THURSDAY(TIMESTAMP'2012-05-08 10:00:00',NULL,NULL);
```

returns `2012-05-03 00:00:00`.

where NULL indicates the session calendar, which in this case is `TERADATA`.

td_friday

Purpose

Returns the Friday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┌──────────────────────┐
└─ TD_SYSFNLIB. ───────┘── td_friday ── ( expression_1 ──┬──────────────────────┬──┬──────────────────────┬── ) ──
                                                        └─ , ─┬─ calendar_name ─┬┘  └─ , ─ expression_2 ─┘
                                                              └─ NULL ──────────┘
                                                                                                  1145-024A
```

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_FRIDAY(TIMESTAMP'2012-05-08 10:00:00',NULL,TIME'05:00:00');
```

returns `2012-05-04 05:00:00`.

where NULL indicates the session calendar, which in this case is `TERADATA`.

td_saturday

Purpose

Returns the Saturday that falls immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬─ td_saturday ── ( expression_1 ┬──────────────────────────┬──────────────────────┬─ ) ─
└─ TD_SYSFNLIB. ───┘                                │                          │                      │
                                                    └─ , ┬─ calendar_name ─┬─┘   └─ , ─ expression_2 ─┘
                                                         └─ NULL ──────────┘
                                                                                                  1145-025A
```

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_SATURDAY(TIMESTAMP'2012-05-08
10:00:00',NULL,TIME'05:00:00+03:00');
```

returns `2012-05-05 02:00:00`.

where NULL indicates the session calendar, which in this case is `TERADATA`.

DayNumber_Of_Week

Purpose

Returns the number of days from the beginning of the week to the specified date.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘─ DAYNUMBER_OF_WEEK ── ( expression ─┬─────┬─┬──────────────────┬─ ) ──
                                                            └─ , ─┘ │ calendar_name ───┬│
                                                                    └──────── NULL ────┘
                                                                                  1101A929
```

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value between 1 and 7, representing the day of the week, where the first day of the week = 1 and the last day of the week = 7. The first day of the week is defined by the business calendar the session is using. For example, in the ISO calendar, Monday = 1 and in the Teradata business calendar Monday = 2.

Usage Notes

The DayNumber_Of_Week function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

The following query returns the names of employees who joined the company on the first day of the week:

```
SELECT empname from emp
WHERE DAYNUMBER_OF_WEEK (date_of_join)= 1;
```

td_month_begin

Purpose

Returns the month that begins immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬ td_month_begin ─ ( expression_1 ┬──────────────────────────┬┬──────────────────────┬ ) ─
└ TD_SYSFNLIB. ────┘                                │ , ┬ calendar_name ┬ │└ , ─ expression_2 ────┘
                                                    │   └ NULL ─────────┘ │
                                                    └──────────────────────┘
```

1145-013A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_MONTH_BEGIN(DATE'2012-01-15');
```

returns `12/01/01`.

where no calendar specified indicates the session calendar, which in this case is `Teradata`.

td_month_end

Purpose

Returns the month that ends immediately after the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬─ td_month_end ─ ( expression_1 ┬──────────────────────┬┬──────────────────┬ ) ─
└─ TD_SYSFNLIB. ───┘                                └─ , ┬─ calendar_name ─┬┘└─ , ─ expression_2 ─┘
                                                         └─ NULL ──────────┘
```

1145-014A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_MONTH_END(DATE'2012-01-15',NULL);
```

returns `12/01/31`.

where NULL indicates the session calendar, which in this case is `Teradata`.

DayNumber_Of_Month

Purpose

Returns the number of days from the beginning of the month to the specified date.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value from 1 through 31.

Usage Notes

The DayNumber_Of_Month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Examples

In both of the following examples, no business calendar is set for the session, so the query uses the system-defined business calendar Teradata.

The following query gives the day number of the month as `31` instead of `1` because the input is converted to UTC 2009-12-31 19:00:00.

```
SET TIME ZONE INTERVAL ‘05:00’ HOUR TO MINUTE;
SELECT DAYNUMBER_OF_MONTH (TIMESTAMP‘2010-01-01 00:00:00’);
```

The following query gives the day number of the month as `31` instead of `1` because the input is converted to UTC 2009-12-31 21:00:00.

```
SEL DAYNUMBER_OF_MONTH (TIMESTAMP’2010-01-01 00:00:00+03:00’);
```

DayOccurrence_Of_Month

Purpose

Returns the *n*th occurrence of the weekday in the month for the specified date.

Syntax

```
┌──────────────────┐
└─TD_SYSFNLIB.─────┘─DAYOCCURRENCE_OF_MONTH─( expression ┌──────────────────────┐ )─
                                                        └─,─┬─calendar_name─┬─┘
                                                            └──────NULL─────┘
```

1101A936

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value between 1 and 5, representing the *n*th occurrence of the weekday in the month.

Usage Notes

The DayOccurrence_of_Month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is May 1, 2010, the following queries return `1` because May 1, 2010, falls on the first Saturday of the month:

```
SELECT TD_SYSFNLIB.DAYOCCURRENCE_OF_MONTH(CURRENT_DATE);

SELECT TD_SYSFNLIB.DAYOCCURRENCE_OF_MONTH(DATE '2010-05-01');
```

WeekNumber_Of_Month

Purpose

Returns the number of weeks from the beginning of the month to the specified date.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value between 1 and 5, representing the *n*th occurrence of the week in the month. The value 0 means the partial week, Week0.

Usage Notes

The WeekNumber_Of_Month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is January 5, 1901, the following query returns `0` because January 5, 1901, is in week 0 of January in the Teradata calendar:

```
SELECT TD_SYSFNLIB.WEEKNUMBER_OF_MONTH (CURRENT_DATE);
```

The following query returns `1` because January 5, 1901, is in week 1 of January in the ISO calendar:

```
SELECT TD_SYSFNLIB.WEEKNUMBER_OF_MONTH (DATE '1901-01-05', 'ISO');
```

td_year_begin

Purpose

Returns the year that begins immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬─ td_year_begin ── ( expression_1 ┬──────────────────────────┬┬──────────────────┬ ) ──
└─ TD_SYSFNLIB. ───┘                                 └─ , ┬─ calendar_name ─┬─┘└─ , ─ expression_2 ─┘
                                                          └─ NULL ──────────┘
```

1145-017A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_YEAR_BEGIN(TIMESTAMP'2012-01-15 12:00:00','ISO',NULL);
```

returns `2012-01-02 00:00:00`.

td_year_end

Purpose

Returns the year that ends immediately after the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── td_year_end ── ( expression_1 ┌──────────────────────────┐┌──────────────────────┐ ) ──
                                                        │                          │└─ , ─ expression_2 ─┘
                                                        └─ , ─┬─ calendar_name ─┬─┘
                                                              └─ NULL ──────────┘
```

1145-018A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_YEAR_END(TIMESTAMP'2012-01-15 10:00:00','ISO');
```

returns `2012-12-30 23:59:59`.

DayNumber_Of_Year

Purpose

Returns the number of days from the beginning of the year to the specified date.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── DAYNUMBER_OF_YEAR ── ( expression ┌─ , ─┐┌──────────────────────┐ ) ──
                                                            │     ├─ calendar_name ──────┤
                                                            │     └─ NULL ───────────────┘
                                                            └─────────────────────────────┘
```

1101A931

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value from 1 through 366.

Usage Notes

The DayNumber_Of_Year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is February 10, 2010, the following queries return `41` because February 10, 2010, is the 41st day since the beginning of the Teradata calendar year:

```
SELECT TD_SYSFNLIB.DAYNUMBER_OF_YEAR(CURRENT_DATE);

SELECT TD_SYSFNLIB.DAYNUMBER_OF_YEAR(DATE '2010-02-10');
```

WeekNumber_Of_Year

Purpose

Returns the number of weeks from the beginning of the year to the specified date.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value from 0 through 53, where 1 is the first week of the year and 53 is the last week of the year. The value 0 means that there is a partial week in the year, Week0.

Usage Notes

The WeekNumber_Of_Year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is January 5, 1901, the following query returns `0` because January 5, 1901, is in week 0 of 1901 in the Teradata calendar:

```
SELECT TD_SYSFNLIB.WEEKNUMBER_OF_YEAR (CURRENT_DATE);
```

The following query returns `1` because January 5, 1901, is in week 1 of 1901 in the COMPATIBLE calendar:

```
SELECT TD_SYSFNLIB.WEEKNUMBER_OF_YEAR (DATE '1901-01-05', 'COMPATIBLE');
```

MonthNumber_Of_Year

Purpose

Returns the number of months from the beginning of the year to the specified date.

Syntax

```
┌──────────────────┐
└─TD_SYSFNLIB.─────┘─MONTHNUMBER_OF_YEAR──( expression ──┌─,─┐──┌──────────────────┐──)──
                                                         └─┘  ├─ calendar_name ─┤
                                                              └──── NULL ───────┘
```

1101A937

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value between 1 and 12, representing the *n*th month of the year.

Usage Notes

The MonthNumber_Of_Year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is March 15, 2011, the following query returns `3` because March 15, 2011, is in the third month of the Teradata calendar:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_YEAR(CURRENT_DATE);
```

The following query returns `3` because March 15, 2011, is in the third month of the ISO calendar:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_YEAR(DATE '2011-03-15', 'ISO');
```

td_quarter_begin

Purpose

Returns the quarter that begins immediately before the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬ td_quarter_begin ─ ( expression_1 ┬──────────────────────────┬┬──────────────────┬ ) ─
└ TD_SYSFNLIB. ────┘                                  └ , ┬ calendar_name ┬┘└ , ─ expression_2 ────┘
                                                          └ NULL ─────────┘
```

1145-015A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_QUARTER_BEGIN(DATE'2012-01-15','COMPATIBLE');
```

returns `12/01/01`.

td_quarter_end

Purpose

Returns the quarter that ends immediately after the DATE or TIMESTAMP specified in *expression_1*.

Syntax

```
┬──────────────────┬─ td_quarter_end ─ ( expression_1 ┬──────────────────────────┬──────────────────────┬ ) ─
└─ TD_SYSFNLIB. ───┘                                 └─ , ┬─ calendar_name ─┬─┘└─ , ─ expression_2 ─┘
                                                          └─ NULL ──────────┘
```

1145-016A

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression_1

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

If *expression_1* is DATE, *expression_2* cannot be specified.

expression_2

A TIME, TIME WITH TIME ZONE, or NULL value. If NULL is specified, TIME is assumed to be 00:00:00+00:00.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The function is overloaded and can be invoked with one, two, or three arguments.

The first argument can be one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument can be one of the following system-defined calendars (TERADATA, ISO, or COMPATIBLE) or NULL

The third argument can be one of the following data types:

- TIME
- TIME ITH TIME ZONE
- NULL

Result

The return value is one of the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

Example

The following SQL statement:

```
SELECT TD_QUARTER_END(TIMESTAMP'2012-01-15 10:00:00','COMPATIBLE');
```

returns `2012-03-31 23:59:59`.

WeekNumber_Of_Quarter

Purpose

Returns the number of weeks from the beginning of the quarter to the specified date.

Syntax

```
┌──────────────────┐
│                  │
└─TD_SYSFNLIB.─────┘─WEEKNUMBER_OF_QUARTER─( expression ─┌─,─┐─┌──────────────────────┐─)──
                                                         │   │ │                      │
                                                         └───┘ │ calendar_name        │
                                                               │ NULL                 │
                                                               └──────────────────────┘
```

1101A943

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value between 1 and 14, representing the *n*th week of the quarter. The value 0 means the partial week, Week0.

Example

If the current date is April 20, 2011, the following query returns `3` because April 20, 2011, is in the third week of the quarter in the Teradata calendar:

```
SELECT TD_SYSFNLIB.WEEKNUMBER_OF_QUARTER(CURRENT_DATE);
```

The following query returns `3` because April 20, 2011, is in the third week of the quarter in the ISO calendar:

```
SELECT TD_SYSFNLIB.WEEKNUMBER_OF_QUARTER (DATE '2011-04-20', 'ISO');
```

MonthNumber_Of_Quarter

Purpose

Returns the number of months from the beginning of the quarter to the specified date.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value between 1 to 3, representing the *n*th month of the quarter.

Usage Notes

The MonthNumber_Of_Quarter function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is April 20, 2011, the following query returns `1` because April 20, 2011, is in the first month of the quarter in the Teradata calendar:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_QUARTER(CURRENT_DATE);
```

The following query returns `1` because April 20, 2011, is in the first month of the quarter in the ISO calendar:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_QUARTER(DATE '2011-04-20', 'ISO');
```

QuarterNumber_Of_Year

Purpose

Returns the number of quarters from the beginning of the year to the specified date.

Syntax

```
┌──────────────────┐
└TD_SYSFNLIB.──────┘ QUARTERNUMBER_OF_YEAR ─( expression ┌──┐┌──────────────────┐ )──
                                                        └, ┘└─ calendar_name ─┘
                                                            └──── NULL ───────┘
```

1101A940

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value between 1 and 4, representing the *n*th quarter of the year.

Usage Notes

The QuarterNumber_Of_Year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is December 20, 2010, the following query returns `4` because December 20, 2010, is in the fourth quarter in the Teradata calendar:

```
SELECT TD_SYSFNLIB.QUARTERNUMBER_OF_YEAR(CURRENT_DATE);
```

The following query returns `4` because December 10, 2010, is in the fourth quarter in the ISO calendar:

```
SELECT TD_SYSFNLIB.QUARTERNUMBER_OF_YEAR(DATE '2010-12-20', 'ISO');
```

DayNumber_Of_Calendar

Purpose

Returns the number of days from the beginning of the business calendar to the specified date.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value representing the number of days since and including the beginning of the business calendar.

Usage Notes

The DayNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is January 05, 1901, the following queries return `370` since January 05, 1901, is the 370th day since January 1, 1900:

```
SELECT TD_SYSFNLIB.DAYNUMBER_OF_CALENDAR(CURRENT_DATE);

SELECT TD_SYSFNLIB.DAYNUMBER_OF_CALENDAR(DATE '1901-01-05');
```

WeekNumber_Of_Calendar

Purpose

Returns the number of weeks from the beginning of the business calendar to the specified date.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value. The value 0 means there is a partial week in the year, which is Week0.

Usage Notes

The WeekNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the view, see *Data Dictionary*.

Example

If the current date is January 5, 1901, the following query returns `52` because January 5, 1901, is in week 52 since the start of the Teradata calendar on 1900-01-01:

```
SELECT TD_SYSFNLIB.WEEKNUMBER_OF_CALENDAR (CURRENT_DATE);
```

The following query returns `53` because January 5, 1901, is in week 53 since the start of the COMPATIBLE calendar on 1900-01-01:

```
SELECT TD_SYSFNLIB.WEEKNUMBER_OF_CALENDAR (DATE '1901-01-05',
'COMPATIBLE');
```

MonthNumber_Of_Calendar

Purpose

Returns the number of months from the beginning of the calendar to the specified date.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value representing the number of months since and including the beginning of the calendar.

Usage Notes

The MonthNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is May 5, 1901, the following query returns `17` because May 5, 1901, is in the 17th month since the start of the Teradata calendar on 1900-01-01:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_CALENDAR(CURRENT_DATE);
```

The following query returns `17` because May 5, 1901, is in the 17th month since the start of the COMPATIBLE calendar on 1900-01-01:

```
SELECT TD_SYSFNLIB.MONTHNUMBER_OF_CALENDAR(DATE '1901-05-05',
'COMPATIBLE');
```

QuarterNumber_Of_Calendar

Purpose

Returns the number of quarters from the beginning of the calendar to the specified date.

Syntax

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value, representing the number of quarters since and including the beginning of the calendar.

Usage Notes

The QuarterNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is May 5, 1901, the following query returns `6` because May 5, 1901, is in the 6th quarter since 1900-01-01, the beginning of the Teradata calendar:

```
SELECT TD_SYSFNLIB.QUARTERNUMBER_OF_CALENDAR(CURRENT_DATE);
```

The following query returns `6` because May 5, 1901, is in the 6th quarter since 1900-01-01, the beginning of the COMPATIBLE calendar:

```
SELECT TD_SYSFNLIB.QUARTERNUMBER_OF_CALENDAR (DATE '1901-05-05',
'COMPATIBLE');
```

YearNumber_Of_Calendar

Purpose

Returns the year of the specified date.

Syntax

```
┌──────────────────┐
└─TD_SYSFNLIB. ────┘─ YEARNUMBER_OF_CALENDAR ── ( expression ─┬─ , ─┬─┬──────────────────┬─┬─ ) ──
                                                              │     │ calendar_name    │ │
                                                              │     └──────── NULL ────┘ │
                                                              └─────────────────────────┘
```

1101A942

Syntax Elements

calendar_name

A calendar name. The possible values are Teradata, ISO, and COMPATIBLE.

This argument must be a character literal and cannot be a table column or expression. If you do not name a calendar, Teradata uses the calendar for the session.

expression

An expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

NULL

The business calendar for the session.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Argument Types

The first argument, *expression*, is defined with the following data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

The second argument, *calendar_name*, is optional and is defined with the following argument data types:

- VARCHAR
- NULL

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

Result

The result is an INTEGER value representing the year in 4-digit format.

Usage Notes

The YearNumber_Of_Calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is June 14, 1906, the following queries return `1906`:

```
SELECT TD_SYSFNLIB.YEARNUMBER_OF_CALENDAR(CURRENT_DATE);
```

```
SELECT TD_SYSFNLIB.YEARNUMBER_OF_CALENDAR (DATE '1906-06-14',
'COMPATIBLE');
```

CHAPTER 9 Calendar Functions

This chapter describes the functions that provide support for DateTime operations that use calendar attributes.

Prerequisites

Before you can use these functions, you must run the Database Initialization Program (DIP) utility and run the DIPUDT script. The DIPALL and DIPUDT scripts create the calendar functions in the SYSLIB database. For more information about the DIP utility, see *Utilities*.

If you have a user-developed UDF with the same name as a calendar function, you must remove that user-developed UDF from the normal UDF search path before you can invoke the calendar function. If the calendar function is not found in the current database, Teradata Database searches for the function in the SYSLIB database. Alternatively, you may invoke the calendar function by using the fully qualified syntax, SYSLIB.*calendar_function_name*.

Calendar Function Compatibility with Previous Releases

Beginning with Release 14.0, Teradata calendar functions have different names. For example, the Release 13.10 Day_Of_Week function is replaced by TD_Day_Of_Week for Release 14.0 and up.

You may need to change the way you use calendar functions to assure compatibility with the current Teradata Database release.

td_day_of_week

Purpose

Returns the day of the week on which the specified date falls.

Syntax

```
┬──────────┬── td_day_of_week ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B725

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

Argument Types

td_day_of_week is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 1 through 7, representing the day of the week, where Sunday = 1 and Saturday = 7.

Usage Notes

The td_day_of_week function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is October 18, 2010, which is a Monday, the following queries return the value 2 as the result since Monday is the 2nd day of the week.

```
SELECT td_day_of_week(CURRENT_DATE);

SELECT td_day_of_week(DATE '2010-10-18');
```

td_day_of_month

Purpose

Returns the number of days from the beginning of the month through the specified date.

Syntax

```
┬──────────┬── td_day_of_month ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B726

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_day_of_month is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 1 through 31.

Usage Notes

The td_day_of_month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is May 27, 2010, the following queries return the value 27 as the result since May 27, 2010 is the 27th day from the beginning of the month of May.

```
SELECT td_day_of_month(CURRENT_DATE);

SELECT td_day_of_month(DATE '2010-05-27');
```

td_day_of_year

Purpose

Returns the number of days from the beginning of the year (January 1st) to the specified date.

Syntax

```
┬──────────┬─ td_day_of_year ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B727

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_day_of_year is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 1 through 366.

Usage Notes

The td_day_of_year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is February 10, 2010, the following queries return the value 41 as the result since February 10, 2010 is the 41st day from the beginning of the year.

```
SELECT td_day_of_year(CURRENT_DATE);

SELECT td_day_of_year(DATE '2010-02-10');
```

td_day_of_calendar

Purpose

Returns the number of days from the beginning of the calendar starting on 01/01/1900 to the specified date.

Syntax

```
┬──────────┬─ td_day_of_calendar ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B728

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_day_of_calendar is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value representing the number of days since and including 01/01/1900.

Usage Notes

The td_day_of_calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is January 05, 1901, the following queries return the value 370 as the result since January 05, 1901 is the 370th day since January 01, 1900.

```
SELECT td_day_of_calendar(CURRENT_DATE);

SELECT td_day_of_calendar(DATE '1901-01-05');
```

td_weekday_of_month

Purpose

Returns the *n*th occurrence of the weekday in the month for the specified date.

Syntax

```
┬──────────────┬── td_weekday_of_month ── (expression) ──────────────────────
└─ SYSLIB. ────┘
```

1101B729

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_weekday_of_month is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 1 to 5, representing the *n*th occurrence of the weekday in the month.

Usage Notes

The td_weekday_of_month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is May 01, 2010, the following queries return the value 1 as the result since May 01, 2010 falls on the first Saturday of the month.

```
SELECT td_weekday_of_month(CURRENT_DATE);

SELECT td_weekday_of_month(DATE '2010-05-01');
```

td_week_of_month

Purpose

Returns the *n*th full week from the beginning of the month to the specified date.

Syntax

```
┬──────────┬── td_week_of_month ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B730

where:

Syntax element…	Specifies…
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_week_of_month is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 0 to 5, representing the *n*th full week from the beginning of the month, where the first partial week is 0.

Usage Notes

The td_week_of_month function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is May 01, 2010, the following queries return the value 0 as the result since May 01, 2010 falls on the first partial week of May.

```
SELECT td_week_of_month(CURRENT_DATE);

SELECT td_week_of_month(DATE '2010-05-01');
```

td_week_of_year

Purpose

Returns the *n*th full week from the beginning of the year (January 1st) to the specified date.

Syntax

```
┬──────────┬── td_week_of_year ── (expression) ──────────────────────────
└ SYSLIB. ┘
```

1101B731

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_week_of_year is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 0 to 53, representing the *n*th full week from the beginning of the year, where the first partial week is 0.

Usage Notes

The td_week_of_year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is May 04, 2010, the following queries return the value 18 as the result since May 04, 2010 falls on the 18th week of the year.

```
SELECT td_week_of_year(CURRENT_DATE);

SELECT td_week_of_year(DATE '2010-05-04');
```

td_week_of_calendar

Purpose

Returns the number of weeks from the beginning of the calendar starting on 01/01/1900 to the specified date.

Syntax

```
┬──────────┬── td_week_of_calendar ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B732

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_week_of_calendar is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value representing the number of full weeks since and including the week of 01/01/1900, where the first partial week is 0.

Usage Notes

The td_week_of_calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is January 10, 1901, the following queries return the value 53 as the result since January 10, 1901 falls on the 53rd week since January 01, 1900.

```
SELECT td_week_of_calendar(CURRENT_DATE);

SELECT td_week_of_calendar(DATE '1901-01-10');
```

td_month_of_quarter

Purpose

Returns the number of months from the beginning of the quarter to the specified date.

Syntax

```
┬──────────────┬── td_month_of_quarter ── (expression) ──────────────────────────
└─ SYSLIB. ────┘
```

1101B733

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_month_of_quarter is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 1 to 3.

Usage Notes

The td_month_of_quarter function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is June 12, 2010, the following queries return the value 3 as the result because June 12, 2010 falls on the 3rd month of the 2nd quarter.

```
SELECT td_month_of_quarter(CURRENT_DATE);

SELECT td_month_of_quarter(DATE '2010-06-12');
```

td_month_of_year

Purpose

Returns the number of months from the beginning of the year (January 1st) to the specified date.

Syntax

```
┬──────────────┬─ td_month_of_year ── (expression) ──────────────────────────
└─ SYSLIB. ────┘
                                                                  1101B734
```

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_month_of_year is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 1 to 12.

Usage Notes

The td_month_of_year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is August 29, 2010, the following queries return the value 8 as the result because August 29, 2010 falls on the 8th month of the year.

```
SELECT td_month_of_year(CURRENT_DATE);

SELECT td_month_of_year(DATE '2010-08-29');
```

td_month_of_calendar

Purpose

Returns the number of months from the beginning of the calendar starting on 01/01/1900 to the specified date.

Syntax

```
┬──────────┬── td_month_of_calendar ── (expression) ──────────────────────
└─ SYSLIB. ┘
```

1101B735

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_month_of_calendar is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value representing the number of months since and including January, 1900.

Usage Notes

The td_month_of_calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is August 29, 1901, the following queries return the value 20 as the result since August 29, 1901 falls on the 20th month since January 01, 1900.

```
SELECT td_month_of_calendar(CURRENT_DATE);

SELECT td_month_of_calendar(DATE '1901-08-29');
```

td_quarter_of_year

Purpose

Returns the quarter number of the year for the specified date.

Syntax

```
┬──────────┬── td_quarter_of_year ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B736

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_quarter_of_year is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value from 1 to 4, representing the quarter number from the beginning of the year, where 1 = first quarter (Jan/Feb/Mar) and 4 = fourth quarter (Oct/Nov/Dec).

Usage Notes

The td_quarter_of_year function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is November 14, 1983, the following queries return the value 4 as the result since November 14, 1983 falls on the 4th quarter of the year.

```
SELECT td_quarter_of_year(CURRENT_DATE);

SELECT td_quarter_of_year(DATE '1983-11-14');
```

td_quarter_of_calendar

Purpose

Returns the number of quarters from the beginning of the calendar starting on 01/01/1900 to the specified date.

Syntax

```
┬──────────┬─ td_quarter_of_calendar ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B737

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_quarter_of_calendar is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value representing the number of quarters since and including the first quarter of 1900.

Usage Notes

The td_quarter_of_calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is November 14, 1901, the following queries return the value 8 as the result since November 14, 1901 falls on the 8th quarter since January 01, 1900.

```
SELECT td_quarter_of_calendar(CURRENT_DATE);

SELECT td_quarter_of_calendar(DATE '1901-11-14');
```

td_year_of_calendar

Purpose

Returns the year of the specified date.

Syntax

```
┬──────────┬── td_year_of_calendar ── (expression) ──────────────────────────
└─ SYSLIB. ┘
```

1101B738

where:

Syntax element...	Specifies...
SYSLIB	the name of the database where the function is located.
expression	an expression that results in a DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value.

ANSI Compliance

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

Argument Types

td_year_of_calendar is an overloaded scalar function. It is defined with the following parameter data types:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE

If the argument passed to the function does not match one of these declared data types, an error is returned indicating that the function does not exist.

For more information on overloaded functions, see “Function Name Overloading” in *SQL External Routine Programming*.

Result

The result is an INTEGER value in 4 digit format representing the year of the specified date.

Usage Notes

The td_year_of_calendar function provides improved performance compared to using the Sys_Calendar.Calendar system view to obtain similar results.

For more information about the CALENDAR system view, see *Data Dictionary*.

Example

If the current date is November 14, 1977, the following queries return the value 1977 as the result, which is the year of the specified date.

```
SELECT td_year_of_calendar(CURRENT_DATE);

SELECT td_year_of_calendar(DATE '1977-11-14');
```

CHAPTER 10 Comparison Operators and Functions

This chapter describes SQL comparison operators.

Comparison Operators

Purpose

Comparison operators, which are types of logical predicates (also called conditional expressions), test the truth of relations between expressions.

Comparison operators can appear in:

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

Syntax

where:

Syntax Element ...	Specifies ...
operator	one of the comparison operators.
expression_1 *expression_2*	an SQL scalar expression, including derived period expressions.

Syntax Element ...	Specifies ...
quantifier	one of the following quantifier keywords: • ANY • SOME • ALL For information, see "ANY / ALL / SOME Quantifiers" on page 907.
literal	one or more literal values. A literal may be any of the following: • Defined value • Macro parameter • Built-in value such as TIME, DATE, or USER The comparison operation may compare an expression against a list of explicit literals. The data types of *expression* and *literal* must be compatible. If the data types of the operands differ, Teradata Database performs an implicit conversion from one type to another in some cases. For details, see "Implicit Type Conversion of Comparison Operands" on page 502.
subquery	an SQL SELECT statement. Using a subquery in a condition is restricted in certain cases.
scalar_expression	an expression to be evaluated in comparison with a second *scalar_expression*, including derived periods. Comparison operators do not support BLOB or CLOB type expressions. You can explicitly cast BLOBs to BYTE or VARBYTE and cast CLOBs to CHARACTER or VARCHAR, and use the result with comparison operators. An expression that results in a UDT data type can only be compared with another expression that results in the same UDT data type.
comparison_operator	the type of comparison to be evaluated for truth. For a list of the supported comparison operators, see "Supported Comparison Operators" on page 497.

Results

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

ANSI Compliance

The following comparison operators are ANSI SQL:2011 compliant.

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

The following comparison operators are Teradata extensions to the ANSI SQL:2011 standard. Their use is deprecated.

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

Supported Comparison Operators

Teradata Database supports the following comparison operators.

ANSI Operator	Teradata Extensions	Function
=	EQ	Tests for equality.
<>	^= NE NOT=	Tests for inequality.
<	LT	Tests for less than.
<=	LE	Tests for less than or equal.
>	GT	Tests for greater than.
>=	GE	Tests for greater than or equal.

Comparison Operators Using Subqueries

A subquery is a SELECT statement that returns values used to satisfy the comparison operation. The subquery must be enclosed in parentheses, and it does not end with a semicolon.

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

A comparison operation may be used with a subquery whether or not a quantifier is used. If a quantifier is not used, however, then an error condition results if the subquery returns more than one value.

If a subquery returns no values, and if a quantifier is not used, then the result of the comparison is false. Therefore, if the following form is used, the subquery must return either no values (in which case the comparison evaluates to false), or it returns one value.

```
expression > (subquery)
```

With the following form, subquery must select the same number of expressions as are specified in the expression list.

The two expression lists are equal if each of the respective expressions are equal.

If the respective expressions are not equal, then the result of the comparison is determined by comparing the first pair of expressions (from the left) for which the comparison is not true.

A subquery in a comparison operation cannot specify a SELECT AND CONSUME statement.

Example

The following statement uses the ALL quantifier to compare two expressions with the values returned from a subquery to find the employee(s) with the most years of experience in the group of employees having the highest salary:

```
SELECT EmpNo, Name, DeptNo, JobTitle, Salary, YrsExp
FROM Employee
WHERE (Salary,YrsExp) >= ALL
 (SELECT Salary,YrsExp FROM Employee) ;
```

Comparisons That Produce TRUE Results

Conditions

The following table provides the conditions when comparisons produce TRUE results.

For simplicity, assume the syntax:

expression_1 — *operator* — *expression_2*

expression_1 and *expression_2* must contain the same number of scalar values and range from 1 through *n* rows, represented by *r*, so that the 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. If the data types of the expressions differ, Teradata Database performs an implicit conversion from one type to another in some cases. For details, see "Implicit Type Conversion of Comparison Operands" on page 502.

For an explanation of the symbols used in this table, see "Predicate Calculus Notation Used In This Book" on page 1336.

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.

This comparison ...	Is TRUE iff ...
expression_1 <= *expression_2*	*expression_1* < *expression_2* is TRUE or *expression_1* = *expression_2* is TRUE.
expression_1 => *expression_2*	*expression_1* > *expression_2* is TRUE or *expression_1* = *expression_2* is TRUE.

Null Expressions

If any expression in a comparison is null, the result of the comparison is unknown.

For a comparison to provide a TRUE result when comparing fields that might result in nulls, the statement must include the IS [NOT] NULL operator.

Floating Point Expressions

Calculations involving floating point values often produce results that are not what you expect. If you perform a floating point calculation and then compare the results against some expected value, it is unlikely that you get the intended result.

Instead of comparing the results of a floating point calculation, make sure that the result is greater or less than what is needed, with a given error. Here is an example:

```
SELECT i, SUM(a) as sum_a, SUM(b) as sum_b
FROM t1
GROUP BY i
HAVING ABS(sum_a - sum_b) > 1E-10;
```

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

More Information on Comparison Operators

FOR more information on ...	SEE ...
using comparison operators in conditional expressions in searched CASE expressions	Chapter 12: "CASE Expressions."
using comparison operators in conditional expressions in WHERE, ON, or HAVING clauses in SELECT statements	"The SELECT Statement" in *SQL Data Manipulation Language*.
using comparison operators in conditional expressions in IF, WHILE, or REPEAT statements in stored procedures	*SQL Stored Procedures and Embedded SQL*.

FOR more information on ...	SEE ...
other comparison operator, including: • [NOT] EXISTS • [NOT] IN • LIKE • IS [NOT] NULL • [NOT] BETWEEN … AND …	Chapter 19: "Logical Predicates."
predicate quantifiers: • ALL • ANY • SOME	

Data Type Evaluation

Different data types define equality and inequality differently. The following table explains the foundations for how the various data types are compared:

This data type ...	Is evaluated in this way ...
Numeric	Algebraically, with negatives considered to be smaller irrespective of their absolute value.

This data type ...	Is evaluated in this way ...
Byte	Bit-by-bit from left to right. A 0 bit is less than a 1 bit. • If every pair comparison is equal, then the two byte string are equal. • If any pairwise comparison is not equal, then that comparison determines the result. • If two byte strings of different lengths are compared, then the shorter string is padded to the right with binary zeros to make the lengths equal prior to making the comparison.
Character	Character-by-character from left to right. Exact comparisons depend on the collation sequence assigned and whether the comparison is case specific or case blind. The available collations are: • ASCII • EBCDIC • MULTINATIONAL • CHARSET_COLL • JIS_COLL If every pairwise comparison is equal, then the two character strings are equal. If any pairwise comparison is not equal, then that comparison determines the result. For more information on character comparison, see "Character String Comparisons" on page 507.
DateTime	Chronologically. For information on how Time Zone affects Time comparison, see "Time Zone Sort Order" on page 829.
Interval	According to sign and magnitude.

Period	Assuming p1 and p2 are Period expressions or derived periods, the evaluation of a Period comparison predicate uses the following logic:
	`IF BEGIN(p1) = BEGIN(p2) is TRUE, return END(p1) operator END(p2)`
	`ELSE return (BEGIN(p1) operator BEGIN(p2))`
	For details on BEGIN and END, see Chapter 22: "Period Functions and Operators."

UDT	According to the ordering definition of the UDT. Teradata Database generates ordering functionality for distinct UDTs where the source types are not LOBs. To create an ordering definition for structured UDTs or distinct UDTs where the source types are LOBs, or to replace system-generated ordering functionality, use CREATE ORDERING. For more information on CREATE ORDERING, see *SQL Data Definition Language*.

Implicit Type Conversion of Comparison Operands

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

Data Types on Which Implicit Conversion is Performed

If operand data types differ, Teradata Database performs an implicit conversion according to the following table. Implicit conversions are Teradata extensions to the ANSI SQL:2011 standard.

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

IF one expression operand is ...	AND the other expression operand is ...	THEN Teradata Database compares the data as ...
Character	Character	Character. For more details, see "Character String Comparisons" on page 507.
	Date	Date. Teradata Database returns an error for character data with GRAPHIC server character set.
	BYTEINT SMALLINT INTEGER FLOAT	FLOAT. Teradata Database returns an error for character data with GRAPHIC server character set.
	Period	Period.
CHAR(*k*) VARCHAR(*k*)) WHERE *k* <= 16	BIGINT DECIMAL(*m*,*n*)	FLOAT. Teradata Database returns an error for character data with GRAPHIC server character set.
CHAR(*k*) VARCHAR(*k*) where *k* > 16	DECIMAL(*m*,*n*) where *m* <= 16	FLOAT. Teradata Database returns an error for character data with GRAPHIC server character set.
	BIGINT DECIMAL(*m*,*n*) where *m* > 16	Teradata Database returns an error.

IF one expression operand is ...	AND the other expression operand is ...	THEN Teradata Database compares the data as ...
CHAR(*k*) VARCHAR(*k*)	NUMBER and *k* <=16	FLOAT. Teradata Database returns an error for character data with GRAPHIC server character set.
	NUMBER and *k* > 16	Teradata Database returns an error.
BYTEINT	SMALLINT	SMALLINT.
BYTEINT SMALLINT	INTEGER	INTEGER.
BYTEINT SMALLINT INTEGER BIGINT	BIGINT	BIGINT.
BYTEINT	DECIMAL(*m*,*n*) where *m* <= 18 and *m*-*n* >= 3	DECIMAL(18,*n*).
SMALLINT	DECIMAL(*m*,*n*) where *m* <= 18 and *m*-*n* >= 5	
INTEGER	DECIMAL(*m*,*n*) where *m* <= 18 and *m*-*n* >= 10	
DATE		
BYTEINT	DECIMAL(*m*,*n*) where *m* > 18 or *m*-*n* < 3	DECIMAL(38,*n*).
SMALLINT	DECIMAL(*m*,*n*) where *m* > 18 or *m*-*n* < 5	
INTEGER	DECIMAL(*m*,*n*) where *m* > 18 or *m*-*n* < 10	
DATE		
BIGINT	DECIMAL(*m*,*n*)	
DECIMAL(*m*,*n*)	DECIMAL(*k*,*j*) where max(*m*-*n*,*k*-*j*) + max(*j*,*n*) <= 18	DECIMAL(18,max(*j*,*n*)).
	DECIMAL(*k*,*j*) where max(*m*-*n*,*k*-*j*) + max(*j*,*n*) > 18	DECIMAL(38,max(*j*,*n*)).

IF one expression operand is ...	AND the other expression operand is ...	THEN Teradata Database compares the data as ...
BYTEINT SMALLINT INTEGER BIGINT DECIMAL(*m*,*n*) NUMBER(*m*,*n*) NUMBER(*m*) NUMBER(*,*n*) NUMBER	NUMBER(*k*,*j*) NUMBER(*k*) NUMBER(*,*j*) NUMBER	NUMBER
DATE	BYTEINT SMALLINT INTEGER	INTEGER.
	BIGINT	BIGINT.
	FLOAT	FLOAT.
FLOAT	BYTEINT SMALLINT INTEGER BIGINT DECIMAL(*m*,*n*) NUMBER(*m*,*n*) NUMBER(*m*) NUMBER(*,*n*) NUMBER	FLOAT.
Period	Character	Period.

Implicit Conversion of DateTime Types

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

IF one expression operand is ...	AND the other expression operand is ...	THEN Teradata Database compares the data as ...
TIMESTAMP	DATE[b]	DATE. See "Implicit TIMESTAMP-to-DATE Conversion" on page 730.
TIMESTAMP WITH TIME ZONE		
Interval[a]	Exact Numeric	Numeric. See "Implicit INTERVAL-to-Numeric Conversion" on page 657.

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

b. ANSIDate dateform mode or IntegerDate dateform mode

Data Types on Which Implicit Conversion is Not Performed

The following table identifies data types on which Teradata Database does not perform implicit type conversion.

Type	Rules
Byte	Byte data types can only be compared with byte data types. Attempts to compare a byte type with another type produces an error.
TIME	Teradata Database does not perform implicit type conversion from TIME to TIMESTAMP and from TIMESTAMP to TIME in comparison operations.
TIMESTAMP	
UDT	Teradata Database does not perform implicit type conversion on UDTs for comparison operations. A UDT value can only be compared with another value of the same UDT type. To compare UDTs with other data types, you must use explicit data type conversion. For more information, see Chapter 13: "Data Type Conversions."

Comparison of ANSI DateTime and Interval in USING Clause

External values for ANSI DateTime and Interval data are expressed as fixed length character strings in the designated client character set for the session.

When you import ANSI DateTime and Interval values with a USING phrase, you must explicitly cast them from the external character format to the proper ANSI DateTime and Interval types for comparison.

For example, consider the following statement, where the data type of the TimeField column is TIME(2):

```
USING (TimeVal CHARACTER(11), NumVal INTEGER)
UPDATE TABLE_1
SET TimeField=:TimeVal, NumField=:NumVal
WHERE CAST(:TimeVal AS TIME(2)) > TimeField;
```

Although you can use TimeVal CHAR(11) directly for assignment in this USING phrase, you must CAST the column data definition explicitly as TIME(2) in order to compare the field value TimeField in the table because TimeField is an ANSI TIME defined as TIME(2).

Proper Forms of DATE Types in Comparisons

A DATE operand must be submitted in the proper form in order to achieve a correct comparison.

Arithmetic on DATE operands causes an error if a created value is not a valid date. Therefore, although a date value can be submitted in integer form for comparison purposes, a column that contains date data should be defined as data type DATE, not INTEGER.

If an integer is used for input to DATE (this is *not* recommended), the way to enter the first date of the year 2000 is 1000101.

For more information, see "Teradata Date and Time Expressions" on page 841.

Proper forms for submitting a DATE operand are:

- An integer in the form (year-1900)*10000 + month*100 + day. The form YYMMDD is only valid for the years 1900 - 1999. For the years 2000 - 2099, the form is 1YYMMDD.
- As a character string in the same form as the date against which the compare is being done or as the date field the assignment is being done.
- A character string that is qualified with a data type phrase defining the appropriate data conversion, and a FORMAT phrase defining the format.
- As an ANSI date literal, which is always valid for a date comparison with any date format.

Examples

The following examples use a comparison operator on a value in the Employee.DOB column (defined as DATE FORMAT 'MMMbDDbYYYY') to illustrate correct forms for a DATE operand.

Example 1

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

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

Example 2

In the second example, the character string is entered in a form that agrees with the format of the DOB column.

```
SELECT *
FROM Employee
WHERE DOB = 'Mar 27 1942';
```

Example 3

In the third example, the value is entered as a character string, and so is cast with both a data type phrase (DATE) and a FORMAT phrase.

```
SELECT *
FROM Employee
WHERE DOB = CAST ('03/27/42' AS DATE FORMAT 'MM/DD/YY');
```

Example 4

In the fourth example, the value is entered as an ANSI date literal, which works regardless of the date format of the column.

```
SELECT *
FROM Employee
WHERE DOB = DATE '1942-03-27';
```

Character String Comparisons

Comparison of Character Strings of Unequal Length

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

Character Strings and Server Character Sets

When comparing character strings, data characters must have the same server character set. If they do not, then the system translates them using the implicit translation rules described in "Implicit Character-to-Character Conversion" on page 597.

Effect of Collation on Character String Comparisons

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

You can set the default collation to a sequence that is compatible with the character set for your session. Use the HELP SESSION SQL statement to determine the collation setting for your current session.

The availability of diacritical or Japanese character sets, and your default collation sequence are under the control of your database administrator.

To ensure that sorting and comparison of character data are identical with the same operations performed by the client, users on a Japanese language site should set collation to CHARSET_COLL.

For collation details, see:

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

Case Sensitivity

All character data, except for CLOBs, accessed in the execution of a Teradata SQL statement has an attribute of CASESPECIFIC or NOT CASESPECIFIC, either by default or by explicit designation. Character string comparisons use this attribute to determine whether the comparison is case blind or case specific. Case specificity does not apply to CLOBs.

This is not an ANSI SQL:2011 compatible attribute—ANSI does *all* character comparisons as the equivalent of CASESPECIFIC.

The CASESPECIFIC attribute has higher precedence over the NOT CASESPECIFC attribute:

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

The exception is comparisons on GRAPHIC character data, which are always CASESPECIFIC.

To apply a case specification attribute to a character string, you can:

- Use the default case specification for the session.

IF the session mode is ...	THEN the default case specification is ...
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. The exception is character data of type GRAPHIC, which is always CASESPECIFIC.

  Default case specification applies to all character data, including literals.
- Use the CASESPECIFIC or NOT CASESPECIFIC phrase with a character column in a CREATE TABLE or ALTER TABLE statement.

  For example:

  ```
  CREATE TABLE Students
    (StudentID INTEGER
    ,Firstname CHAR(10) CASESPECIFIC
    ,Lastname CHAR(20) NOT CASESPECIFIC);
  ```

  Table columns carry the attribute assigned at the time the columns were defined or altered unless a CASESPECIFIC or NOT CASESPECIFIC phrase is used in their access.
- Apply the CASESPECIFIC or NOT CASESPECIFIC phrase to a character expression in the comparison.

  For example, the following statement applies the CASESPECIFIC phrase to a character literal:

  ```
  SELECT *
  FROM Students
  WHERE Firstname = 'Ike' (CASESPECIFIC);
  ```

  Use this to override the default case specification for character data, or to override the case specification attribute assigned at the time a character column was defined or altered.

For case blind comparisons, any lowercase single byte Latin letters are converted to uppercase before comparison begins. The prepared strings are compared and any trailing pad characters are ignored.

A case blind comparison always considers lowercase and uppercase Cyrillic, Greek and full-width ASCII letters to be equivalent. To distinguish lowercase and uppercase Cyrillic, Greek, and fullwidth ASCII letters you must explicitly declare CASESPECIFIC comparison.

These options work for the KANJISJIS character set as if the data were first converted to the Unicode type and then the options applied.

Using UPPER for Case Blind Comparisons

Case blind comparisons can be accomplished using the UPPER function, to make sure a character string value contains no lowercase Latin letters.

The UPPER function is *not* the same as declaring a value UPPERCASE.

For a description of the UPPER function, see "UPPER" on page 1252.

Example

Consider the following query:

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

The behavior of the comparison Firstname = 'George' under different case specification attributes and session modes is described in the table that follows.

IF column Firstname is ...	THEN ...
CASESPECIFIC	• If the session mode is ANSI, then 'George' is CASESPECIFIC and the match succeeds for rows with Firstname containing 'George'. • If the session mode is Teradata, then "George' is NOT CASESPECIFIC and the match succeeds for rows with Firstname containing 'George'. When either character sting is CASESPECIFIC, the comparison is case specific.
NOT CASESPECIFIC	• If the session mode is ANSI, then 'George' is CASESPECIFIC and the match succeeds for rows with Firstname containing 'George'. When either character string is CASESPECIFIC, the comparison is case specific. • If the session mode is Teradata, then 'George' is NOT CASESPECIFIC and the match succeeds for rows with Firstname containing any combination of cases that spell the name George, such as 'george' or 'GEORGE' or 'George'. When both character strings are NOT CASESPECIFIC, the comparison is case blind.

Comparison of KANJI1 Characters

The following sections describe how Teradata Database compares KANJI1 characters.

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see "KANJI1 Character Set" in *International Character Set Support*.

Equality Comparison

Comparison of character strings, which can contain mixed single byte and multibyte character data, is handled as follows:

- If *expression_1* and *expression_2* have different server character sets, then they are converted to the same type. For details, see "Implicit Character-to-Character Translation" on page 599.
- If *expression_1* and *expression_2* are of different lengths, the shorter string is padded with enough pad characters to make both the same length.
- Session mode is identified:

In this mode ...	The default case specification for a character string is ...
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. Unless the CASESPECIFIC phrase is applied to one or both of the expressions, any simple Latin letters in both *expression_1* and *expression_2* are converted to uppercase before comparison begins.

To override the default case specification of a character expression, apply the CASESPECIFIC or NOT CASESPECIFIC phrase.

- Case specification is determined:

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

- Trailing pad characters are ignored.

Nonequality Comparison

Nonequality comparisons are handled as follows:

1 If *expression_1* and *expression_2* are of different lengths, the shorter string is padded with enough pad characters to make both the same length.

2 Session mode is identified.

In this mode ...	The default case specification for a character string is ...
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. Unless the CASESPECIFIC qualifier is applied to one or both of the expressions, any simple Latin letters in both *expression_1* and *expression_2* are converted to uppercase before comparison begins.

To override the default case specification of a character expression, apply the CASESPECIFIC or NOT CASESPECIFIC phrase.

3 Characters identified as single byte characters under the current character set are converted according to the collation sequence in effect for the session.

4 For the KanjiEUC character set, the ss3 0x8F character is converted to 0xFF. This means that a user-defined KanjiEUC codeset 3 is not properly ordered with respect to other KanjiEUC code sets.

The ordering of other KanjiEUC codesets is proper; that is, ordering is the same as the binary ordering on the client system.

5 The prepared strings are compared and trailing pad characters are ignored.

Nonequality comparisons involve the collation in effect for the session. Five collations are available:

- EBCDIC
- ASCII
- MULTINATIONAL
- CHARSET_COLL
- JIS_COLL

Collation can be set at the user level with the COLLATION option of the CREATE USER or MODIFY USER statements, and at the session level with the [[.]SET] SESSION COLLATION statement or the CLIv2 CHARSET call.

If the MULTINATIONAL collation sequence is in effect, the collation sequence of a Japanese language site is determined by the collation setting installed during start-up.

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

Comparison Operators and the DEFAULT Function in Predicates

The DEFAULT function returns the default value of a column. It has two forms: one that specifies a column name and one that omits the column name.

Predicates using comparison operators support both forms of the DEFAULT function, but when the DEFAULT function omits the column name, the following conditions must be true:

- The comparison can only involve a single column reference.
- The DEFAULT function cannot be part of an expression.

For example, the following statement uses DEFAULT to compare the values of the Dept_No column with the default value of the Dept_No column. Because the comparison operation involves a single column reference, Teradata Database can derive the column context of the DEFAULT function even though the column name is omitted.

```
SELECT * FROM Employee WHERE Dept_No < DEFAULT;
```

Note that if the DEFAULT function evaluates to null, the predicate is unknown and the WHERE condition is false.

For more information on the DEFAULT function, see "DEFAULT" on page 293.

DECODE

Purpose

Compares *expr* to each *search* parameter with its corresponding *result* parameter.

Syntax

1145-047A

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
expr	a numeric or character argument. • If *expr* is equal to one of the *search* arguments, the function returns the corresponding *result* value. • If *expr* is not equal to any of the *search* arguments, the functions returns *default*, if the default value is specified. • If *expr* is NULL, the function returns the result of the first search parameter that is NULL.
search	a numeric or character argument. DECODE supports 1-10 *search* arguments tied to an equal number of *result* arguments.
result	a numeric or character argument. DECODE supports 1-10 *result* arguments tied to an equal number of *search* arguments.
default	a numeric or character argument. If *default* is not specified and there are no matches, the function returns NULL.

SYSLIB

Name of the database where the function is located.

ANSI Compliance

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

Invocation

DECODE is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must be one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, NUMBER, CHAR, VARCHAR

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

DECODE is a scalar function whose return value data type depends on the data type associated with the *result* parameter passed into the function.

- If *result* or *default* is a numeric type, DECODE determines which argument has the highest precedence and converts the other *result* or *default* arguments to that data type and returns that data type. For details about the order of precedence, see "Compatible Types" in *SQL External Routine Programming*.

  If that data type is DECIMAL/NUMERIC and the precision and scale of the *result* or *default* argument is different, the precision and scale of the return type is set to achieve the maximum precision possible.

  For example, if the *result* / *default* argument are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return type would need 3 digits to the left of the decimal point and 7 digits to the right of the decimal point to avoid any reduction in precision. In this case, the return data type is set to DECIMAL(10,7).

  In cases where it is not possible to maintain the maximum precision, the data is rounded according to the DBS Control RoundHalfWayMagUp field. For example, if the *result* and *default* argument are DECIMAL(32, 8) and DECIMAL(30, 28), the return type is DECIMAL(38,14). This allows for 24 digits to the left of the decimal point required for DECIMAL(32, 8) and 14 digits to the right of the decimal point. If the DECIMAL(30,28) *result* or *default* argument is the greatest value, it is rounded to 14 places to the right of the decimal point.

  If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is

changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).

- If *result* or *default* is a character data type, the function returns a VARCHAR in the character set of the first result argument.

For information on the default data type format for DOUBLE PRECISION, see *SQL Data Types and Literals*.

Example 1

The following query:

```
SELECT DECODE(country_id, 1, 'United States',
                          2, 'England',
                          3, 'France',
                          'United States')
    FROM customers;
```

returns:

- 'United States' if the country_id is 1
- 'England' if the country_id is 2
- 'France' if the country_id is 3
- 'United States' if the country_id is not equal to 1, 2, or 3

Example 2

The following query

```
SELECT DECODE(country_id, 1, 'United States',
                          2, 'England')
    FROM customers;
```

returns:

- 'United States' if the country_id is 1
- 'England' if country_id is 2
- NULL if country_id isn't in the range 1 to 2

Example 3

The following query:

```
SELECT DECODE(country_id, 1, 'United States',
                          2, 'England',
                          NULL, 'France')
    FROM customers;
```

returns:

- 'United States' if the country_id is 1
- 'England' if the country_id is 2
- NULL if the country_id is NULL
- NULL if the country_id is not equal to 1, 2, or NULL

GREATEST

Purpose

Returns the largest value in the list of input arguments.

Syntax

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
numeric_value	a numeric value.
string_value	a string value.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Invocation

GREATEST is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, NUMBER, CHAR, or VARCHAR

All of the input arguments must be the same data type or else the types must be compatible.

Result Type

If the input arguments are numeric types, the function determines which argument has the highest precedence, converts the other arguments to that data type, and returns that data type. For details about the order of precedence, see "Compatible Types" in *SQL External Routine Programming*.

If the data type is DECIMAL/NUMERIC and the precision and scale of the input arguments are different, the precision and scale of the return type is set to achieve the maximum precision possible. For example, if the input arguments are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return type would need 3 digits to the left of the decimal point and 7 digits to the right of the decimal point to avoid any reduction in precision. In this case, the return data type is set to DECIMAL(10,7).

In cases where it is not possible to maintain the maximum precision, the data is rounded according to the DBS Control Record RoundHalfWayMagUp field. For example, if the input arguments are DECIMAL(32, 8) and DECIMAL(30, 28), the return type is DECIMAL(38,14). This allows for 24 digits to the left of the decimal point (required for the DECIMAL(32,8) argument), and 14 digits to the right of the decimal point. If the DECIMAL(30,28) input argument is the greatest value, it is rounded to 14 places to the right of the decimal point.

If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).

If the input arguments are character types, the function converts the 2nd through 10th arguments to the data type of the 1st argument and returns the type as VARCHAR in the character set of the 1st argument.

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

Usage Notes

If the arguments are character types, the string comparison uses non-padded comparison semantics. Character comparison is binary and based on the numerical codes of the characters. The string is treated as a sequence of bytes for the comparison rather than character by character.

If any of the input arguments is NULL, the function returns NULL.

Example 1

The following query returns the result 13.

```
SELECT GREATEST(13, 6);
```

Example 2

In the following query, if the input arguments have data types of DECIMAL(4,2) and DECIMAL(5,4), the return data type is DECIMAL(6,4) and the result value is 13.1200.

```
SELECT GREATEST(13.12, 6.1234);
```

Example 3

The following query returns the result 'apples'.

```
SELECT GREATEST('apples', 'alpha');
```

LEAST

Purpose

Returns the least value among a list of *numeric_values* or *string_values*.

Syntax

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
numeric_value	a numeric argument. LEAST supports 1-10 numeric values. If *numeric_value* is the data type of the first argument, the return data type is numeric. The remaining arguments in the input list must be the same or compatible types. If either input parameter is NULL, NULL is returned.
string_value	a character argument. LEAST supports 1-10 string values. If *string_value* is the data type of the first argument, the return data type is string. The remaining arguments in the input list must be the same or compatible types. String comparison uses non-padded comparison semantics. Character comparison is binary and based on the numerical codes of the characters and the string is treated as a sequence of bytes for the comparison rather than character by character. If either input parameter is NULL, NULL is returned.

TD_SYSFNLIB

Name of the database where the function is located.

ANSI Compliance

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

Invocation

LEAST is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *string_value* = CHAR or VARCHAR

Result Type

LEAST is a scalar function whose return value data type depends upon the data type associated with the *numeric_value* or *string_value* passed to the function.

- If *numeric_value* is a numeric data type, LEAST determines which argument has the highest precedence, converts the other arguments to that data type, and returns that data type. For details about the order of precedence, see "Compatible Types" in *SQL External Routine Programming*.

  If the data type is DECIMAL/NUMERIC and the precision and scale of the input arguments are different, the precision and scale of the return data types are set to achieve the maximum precision possible. For example, if the input arguments are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return data type would need 3 digits to the left of the decimal point and 7 digits to the right of the decimal point to avoid any reduction in precision. In this case, the return data type is set to DECIMAL(10,7).

  In cases where it is not possible to maintain the maximum precision, the data is rounded according to the DBS Control Record RoundHalfWayMagUp field. For example, if the input arguments are DECIMAL(32, 8) and DECIMAL(30, 28), the return data type is DECIMAL(38,14). This allows for 24 digits to the left of the decimal point (required for the DECIMAL(32,8) parameter), and 14 digits to the right of the decimal point. If the DECIMAL(30,28) input argument is the least value, it is rounded to 14 places to the right of the decimal point.

  If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).

- If *string_value* is a character data type, LEAST converts the 2nd through 10th argument to the data type of the 1st argument and returns values as VARCHAR in the character set of the first argument.

Example 1

The following query returns 6.

```
SELECT LEAST(13, 6);
```

Example 2

In the following query, if the input arguments have data types of DECIMAL(5,4) and DECIMAL(4,2), the return data type is DECIMAL(6,4) and the return value is 1.1234.

```
SELECT LEAST(1.1234, 36.12);
```

Example 3

The following query returns 'alpha'.

```
SELECT LEAST('apples', 'alpha');
```

CHAPTER 11 Compression / Decompression Functions

This chapter describes the functions that you can use with algorithmic compression (ALC) to compress and uncompress column data of character or byte type. Compression of data reduces space usage and the size of disk I/Os.

For a detailed comparison between the compression functions and guidelines for choosing a compression function, see "Reducing Space Usage with Data Compression" in *Database Administration*.

If the compression and decompression functions described in this chapter are not optimal for your data, you can write your own user-defined functions (UDFs) to compress and uncompress table columns.

Prerequisites

The functions in this chapter are embedded services system functions. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Related Topics

FOR more information on...	SEE...
ALC	• "COMPRESS and DECOMPRESS Phrases" in *SQL Data Types and Literals*. • "CREATE TABLE" in *SQL Data Definition Language*.
writing UDFs for ALC	• "Defining Functions for Algorithmic Compression" in *SQL External Routine Programming*. • "CREATE TABLE" in *SQL Data Definition Language*.
compression methods supported by Teradata Database and a comparison of the various methods	"Reducing Space Usage with Data Compression" in *Database Administration*.

CAMSET

Purpose

Compresses the specified Unicode character data into the following possible values using a proprietary Teradata algorithm:

- partial byte values (for example, 4-bit digits or 5-bit alphabetic letters)
- one byte values (for example, other Latin characters)
- two byte values (for example, other Unicode characters)

Syntax

```
┌──────────────────┐
│                  │
└─ TD_SYSFNLIB. ───┘── CAMSET (Unicode_string) ──────────────────────────────
```

1101A781

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
Unicode_string	a Unicode character string or string expression.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET UNICODE, where the maximum supported size (*n*) is 32000. You can also pass arguments with data types that can be converted to VARCHAR(32000) CHARACTER SET UNICODE using the implicit data type conversion rules that apply to UDFs. For example, CAMSET(CHAR) is allowed because it can be implicitly converted to CAMSET(VARCHAR).

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

The input to this function must be Unicode character data.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARBYTE(64000).

Usage Notes

Uncompressed character data in Teradata Database requires 2 bytes per character when storing Unicode data. CAMSET takes Unicode character input, compresses it into partial byte, one byte, or two byte values, and returns the compressed result.

CAMSET provides best results for short or medium Unicode strings that:

- contain mainly digits and English alphabet letters.
- do not frequently switch between:
  - lowercase and uppercase letters.
  - digits and letters.
  - Latin and non-Latin characters.

For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Database Administration*.

Although you can call the function directly, CAMSET is normally used with algorithmic compression (ALC) to compress table columns. If CAMSET is used with ALC, nulls are also compressed if those columns are nullable.

For more information about ALC, see "COMPRESS and DECOMPRESS Phrases" in *SQL Data Types and Literals*.

Restrictions

CAMSET currently can only compress Unicode characters from U+0000 to U+00FF.

Uncompressing Data Compressed with CAMSET

To uncompress Unicode data that was compressed using CAMSET, use the DECAMSET function. See "DECAMSET" on page 530.

Example 1

In this example, the Unicode values in the Description column are compressed using the CAMSET function with ALC. The DECAMSET function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE CHARACTER SET UNICODE,
    Description VARCHAR(1000) CHARACTER SET UNICODE
       COMPRESS USING TD_SYSFNLIB.CAMSET
       DECOMPRESS USING TD_SYSFNLIB.DECAMSET);
```

Example 2

Given the following table definition:

```
CREATE TABLE Pendants
   (ItemNo INTEGER,
    Description VARCHAR(100) CHARACTER SET UNICODE);
```

The following query returns the compressed values of the Description column.

```
SELECT TD_SYSFNLIB.CAMSET(Pendants.Description);
```

CAMSET_L

Purpose

Compresses the specified Latin character data into the following possible values using a proprietary Teradata algorithm:

- partial byte values (for example, 4-bit digits or 5-bit alphabetic letters)
- one byte values (for example, other Latin characters)

Syntax

```
┬──────────────┬─ CAMSET_L (Latin_string) ──────────────────────────────
└─ TD_SYSFNLIB. ─┘
```

1101A782

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
Latin_string	a Latin character string or string expression.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET LATIN, where the maximum supported size (*n*) is 64000. You can also pass arguments with data types that can be converted to VARCHAR(64000) CHARACTER SET LATIN using the implicit data type conversion rules that apply to UDFs. For example, CAMSET_L(CHAR) is allowed because it can be implicitly converted to CAMSET_L(VARCHAR).

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

The input to this function must be Latin character data.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARBYTE(64000).

Usage Notes

Uncompressed character data in Teradata Database requires one byte per character when storing Latin character data. CAMSET_L takes Latin character input, compresses it into partial byte or one byte values, and returns the compressed result.

CAMSET_L provides best results for short or medium Latin strings that:

- contain mainly digits and English alphabet letters.
- do not frequently switch between:
  - lowercase and uppercase letters.
  - digits and letters.

For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Database Administration*.

Although you can call the function directly, CAMSET_L is normally used with algorithmic compression (ALC) to compress table columns. If CAMSET_L is used with ALC, nulls are also compressed if those columns are nullable.

For more information about ALC, see "COMPRESS and DECOMPRESS Phrases" in *SQL Data Types and Literals*.

Uncompressing Data Compressed with CAMSET_L

To uncompress Latin character data that was compressed using CAMSET_L, use the DECAMSET_L function. See "DECAMSET_L" on page 532.

Example 1

In this example, the Latin values in the Description column are compressed using the CAMSET_L function with ALC. The DECAMSET_L function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE CHARACTER SET LATIN,
    Description VARCHAR(1000) CHARACTER SET LATIN
       COMPRESS USING TD_SYSFNLIB.CAMSET_L
       DECOMPRESS USING TD_SYSFNLIB.DECAMSET_L);
```

Example 2

Given the following table definition:

```
CREATE TABLE Pendants
   (ItemNo INTEGER,
    Description VARCHAR(100) CHARACTER SET LATIN);
```

The following query returns the compressed values of the Description column.

```
SELECT TD_SYSFNLIB.CAMSET_L(Pendants.Description);
```

DECAMSET

Purpose

Uncompresses the Unicode data that was compressed using the CAMSET function.

Syntax

```
┬──────────────┬─ DECAMSET (compressed_string) ──────────────────────────
└ TD_SYSFNLIB. ┘
```

1101A784

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
compressed_string	Unicode character data that was compressed using the CAMSET function.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(*n*), where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the CAMSET function.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARCHAR(32000) CHARACTER SET UNICODE.

Usage Notes

DECAMSET takes Unicode data that was compressed using the CAMSET function, uncompresses it, and returns an uncompressed Unicode character string as the result.

Although you can call the function directly, DECAMSET is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with CAMSET.

For more information about ALC, see “COMPRESS and DECOMPRESS Phrases” in *SQL Data Types and Literals.*

Example

In this example, the Unicode values in the Description column are compressed using the CAMSET function with ALC. The DECAMSET function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE CHARACTER SET UNICODE,
    Description VARCHAR(1000) CHARACTER SET UNICODE
       COMPRESS USING TD_SYSFNLIB.CAMSET
       DECOMPRESS USING TD_SYSFNLIB.DECAMSET);
```

DECAMSET_L

Purpose

Uncompresses the Latin data that was compressed using the CAMSET_L function.

Syntax

```
┬──────────────┬─ DECAMSET_L (compressed_string) ──────────────────────────
└─ TD_SYSFNLIB.┘
```

1101A783

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
compressed_string	Latin character data that was compressed using the CAMSET_L function.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(*n*), where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the CAMSET_L function.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARCHAR(64000) CHARACTER SET LATIN.

Usage Notes

DECAMSET_L takes Latin data that was compressed using the CAMSET_L function, uncompresses it, and returns an uncompressed Latin character string as the result.

Although you can call the function directly, DECAMSET_L is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with CAMSET_L.

For more information about ALC, see “COMPRESS and DECOMPRESS Phrases” in *SQL Data Types and Literals*.

Example

In this example, the Latin values in the Description column are compressed using the CAMSET_L function with ALC. The DECAMSET_L function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE CHARACTER SET LATIN,
    Description VARCHAR(1000) CHARACTER SET LATIN
       COMPRESS USING TD_SYSFNLIB.CAMSET_L
       DECOMPRESS USING TD_SYSFNLIB.DECAMSET_L);
```

LZCOMP

Purpose

Compresses the specified Unicode character data using the Lempel-Ziv algorithm.

Syntax

```
┬──────────────┬─ LZCOMP (Unicode_string) ──────────────────────────────
└─ TD_SYSFNLIB.─┘
```

1101A766

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
Unicode_string	a Unicode character string or string expression.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET UNICODE, where the maximum supported size (*n*) is 32000. You can also pass arguments with data types that can be converted to VARCHAR(32000) CHARACTER SET UNICODE using the implicit data type conversion rules that apply to UDFs. For example, LZCOMP(CHAR) is allowed because it can be implicitly converted to LZCOMP(VARCHAR).

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

The input to this function must be Unicode character data.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARBYTE(64000).

Usage Notes

Uncompressed character data in Teradata Database requires 2 bytes per character when storing Unicode data. LZCOMP takes Unicode character input, compresses it using the Lempel-Ziv algorithm, and returns the compressed result.

See http://zlib.net for information about the compression algorithm used by LZCOMP.

LZCOMP provides good compression results for long Unicode strings, but might not be as effective for short strings. It can also provide good results for medium strings that have many repeating characters.

For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Database Administration*.

Although you can call the function directly, LZCOMP is normally used with algorithmic compression (ALC) to compress table columns. If LZCOMP is used with ALC, nulls are also compressed if those columns are nullable.

For more information about ALC, see "COMPRESS and DECOMPRESS Phrases" in *SQL Data Types and Literals.*

Uncompressing Data Compressed with LZCOMP

To uncompress Unicode data that was compressed using LZCOMP, use the LZDECOMP function. See "LZDECOMP" on page 538.

Example 1

In this example, the Unicode values in the Description column are compressed using the LZCOMP function with ALC. The LZDECOMP function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE CHARACTER SET UNICODE,
    Description VARCHAR(1000) CHARACTER SET UNICODE
       COMPRESS USING TD_SYSFNLIB.LZCOMP
       DECOMPRESS USING TD_SYSFNLIB.LZDECOMP);
```

Example 2

Given the following table definition:

```
CREATE TABLE Pendants
   (ItemNo INTEGER,
    Description VARCHAR(100) CHARACTER SET UNICODE);
```

The following query returns the compressed values of the Description column.

```
SELECT TD_SYSFNLIB.LZCOMP(Pendants.Description);
```

LZCOMP_L

Purpose

Compresses the specified Latin character data using the Lempel-Ziv algorithm.

Syntax

```
┌──────────────────┐
│                  │
└─ TD_SYSFNLIB. ───┘── LZCOMP_L (Latin_string) ──────────────────────────────
```

1101A765

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
Latin_string	a Latin character string or string expression.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET LATIN, where the maximum supported size (*n*) is 64000. You can also pass arguments with data types that can be converted to VARCHAR(64000) CHARACTER SET LATIN using the implicit data type conversion rules that apply to UDFs. For example, LZCOMP_L(CHAR) is allowed because it can be implicitly converted to LZCOMP_L(VARCHAR).

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

The input to this function must be Latin character data.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARBYTE(64000).

Usage Notes

Uncompressed character data in Teradata Database requires one byte per character when storing Latin character data. LZCOMP_L takes Latin character input, compresses it using the Lempel-Ziv algorithm, and returns the compressed result.

See http://zlib.net for information about the compression algorithm used by LZCOMP_L.

LZCOMP_L provides good compression results for long Latin character strings, but might not be as effective for short strings. It can also provide good results for medium strings that have many repeating characters.

For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Database Administration*.

Although you can call the function directly, LZCOMP_L is normally used with algorithmic compression (ALC) to compress table columns. If LZCOMP_L is used with ALC, nulls are also compressed if those columns are nullable.

For more information about ALC, see "COMPRESS and DECOMPRESS Phrases" in *SQL Data Types and Literals.*

Uncompressing Data Compressed with LZCOMP_L

To uncompress Latin data that was compressed using LZCOMP_L, use the LZDECOMP_L function. See "LZDECOMP_L" on page 540.

Example 1

In this example, the Latin values in the Description column are compressed using the LZCOMP_L function with ALC. The LZDECOMP_L function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE CHARACTER SET LATIN,
    Description VARCHAR(1000) CHARACTER SET LATIN
       COMPRESS USING TD_SYSFNLIB.LZCOMP_L
       DECOMPRESS USING TD_SYSFNLIB.LZDECOMP_L);
```

Example 2

Given the following table definition:

```
CREATE TABLE Pendants
   (ItemNo INTEGER,
    Description VARCHAR(100) CHARACTER SET LATIN);
```

The following query returns the compressed values of the Description column.

```
SELECT TD_SYSFNLIB.LZCOMP_L(Pendants.Description);
```

LZDECOMP

Purpose

Uncompresses the Unicode data that was compressed using the LZCOMP function.

Syntax

```
┬──────────────┬─ LZDECOMP (compressed_string) ──────────────────────────
└─ TD_SYSFNLIB.┘
```

1101A763

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
compressed_string	Unicode character data that was compressed using the LZCOMP function.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(*n*), where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the LZCOMP function.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARCHAR(32000) CHARACTER SET UNICODE.

Usage Notes

LZDECOMP takes Unicode data that was compressed using the LZCOMP function, uncompresses it, and returns an uncompressed Unicode character string as the result.

See http://zlib.net for information about the decompression algorithm used by LZDECOMP.

Although you can call the function directly, LZDECOMP is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with LZCOMP.

For more information about ALC, see “COMPRESS and DECOMPRESS Phrases” in *SQL Data Types and Literals.*

Example

In this example, the Unicode values in the Description column are compressed using the LZCOMP function with ALC. The LZDECOMP function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE CHARACTER SET UNICODE,
    Description VARCHAR(1000) CHARACTER SET UNICODE
       COMPRESS USING TD_SYSFNLIB.LZCOMP
       DECOMPRESS USING TD_SYSFNLIB.LZDECOMP);
```

LZDECOMP_L

Purpose

Uncompresses the Latin data that was compressed using the LZCOMP_L function.

Syntax

```
┬──────────────┬─ LZDECOMP_L (compressed_string) ──────────────────────────
└─ TD_SYSFNLIB.┘
```

1101A764

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
compressed_string	Latin character data that was compressed using the LZCOMP_L function.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(*n*), where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the LZCOMP_L function.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARCHAR(64000) CHARACTER SET LATIN.

Usage Notes

LZDECOMP_L takes Latin data that was compressed using the LZCOMP_L function, uncompresses it, and returns an uncompressed Latin character string as the result.

See http://zlib.net for information about the decompression algorithm used by LZDECOMP_L.

Although you can call the function directly, LZDECOMP_L is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with LZCOMP_L.

For more information about ALC, see “COMPRESS and DECOMPRESS Phrases” in *SQL Data Types and Literals.*

Example

In this example, the Latin values in the Description column are compressed using the LZCOMP_L function with ALC. The LZDECOMP_L function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE CHARACTER SET LATIN,
    Description VARCHAR(1000) CHARACTER SET LATIN
       COMPRESS USING TD_SYSFNLIB.LZCOMP_L
       DECOMPRESS USING TD_SYSFNLIB.LZDECOMP_L);
```

TD_LZ_COMPRESS

Purpose

Compresses any supported ALC data type or predefined type data using the Lempel-Ziv algorithm.

Syntax

```
┬──────────────┬─ TD_LZ_COMPRESS ( expression ) ──────────────────────────
└─ TD_SYSFNLIB. ┘
```

1145-001A

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
expression	• any supported ALC data type when using the function as the compression routine for ALC on a column. • any supported ALC data types and all predefined data types and Distinct UDTs when calling the function.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have one of the following data types:

- BYTE
- CHARACTER
- VARBYTE
- VARCHAR
- TIME
- TIMESTAMP
- System-defined UDTs, such as ST_Geometry and XML.
- Distinct UDTs, including LOB UDTs, ARRAY or Period data types.

Data types must match the result data type of the corresponding decompression function, TD_LZ_DECOMPRESS.

Result Type

The result data types are VARBYTE or BLOB.

Usage Notes

TD_LZ_COMPRESS can only be used with data types that are valid for MVC and ALC. Numeric data types are valid when calling the function in an SQL statement, but not when using the function for ALC on a column.

If TD_LZ_COMPRESS is used with ALC, nulls are also compressed if those columns are nullable.

Calling the function is only appropriate for non-LOB data types.

Uncompressing Data Compressed with TD_LZ_COMPRESS

To uncompress data that was compressed using TD_LZ_COMPRESS, use "TD_LZ_DECOMPRESS" on page 544.

Related Topics

For information on ...	See ...
ALC	"COMPRESS and DECOMPRESS Phrases" in *SQL Data Types and Literals*.
the Lempel-Ziv algorithm	http://zlib.net.
comparisons between the different Teradata-supplied compression functions and guidelines for choosing a compression function suitable for your needs	*Database Administration*.

TD_LZ_DECOMPRESS

Purpose

Uncompresses predefined type data that was compressed using TD_LZ_COMPRESS or UDTs.

Syntax

```
┬──────────────┬─ TD_LZ_DECOMPRESS ( expression ) ──────────────────────────
└ TD_SYSFNLIB. ┘
```

1145-002A

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
expression	data that was compressed using the TD_LZ_COMPRESS function, represented as VARBYTE, BLOB, or a combination of the two.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have the same data type as those of the corresponding compress function, TD_LZ_COMPRESS. The input should be of type VARBYTE, BLOB, or a combination of the two.

Result Type

The result data type always matches the ALC-allowable data types.

Usage Notes

TD_LZ_DECOMPRESS is normally used with ALC to uncompress table columns previously compressed with TD_LZ_COMPRESS.

Calling the function directly requires using the RETURNS clause specifying the desired result data type because the return data type for TD_LZ_DECOMPRESS is ANYTYPE. Calling the function directly is not recommended for large data types, such as LOBs.

For an example of the use of RETURN clause to return a user-specified ALC-allowable data type, see TD_ANYTYPE data type in “Parameter Data Types” in *SQL Data Types and Literals*.

See http://zlib.net for information about the decompression algorithm used by TD_LZ_DECOMPRESS.

For more information about ALC, see “COMPRESS and DECOMPRESS Phrases” in *SQL Data Types and Literals.*

TS_COMPRESS

Purpose

Compresses TIME and TIMESTAMP with or without time zone to the minimum possible bytes.

Syntax

```
┬──────────────┬── TS_COMPRESS ── ( td_anytype ) ──────────────────────────
└─ TD_SYSFNLIB.─┘
```

1145-034A

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
td_anytype	in this function TIME or TIMESTAMP with or without time zone data types.

ANSI Compliance

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

Argument Type and Rules

The output of this function is the input to the TS_DECOMPRESS function.

Though the input argument type is *td_anytype*, the algorithm for this function only supports TIME and TIMESTAMP data types. Each value in TIME and TIMESTAMP (for example, year, month, or day) is compressed into the minimum possible bits individually and then concatenated.

Result Type

The result data type is VARBYTE(20).

Usage Notes

When creating algorithmic compression (ALC) columns, you should include the compression and decompression functions by which the column is going to be compressed and uncompressed respectively.

When you insert data into ALC columns, the data is compressed. When you select ALC columns the data is uncompressed.

Uncompressed Data for the Following Data Types...	Requires the Following Bytes...
TIME	6
TIMESTAMP	10
TIME WITH TIME ZONE	8
TIMESTAMP WITH TIME ZONE	12

Example 1

Uncompressed data requires 6 bytes for the TIME data type, broken down into bytes, possible range and minimum possible bits as shown in the following table.

Hour	Byte	Possible Range	Minimum Possible Bits
Hour	1	00 23	5
Minute	1	00 59	6
Second	4	00 000000 61.999999	6 to 26 depending on the seconds precision

If you insert the following value for TIME:

```
INSERT into t1(1, TIME'03:38:06');
```

as specified in the following SQL statement:

```
CREATE TABLE table t1(pk int, col1 time(0) compress using ts_compress
decompress using ts_decompress);
```

the required bits are as follows:

- 3 hours = 5
- 38 minutes = 6
- 6 seconds = 6

The total number of required bits needed to represent TIME is 17. The minimum number of bytes is 3. Since TIME (without TIMEZONE) takes 6 bytes, TS_COMPRESS compresses 6 bytes to 3. The function saves 3 bytes.

For more information about ALC, see *SQL Data Types and Literals*.

Example 2

The following SQL statement creates a TIMESTAMP column in table_t that is both compressed and uncompressed.

```
CREATE TABLE table t_timestamp(i int, j timestamp(0) compress using
td_sysfnlib.ts_compress decompress using td_sysfnlib.ts_decompress);
```

TS_DECOMPRESS

Purpose

Uncompresses TIME and TIMESTAMP data with or without time zone that the TS_COMPRESS function compressed.

Syntax

```
┬──────────────────┬── TS_DECOMPRESS ── ( compressed_string ) ──────────────────
└─ TD_SYSFNLIB. ───┘
```

1145-035A

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
compressed_string	data that was compressed using TS_COMPRESS.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(n), where the maximum supported size (n) is 20.

The input to this function must be the output result of the TS_COMPRESS function.

Result Type

The result data type is *td_anytype*.

Example

The following SQL statement creates a table a TIMESTAMP column that is both compressed and uncompressed.

```
CREATE table t_time(i int, j time(6) compress using
td_sysfnlib.ts_compress decompress using td_sysfnlib.ts_decompress);
```

If you insert the following TIME values into column j:

```
INSERT into t_time(1, TIME'06:26:44.820000');
INSERT into t_time(2, TIME'10:26:44.820000');
```

The following SQL statement:

```
SELECT j from t_time;
```

returns the 2 uncompressed TIME values that were compressed when the values were inserted into the table_t.

TransUnicodeToUTF8

Purpose

Compress the specified Unicode character data into UTF8 format.

Syntax

```
┌──────────────────────────┐
│                          │
└─ TD_SYSFNLIB. ───────────┘─ TransUnicodeToUTF8 (Unicode_string) ──────────────────────────
```

1101A771

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
Unicode_string	a Unicode character string or string expression.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET UNICODE, where the maximum supported size (*n*) is 32000. You can also pass arguments with data types that can be converted to VARCHAR(32000) CHARACTER SET UNICODE using the implicit data type conversion rules that apply to UDFs. For example, TransUnicodeToUTF8(CHAR) is allowed because it can be implicitly converted to TransUnicodeToUTF8(VARCHAR).

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

The input to this function must be Unicode character data.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARBYTE(64000).

Usage Notes

TransUnicodeToUTF8 compresses the specified Unicode character data into UTF8 format, and returns the compressed result. This is useful when the input data is predominantly Latin characters because UTF8 uses one byte to represent Latin characters and Unicode uses 2 bytes.

TransUnicodeToUTF8 provides good compression for Unicode strings of any length and is best used:

- On a Unicode column that contains mostly US-ASCII characters
- When the data frequently switches between:
  - Uppercase and lowercase letters
  - Digits and letters
  - Latin and non-Latin characters
- When the data is very dynamic (under frequent update)

For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Database Administration*.

Although you can call the function directly, TransUnicodeToUTF8 is normally used with algorithmic compression (ALC) to compress table columns. If TransUnicodeToUTF8 is used with ALC, nulls are also compressed if those columns are nullable.

For more information about ALC, see "COMPRESS and DECOMPRESS Phrases" in *SQL Data Types and Literals.*

Restrictions

TransUnicodeToUTF8 can only compress character values in the 7-bit ASCII character range, from U+0000 to U+007F, also known as US-ASCII.

Uncompressing Data Compressed with TransUnicodeToUTF8

To uncompress Unicode data that was compressed using TransUnicodeToUTF8, use the TransUTF8ToUnicode function. See "TransUTF8ToUnicode" on page 552.

Example

In this example, assume that the default server character set is UNICODE. The values of the Description column are compressed using the TransUnicodeToUTF8 function with ALC, which stores the Unicode input in UTF8 format. The TransUTF8ToUnicode function uncompresses the previously compressed values.

```
CREATE TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE,
    Description VARCHAR(1000)
       COMPRESS USING TD_SYSFNLIB.TransUnicodeToUTF8
       DECOMPRESS USING TD_SYSFNLIB.TransUTF8ToUnicode);
```

TransUTF8ToUnicode

Purpose

Uncompresses the Unicode data that was compressed using the TransUnicodeToUTF8 function.

Syntax

```
┌──────────────────┐
│                  │
└─ TD_SYSFNLIB. ───┘ TransUTF8ToUnicode (compressed_string) ──────────────────────
                                                                    1101A770
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
compressed_string	Unicode character data that was compressed using the TransUnicodeToUTF8 function.

ANSI Compliance

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

Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(*n*), where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the TransUnicodeToUTF8 function.

If you specify NULL as input, the function returns NULL.

Result Type

The result data type is VARCHAR(32000) CHARACTER SET UNICODE

Usage Notes

TransUTF8ToUnicode takes Unicode data that was compressed using the TransUnicodeToUTF8 function, uncompresses it, and returns an uncompressed Unicode character string as the result.

Although you can call the function directly, TransUTF8ToUnicode is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with TransUnicodeToUTF8.

For more information about ALC, see "COMPRESS and DECOMPRESS Phrases" in *SQL Data Types and Literals.*

Example

In this example, assume that the default server character set is UNICODE. The values of the Description column are compressed using the TransUnicodeToUTF8 function with ALC, which stores the Unicode input in UTF8 format. The TransUTF8ToUnicode function uncompresses the previously compressed values.

```
CREATE TABLE Pendants
   (ItemNo INTEGER,
    Gem CHAR(10) UPPERCASE,
    Description VARCHAR(1000)
       COMPRESS USING TD_SYSFNLIB.TransUnicodeToUTF8
       DECOMPRESS USING TD_SYSFNLIB.TransUTF8ToUnicode);
```

CHAPTER 12 CASE Expressions

This chapter describes SQL CASE expressions.

CASE

Purpose

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

ANSI Compliance

This is ANSI SQL:2011 compliant.

Overview

CASE provides an efficient and powerful method for application developers to change the representation of data, permitting conversion without requiring host program intervention.

For example, you could code employee status as 1 or 2, meaning full-time or part-time, respectively. For efficiency, the system stores the numeric code but prints or displays the appropriate textual description in reports. This storage and conversion is managed by Teradata Database.

In addition, CASE permits applications to generate nulls based on information derived from the database, again without host program intervention. Conversely, CASE can be used to convert a null into a value.

Two Forms of CASE Expressions

CASE expressions are specified in these forms:

- Valued CASE is described under "Valued CASE Expression" on page 556.
- Searched CASE is described under "Searched CASE Expression" on page 561.

CASE Shorthands for Handling Nulls

Two shorthand forms of CASE are provided to handle nulls:

- COALESCE is described under "COALESCE Expression" on page 575.
- NULLIF is described under "NULLIF Expression" on page 577.

Valued CASE Expression

Purpose

Evaluates a set of expressions for equality with a test expression and returns as its result the value of the scalar expression defined for the first WHEN clause whose value equals that of the test expression. If no equality is found, then CASE returns the scalar value defined by an optional ELSE clause, or if omitted, NULL.

Syntax

where:

Syntax element ...	Specifies ...
value_expression_1	an expression whose value is tested for equality with *value_expression_n*.
value_expression_n	a set of expressions against which the value for *value_expression_1* is tested for equality.
scalar_expression_n	an expression whose value is returned on the first equality comparison of *value_expression_1* and *value_expression_n*.
scalar_expression_m	an expression whose value is returned if evaluation falls through to the ELSE clause.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Teradata Database does not enforce the ANSI restriction that *value_expression_1* must be a deterministic function. In particular, Teradata Database allows the function RANDOM to be used in *value_expression_1*.

Note that if RANDOM is used, nondeterministic behavior may occur, depending on whether *value_expression_1* is recalculated for each comparison to *value_expression_n*.

Usage Notes

WHEN clauses are processed sequentially.

The first WHEN clause *value_expression_n* that equates to *value_expression_1* returns the value of its associated *scalar_expression_n* as its result. The evaluation process then terminates.

If no *value_expression_n* equals *value_expression_1*, then *scalar_expression_m*, the argument of the ELSE clause, is the result.

If no ELSE clause is defined, then the result defaults to NULL.

The data type of *value_expression_1* must be comparable with the data types of all of the *value_expression_n* values.

For information on the result data type of a CASE expression, see "Rules for the CASE Expression Result Type" on page 566.

You can use a scalar subquery in the WHEN clause, THEN clause, and ELSE clause of a CASE expression. If you use a non-scalar subquery (a subquery that returns more than one row), a runtime error is returned.

Recommendation: Do not use the built-in functions CURRENT_DATE or CURRENT_TIMESTAMP in a CASE expression that is specified in a partitioning expression for a partitioned primary index (PPI). In this case, all rows are scanned during reconciliation.

Default Title

The default title for a CASE expression appears as:

```
<CASE expression>
```

Restrictions on the Data Types in a CASE Expression

Restrictions on the Data Types in a CASE Expression

The following restrictions apply to CLOB, BLOB, and UDT types in a CASE expression:

Data Type	Restrictions
BLOB	A BLOB can only appear in *value_expression_1*, *value_expression_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in *value_expression_1*, *value_expression_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to CHAR or VARCHAR.

Data Type	Restrictions
UDT	Multiple UDTs can appear in a CASE expression, with the following restrictions: • The data type of *value_expression_1* through *value_expression_n* must have the same UDT data type if one of them has a UDT data type. • *scalar_expression_n* and *scalar_expression_m* must be the same UDT data type if one them has a UDT data type. Teradata Database does not perform implicit type conversion on UDTs in CASE expressions. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs, and then explicitly invoke the CAST function in the CASE expression. For more information on CREATE CAST, see *SQL Data Definition Language*.

Related Topics

For additional notes on ...	See ...
error conditions	"Error Conditions" on page 565.
the result data type of a CASE expression	"Rules for the CASE Expression Result Type" on page 566.
format of the result of a CASE expression	"Format for a CASE Expression" on page 571.
nulls and CASE expressions	"CASE and Nulls" on page 572.

Example 1

The following example uses a Valued CASE expression to calculate the fraction of cost in the total cost of inventory represented by parts of type '1':

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

Example 2

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

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

Example 3

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

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

Example of Using UDT Data Types in Value Expressions

You use *value_expression_1* through *value_expression_n* to test for equality in a valued CASE expression.

For these examples, the table is defined as follows:

```
create table udtval038_t1(id integer, udt1 testcircleudt, udt2
testrectangleudt) PRIMARY INDEX (id);
```

The following example shows a valued CASE expression, where all value expressions are of the same UDT data type:

```
SELECT CASE udt1
                 WHEN new testcircleudt('1,1,2,yellow,circ')
                 THEN 'Row 1'
                 WHEN new testcircleudt('2,2,4,purple,circ')
                 THEN 'Row 2'
                 WHEN new testcircleudt('3,3,9,green,circ')
                 THEN 'Row 3'
                 ELSE 'Row is NULL'
                 END
FROM t1;
```

*** Query completed. 4 rows found. One column returned.

```
<CASE  expression>
------------------
Row 3
Row 1
Row is NULL
Row 2
```

However, the following example does not complete successfully because *testrectangleudt* does not match the other UDT data types:

```
SELECT CASE udt1
                 WHEN new testcircleudt('1,1,2,yellow,circ')
                 THEN 'Row 1'
                 WHEN new testrectangleudt('2,2,4,4,purple,rect')
                 THEN 'Row 2'
                 WHEN new testcircleudt('3,3,9,green,circ')
                 THEN 'Row 3'
                 ELSE 'Row is NULL'
                 END
FROM t1;
```

Example of Using UDT Data Types in Scalar Expressions

You use *scalar_expression_n* and *scalar_expression_m* as the expressions to return on when the equality comparison on a valued or searched CASE expression evaluates to TRUE, or the value to return on in an ELSE condition.

For these examples, the table is defined as follows:

```
create table udtval038_t1(id integer, udt1 testcircleudt, udt2
testrectangleudt) PRIMARY INDEX (id);
```

Following is an example of a searched CASE Expression where all scalar expressions are of the same UDT data type.

Note: The *search_condition_n* can be a different UDT data type than the *scalar_expression_n*.

```
   SELECT * FROM udtval038_t1
          WHERE udt1 = CASE
          WHEN udt2 <> new testrectangleudt('2,2,4,4,pink,rect')
          THEN new testcircleudt('1,1,2,blue,circ')
          ELSE new testcircleudt('2,2,4,purple,circ')
          END;

*** Query completed. 2 rows found. 3 columns returned.

         id udt1
----------- ----------------------------------------------
          1 1, 1, 2, yellow, circ
          2 2, 2, 4, purple, circ
```

However, the following example does not complete successfully because the scalar expressions are of different data types.

```
   SELECT * FROM udtval038_t1
          WHERE udt1 = CASE
          WHEN udt2 <> new testrectangleudt('2,2,4,4,pink,rect')
          THEN new testcircleudt('1,1,2,blue,circ')
          ELSE new testrectangleudt('2,2,4,4,purple,rect')
          END;
```

Searched CASE Expression

Purpose

Evaluates a search condition and returns one of a WHEN clause-defined set of scalar values when it finds a value that evaluates to TRUE. If no TRUE test is found, then CASE returns the scalar value defined by an ELSE clause, or if omitted, NULL.

Syntax

where:

Syntax element ...	Specifies ...
search_condition_n	a predicate condition to be tested for truth.
scalar_expression_n	a scalar expression whose value is returned when *search_condition_n* is the first search condition that evaluates to TRUE.
scalar_expression_m	a scalar expression whose value is returned when no *search_condition_n* evaluates to TRUE.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Usage Notes

WHEN clauses are processed sequentially.

The first WHEN clause *search_condition_n* that is TRUE returns the value of its associated *scalar_expression_n* as its result. The evaluation process then ends.

If no *search_condition_n* is TRUE, then *scalar_expression_m*, the argument of the ELSE clause, is the result.

If no ELSE clause is defined, then the default value for the result is NULL.

You can use a scalar subquery in the WHEN clause, THEN clause, and ELSE clause of a CASE expression. If you use a non-scalar subquery (a subquery that returns more than one row), a runtime error is returned.

Recommendation: Do not use the built-in functions CURRENT_DATE or CURRENT_TIMESTAMP in a CASE expression that is specified in a partitioning expression for a partitioned primary index (PPI). In this case, all rows are scanned during reconciliation.

Default Title

The default title for a CASE expression appears as:

```
<CASE expression>
```

Rules for WHEN Search Conditions

WHEN search conditions have the following properties:

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

  For example:

  ```
  SELECT CASE
  WHEN t1.x=t2.x THEN t1.y
  ELSE t2.y
  END FROM t1,t2;
  ```

- Cannot contain SELECT statements.

Restrictions on the Data Types in a CASE Expression

The following restrictions apply to CLOB, BLOB, and UDT types in a CASE expression:

Data Type	Restrictions
BLOB	A BLOB can only appear in *value_expression_1*, *value_expression_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in *value_expression_1*, *value_expression_n*, *scalar_expression_m*, or *scalar_expression_n* when it is cast to CHAR or VARCHAR.
UDT	Multiple UDTs can appear in a CASE expression, with the following restrictions: • The data type of *value_expression_1* through *value_expression_n* must have the same UDT data type if one of them has a UDT data type. • *scalar_expression_n* and *scalar_expression_m* must be the same UDT data type if one them has a UDT data type. Teradata Database does not perform implicit type conversion on UDTs in CASE expressions. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs, and then explicitly invoke the CAST function in the CASE expression. For more information on CREATE CAST, see *SQL Data Definition Language*.

Related Topics

For additional notes on ...	See ...
error conditions	"Error Conditions" on page 565.
the result data type of a CASE expression	"Rules for the CASE Expression Result Type" on page 566.
format of the result of a CASE expression	"Format for a CASE Expression" on page 571.
nulls and CASE expressions	"CASE and Nulls" on page 572.

Example 1

The following statement is equivalent to the first example of the valued form of CASE on "Example 1" on page 558:

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

Example 2

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

Note that the following example does not specify an ELSE clause. ELSE clauses are always optional in a CASE expression. If an ELSE clause is not specified and none of the WHEN conditions are TRUE, then a null is returned.

```
SELECT *
FROM t
WHERE x = CASE
           WHEN y=2
           THEN 1
           WHEN (z=3 AND y=5)
           THEN 2
          END;
```

Example 3

The following example uses an ELSE clause.

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

Example 4

The following example shows how using a CASE expression can result in significantly enhanced performance by eliminating multiple passes over the data. Without using CASE, you would have to perform multiple queries for each region and then consolidate the answers to the individual queries in a final report.

```
SELECT SalesMonth, SUM(CASE
                        WHEN Region='NE'
                        THEN Revenue
                        ELSE 0
                       END),
                   SUM(CASE
                        WHEN Region='NW'
                        THEN Revenue
                        ELSE 0
                       END),
                   SUM(CASE
                        WHEN Region LIKE 'N%'
                        THEN Revenue
                        ELSE 0
                       END)
AS NorthernExposure, NorthernExposure/SUM(Revenue),
SUM(Revenue)
FROM Sales
GROUP BY SalesMonth;
```

Example 5

All employees whose salary is less than $40000 are eligible for an across the board pay increase.

IF your salary is less than ...	AND you have greater than this many years of service ...	THEN you receive this percentage salary increase ...
$30000.00	8	15
$35000.00	10	10
$40000.00		5

The following SELECT statement uses a CASE expression to produce a report showing all employees making under $40000, displaying the first 15 characters of the last name, the salary amount (formatted with $and punctuation), the number of years of service based on the current date (in the column named On_The_Job) and which of the four categories they qualify for: '15% Increase', '10% Increase', '05% Increase' or 'Not Qualified'.

```
SELECT CAST(last_name AS CHARACTER(15))
   ,salary_amount (FORMAT '$,$$9,999.99')
   ,(date - hire_date)/365.25 (FORMAT 'Z9.99') AS On_The_Job
   ,CASE
       WHEN salary_amount < 30000 AND On_The_Job > 8
       THEN '15% Increase'
       WHEN salary_amount < 35000 AND On_The_Job > 10
       THEN '10% Increase'
       WHEN salary_amount < 40000 AND On_The_Job > 10
```

```
        THEN '05% Increase'
        ELSE 'Not Qualified'
       END  AS Plan
WHERE salary_amount < 40000
FROM employee
ORDER BY 4;
```

The result of this query appears in the following table:

last_name	salary_amount	On_The_Job	Plan
Trader	$37,850.00	20.61	05% Increase
Charles	$39,500.00	18.44	05% Increase
Johnson	$36,300.00	20.41	05% Increase
Hopkins	$37,900.00	19.99	05% Increase
Morrissey	$38,750.00	18.44	05% Increase
Ryan	$31,200.00	20.41	10% Increase
Machado	$32,300.00	18.03	10% Increase
Short	$34,700.00	17.86	10% Increase
Lombardo	$31,000.00	20.11	10% Increase
Phillips	$24,500.00	19.95	15% Increase
Rabbit	$26,500.00	18.03	15% Increase
Kanieski	$29,250.00	20.11	15% Increase
Hoover	$25,525.00	20.73	15% Increase
Crane	$24,500.00	19.15	15% Increase
Stein	$29,450.00	20.41	15% Increase

Error Conditions

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

Condition or Expression	Example

A condition after the keyword CASE is supplied.	`SELECT CASE a=1`
	`  WHEN 1`
	`  THEN 1`
	`  ELSE 0`
	`  END`
	`FROM t;`

Condition or Expression	Example

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

A non valid WHEN condition is supplied in a searched CASE expression.	`SELECT CASE`
	`  WHEN a`
	`  THEN 1`
	`  ELSE 0`
	`  END`
	`FROM t;`
	
	`SELECT CASE`
	`  WHEN NULL`
	`  THEN 'NULL'`
	`  END`
	`FROM table_1;`

A non-scalar subquery is specified in a WHEN condition of a searched CASE expression.	`SELECT CASE`
	`  WHEN t.a IN`
	`  (SELECT u.a`
	`  FROM u)`
	`  THEN 1`
	`  ELSE 0`
	`  END`
	`FROM t;`

A CASE expression references multiple UDTs that are not identical to each other.	`SELECT CASE t.shape.gettype()`
	`  WHEN 1`
	`  THEN NEW circle('18,18,324')`
	`  WHEN 2`
	`  THEN NEW square('20,20,400')`
	`  END;`

Rules for the CASE Expression Result Type

Because the expressions in CASE THEN/ELSE clauses can be different data types, determining the result type is not always straightforward. You can use the TYPE attribute function with the CASE expression as the argument to find out the result data type. See "TYPE" on page 303.

The following rules apply to the data type of the CASE expression result.

THEN/ELSE Expressions Having the Same Non-Character Data Type

If all of the THEN and ELSE expressions have the same non-character data type, the result of the CASE expression is that type. For example, if all of the THEN and ELSE expressions have an INTEGER type, the result type of the CASE expression is INTEGER.

For information about how the precision and scale of DECIMAL results are calculated, see "Binary Arithmetic Result Data Types" on page 112.

THEN / ELSE Character Type Expressions

The following rules apply to CASE expressions where the data types of all of the THEN/ELSE expressions are character:

- The result of the CASE expression is also a character data type, with the length equal to the maximum length of the different character data types of the THEN/ELSE expressions.
- If the data types of all of the THEN/ELSE expressions are CHARACTER (or CHAR), the result data type is CHARACTER. If one or more expressions are VARCHAR (or LONG VARCHAR), the result data type is VARCHAR.
- The server character set of the result is determined as follows:
  - If the CASE expression contains 1 nonliteral character expression and 1 or more literals, then Teradata Database tries to translate every literal to the character set of the nonliteral. If the translations are successful, then the character set of the nonliteral is used for the result data type. If the translations are not successful, the server character set of the result is Unicode.
  - If the CASE expression contains more than 1 nonliteral character expression and 1 or more literals, then:
    - i If all of the nonliteral expressions have the same character set, then Teradata Database uses this character set as the common data type. Otherwise, if the nonliteral expressions have differing character sets, then Teradata Database uses the Unicode character set as the common data type.
    - ii Teradata Database tries to translate every literal to the character set of the common data type. If the translations are successful, then the result data type has the character set of the common data type. If the translations are not successful, the server character set of the result is Unicode.

Examples of Character Data in a CASE Expression

For the following examples of CHARACTER data behavior, assume the default server character set is KANJI1 and the table definition for the CASE examples is as follow:

```
CREATE TABLE table_1
(
 i        INTEGER,
 column_l CHARACTER(10) CHARACTER SET LATIN,
 column_u CHARACTER(10) CHARACTER SET UNICODE,
 column_j CHARACTER(10) CHARACTER SET KANJISJIS,
 column_g CHARACTER(10) CHARACTER SET GRAPHIC,
 column_k CHARACTER(10) CHARACTER SET KANJI1
);
```

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see “KANJI1 Character Set” in *International Character Set Support*.

Example 1

The server character set of the result of the following query is UNICODE because the CASE expression contains more than 1 nonliteral character expressions with differing character sets.

```
SELECT i, CASE
            WHEN i=2 THEN column_u
            WHEN i=3 THEN column_j
            WHEN i=4 THEN column_g
            WHEN i=5 THEN column_k
            ELSE column_l
          END
FROM table_1
ORDER BY 1;
```

Example 2

The result of the following query is a 5354 failure (Arguments must be of type KANJI1) because one THEN/ELSE expression is a KANJI1 literal, but the server character sets of all the other THEN/ELSE expressions are not KANJI1.

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

Example 3

One THEN/ELSE expression in the following query has a Unicode column. The query is successful and the result data type is UNICODE because the CASE expression contains 1 Unicode column and all other literals can be successfully translated to Unicode.

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

Example 4

One THEN/ELSE expression in the following query has a Latin column. The query is successful and the result data type is Latin because the other literals can be successfully translated to Latin.

```
SELECT i, CASE
            WHEN i=1 THEN 'abc'
            WHEN i=2 THEN column_l
```

```
            ELSE 'def'
          END
FROM table_1
ORDER BY 1;
```

THEN/ELSE Expressions Having Mixed Data Types

The rules for mixed data appear in the following table:

IF the THEN/ELSE clause expressions ...	THEN ...
consist of BYTE and/or VARBYTE data types	if the data types of all of the THEN/ELSE expressions are BYTE, the result data type is BYTE. If one or more expressions are VARBYTE, the result data type is VARBYTE. The result has a length equal to the maximum length of the different byte data types.
contain a DateTime or Interval data type	all of the THEN/ELSE clause expressions must have the same data type.
contain a FLOAT (approximate numeric) and no character strings	the CASE expression returns a FLOAT result. Note: Some inaccuracy is inherent and unavoidable when FLOAT data types are involved.
are composed only of DECIMAL data	the CASE expression returns a DECIMAL result. Note: A DECIMAL arithmetic result can have up to 38 digits. A result larger than 38 digits produces a numeric overflow error. For information about how the precision and scale of DECIMAL results are calculated, see "Binary Arithmetic Result Data Types" on page 112.
are composed only of mixed DECIMAL, BYTEINT, SMALLINT, INTEGER, and BIGINT data	
are a mix of BYTEINT, SMALLINT, INTEGER, and BIGINT data	the resulting type is the largest type of any of the THEN/ELSE clause expressions, where the following list orders the types from largest to smallest: • BIGINT • INTEGER • SMALLINT • BYTEINT
are composed only of numeric and character data	the numeric data is converted to CHARACTER with a length as determined by the format associated with the numeric expression. Then, the rules for the result data type for character, length, and character set are applied. For details, see "THEN/ELSE Character Type Expressions" on page 567. Note: An error is generated if the server character set is GRAPHIC.

Examples of Numeric Data in a CASE Expression

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

```
CREATE TABLE dec22
   (column_l INTEGER
   ,column_2 INTEGER
   ,column_3 DECIMAL(22,2) );
```

Example 1

In the following statement, the CASE expression fails when column_2 contains the value 1 and column_3 contains the value 11223344556677889900.12 because the result is a DECIMAL value that requires more than 38 digits of precision:

```
SELECT SUM (CASE
             WHEN column_2=1
             THEN column_3 * 6.11223344556677880000
             ELSE column_3
            END )
FROM dec22;
```

Example 2

The following query corrects the problem in Example 1 by shortening the scale of the multiplier in the THEN expression:

```
SELECT SUM (CASE
             WHEN column_2=1
             THEN column_3 * 6.1122334455667788
             ELSE column_3
            END )
FROM dec22;
```

Example 3

In the following query, the CASE expression returns a DECIMAL(38,2) result because the THEN and ELSE clauses contain DECIMAL values:

```
SELECT SUM (CASE
             WHEN column_2=1
             THEN column_3 * 6
             ELSE column_3
            END )
FROM dec22;
```

Examples of Character and Numeric Data in a CASE Expression

The following examples illustrate the behavior of queries containing CASE expressions with a THEN/ELSE clause composed of numeric and character data.

Example 1

In the following query, the CASE expression returns a VARCHAR result because the THEN and ELSE clause contains FLOAT and VARCHAR values. The length of the result is 30 since

the default format for FLOAT is a string less than 30 characters, and USER is defined as VARCHAR(30) CHARACTER SET UNICODE. The result is CHARACTER SET UNICODE because USER is UNICODE.

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

Example 2

For this example, assume the following table definition:

```
CREATE table_1
  (i        INTEGER,
   column_l CHARACTER(10) CHARACTER SET LATIN,
   column_u CHARACTER(10) CHARACTER SET UNICODE,
   column_j CHARACTER(10) CHARACTER SET KANJISJIS,
   column_g CHARACTER(10) CHARACTER SET GRAPHIC,
   column_k CHARACTER(10) CHARACTER SET KANJI1);
```

The following query fails because the server character set is GRAPHIC (because the server character set of the first THEN with a character type is GRAPHIC):

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

Format for a CASE Expression

Default Format

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

Consider the following table definition:

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

Assume the default format for the DATE data type is 'YY/MM/DD'.

The following query displays the result of the CASE expression using the 'YY/MM/DD' default DATE format, not the format defined for the *start_date* or *end_date* columns:

```
SELECT i, CASE
           WHEN i=1
           THEN start_date
           WHEN i=2
           THEN end_date
          END
FROM duration
ORDER BY 1;
```

Using Explicit Type Conversion to Change Format

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

Here is an example that uses CAST to change the format of the result of the CASE expression in the previous query:

```
SELECT i, ( CAST ((CASE
             WHEN i=1
             THEN start_date
             WHEN i=2
             THEN end_date
            END) AS DATE FORMAT 'M4BDD,BYYYY'))
FROM duration
ORDER BY 1;
```

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

CASE and Nulls

The ANSI SQL:2011 standard specifies that the CASE expression and its related expressions COALESCE and NULLIF must be capable of returning a null result.

Nulls and CASE Expressions

The rules for null usage in CASE, NULLIF, and COALESCE expressions are as follows.

- If no ELSE clause is specified in a CASE expression and the evaluation falls through all the WHEN clauses, the result is null.
- Nulls and expressions containing nulls are valid as *value_expression_1* in a valued CASE expression.

  The following examples are valid.

  ```
  SELECT CASE NULL
          WHEN 10
          THEN 'TEN'
         END;

  SELECT CASE NULL + 1
  ```

```
         WHEN 10
         THEN 'TEN'
        END;
```

Both of the preceding examples return NULL because no ELSE clause is specified, and the evaluation falls through the WHEN clause because NULL is not equal to any value or to NULL.

- Comparing NULL to any value or to NULL is always FALSE. When testing for NULL, it is best to use a searched CASE expression using IS NULL or IS NOT NULL in the WHEN condition.

  The following example is valid.

  ```
  SELECT CASE
          WHEN column_1 IS NULL
          THEN 'NULL'
         END
  FROM table_1;
  ```

  Often, Teradata Database can detect when an expression that always evaluates to NULL is compared to some other expression or NULL, and gives an error that recommends using IS NULL or IS NOT NULL instead. Note that ANSI SQL does not consider this to be an error; however, Teradata Database reports an error since it is unlikely that comparing NULL in this manner is the intent of the user.

  The following examples are not legal.

  ```
  SELECT CASE column_1
          WHEN NULL
          THEN 'NULL'
         END
  FROM table_1;

  SELECT CASE column_1
          WHEN NULL + 1
          THEN 'NULL'
         END
  FROM table_1;
  SELECT CASE
          WHEN column_1 = NULL
          THEN 'NULL'
         END
  FROM table_1;
  SELECT CASE
          WHEN column_1 = NULL + 1
          THEN 'NULL'
         END
  FROM table_1;
  ```

- Nulls and expressions containing nulls are valid as THEN clause expressions.

  The following example is valid.

  ```
  SELECT CASE
          WHEN column_1 = 10
          THEN NULL
         END
  ```

```
FROM table_1
```

Note that, unlike the previous examples, the NULL in the THEN clause is an SQL keyword and not the value of a character literal.

CASE Shorthands

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

- COALESCE expression (see "COALESCE Expression" on page 575)
- NULLIF expression (see "NULLIF Expression" on page 577)

COALESCE Expression

Purpose

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

1101E227

where:

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

ANSI Compliance

This is ANSI SQL:2011 compliant.

Usage Notes

A *scalar_expression_n* in the argument list may be evaluated twice: once as a search condition and again as a return value for that search condition.

Using a nondeterministic function, such as RANDOM, in a *scalar_expression_n* may have unexpected results, because if the first calculation of *scalar_expression_n* is not NULL, the second calculation of that *scalar_expression_n*, which is returned as the value of the COALESCE expression, might be NULL.

You can use a scalar subquery in a COALESCE expression. However, if you use a non-scalar subquery (a subquery that returns more than one row), a runtime error is returned.

For additional information, such as the rules for evaluation and result data type, see "CASE" on page 555.

Default Title

The default title for a COALESCE expression appears as:

```
<CASE expression>
```

Restrictions on the Data Types in a COALESCE Expression

The following restrictions apply to CLOB, BLOB, and UDT types in a COALESCE expression:

Data Type	Restrictions
BLOB	A BLOB can only appear in the argument list when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in the argument list when it is cast to CHAR or VARCHAR.
UDT	Multiple UDTs can appear in the argument list only when they are identical types because Teradata Database does not perform implicit type conversion on UDTs in a COALESCE expression.

Example 1

The following example returns the home phone number of the named individual (if present), or office phone if HomePhone is null, or MessageService if present and both home and office phone values are null. Returns NULL if all three values are null.

```
SELECT Name, COALESCE (HomePhone, OfficePhone, MessageService)
FROM PhoneDir;
```

Example 2

The following example uses COALESCE with an arithmetic operator.

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

Example 3

The following example uses COALESCE with a comparison operator.

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

NULLIF Expression

Purpose

Returns NULL if its arguments are equal. Otherwise, it returns its first argument, *scalar_expression_1*.

NULLIF is a shorthand expression for the following full CASE expression:

```
CASE
 WHEN scalar_expression_1=scalar_expression_2
 THEN NULL
 ELSE scalar_expression_1
END
```

Syntax

— NULLIF — (— *scalar_expression1, scalar_expression2* —) —

HH01B094

where:

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

ANSI Compliance

This is ANSI SQL:2011 compliant.

Usage Notes

The *scalar_expression_1* argument may be evaluated twice: once as part of the search condition (see the preceding expanded CASE expression) and again as a return value for the ELSE clause.

Using a nondeterministic function, such as RANDOM, may have unexpected results if the first calculation of *scalar_expression_1* is not equal to *scalar_expression_2*, in which case the result of the CASE expression is the value of the second calculation of *scalar_expression_1*, which may be equal to *scalar_expression_2*.

You can use a scalar subquery in a NULLIF expression. However, if you use a non-scalar subquery (a subquery that returns more than one row), a runtime error is returned.

For additional information, such as the rules for evaluation and result data type, see "CASE" on page 555.

Default Title

The default title for a NULLIF expression appears as:

```
<CASE expression>
```

Restrictions on the Data Types in a NULLIF Expression

The following restrictions apply to CLOB, BLOB, and UDT types in a NULLIF expression:

Data Type	Restrictions
BLOB	A BLOB can only appear in the argument list when it is cast to BYTE or VARBYTE.
CLOB	A CLOB can only appear in the argument list when it is cast to CHAR or VARCHAR.
UDT	Multiple UDTs can appear in the argument list only when they are identical types and have an ordering definition.

Examples

The following examples show queries on the following table:

```
CREATE TABLE Membership
   (FullName CHARACTER(39)
   ,Age SMALLINT
   ,Code CHARACTER(4) );
```

Example 1

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

```
SELECT FullName, NULLIF (Age,0) FROM Membership;
```

Example 2

In the following query, blanks indicate no value.

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

Example 3

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

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

CHAPTER 13 Data Type Conversions

This chapter describes the SQL CAST function and the rules for converting data from one type to another, both explicitly and implicitly.

A data type conversion modifies the data definition (data type, data attributes, or both) of an expression and can be either implicit or explicit. Explicit conversions can be made using the CAST function or Teradata conversion syntax.

For details on data types and data attributes, see *SQL Data Types and Literals*.

Forms of Data Type Conversions

Teradata Database supports the following forms of data conversion:

- Implicit

  See "Implicit Type Conversions" on page 579.
- Explicit using the CAST function

  See "CAST in Explicit Data Type Conversions" on page 586.
- Explicit using Teradata conversion syntax

  See "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 589.

Implicit Type Conversions

Teradata Database permits the assignment and comparison of some types without requiring the types to be explicitly converted. Teradata Database also performs implicit type conversions in the following cases:

- On some argument types passed to macros, stored procedures, and SQL functions such as SQRT.
- On the expression that defines a time zone displacement in an AT clause. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

Example 1: Implicit Type Conversion During Assignment

Consider the following tables:

```
CREATE TABLE T1
   (Fname VARCHAR(25)
   ,Fid   INTEGER
   ,Yrs   CHARACTER(2));

CREATE TABLE T2
   (Wname VARCHAR(25)
   ,Wid   INTEGER
   ,Age   SMALLINT);
```

In the following statement, Teradata Database implicitly converts the character string in T1.Yrs to a numeric value:

```
UPDATE T2 SET Age = T1.Yrs + 5;
```

This is not evident in the syntax of the source statement, but becomes evident when the dictionary information for tables T1 and T2 is accessed.

Example 2: Implicit Type Conversion During Comparison

Consider the table T1 in "Example 1: Implicit Type Conversion During Assignment."

In the following statement, Teradata Database implicitly converts both operands of the comparison operation to FLOAT values before performing the comparison:

```
SELECT Fname, Fid
FROM T1
WHERE T1.Yrs < 55;
```

For details on implicit type conversion of operands for comparison operations, see "Implicit Type Conversion of Comparison Operands" on page 502.

Example 3: Implicit Type Conversion in Parameter Passing Operations

Consider the SQRT system function that computes the square root of an argument.

In the following statement, Teradata Database implicitly converts the character argument to a FLOAT type:

```
SELECT SQRT('13147688');
```

Supported Data Types

Teradata Database performs implicit conversion on the following types:

FROM ...	TO ...	For further details, see ...
Byte	Byte Byte types include BYTE, VARBYTE, and BLOB.	"Byte Conversion" on page 592.
	UDT[a]	

FROM ...	TO ...	For further details, see ...
Numeric	Numeric	"Numeric-to-Numeric Conversion" on page 670.
	DATE	"Numeric-to-DATE Conversion" on page 665.
	Character	"Numeric-to-Character Conversion" on page 660.
	UDT[a]	"Numeric-to-UDT Conversion" on page 674.
DATE	Numeric	"DATE-to-Numeric Conversion" on page 637.
	DATE	"DATE-to-DATE Conversion" on page 635.
	Character	"DATE-to-Character Conversion" on page 631.
	UDT[a]	"DATE-to-UDT Conversion" on page 648.
Character	Numeric	"Character-to-Numeric Conversion" on page 608.
	DATE	"Character-to-DATE Conversion" on page 601.
	Character Character types include CHAR, VARCHAR, and CLOB.	"Character-to-Character Conversion" on page 596.
	Period	"Character-to-Period Conversion" on page 614.
	TIME	"Character-to-TIME Conversion" on page 617.
	TIMESTAMP	"Character-to-TIMESTAMP Conversion" on page 623.
	UDT[a]	"Character-to-UDT Conversion" on page 628.
TIME	UDT[a]	"TIME-to-UDT Conversion" on page 721.
TIMESTAMP	UDT[a]	"TIMESTAMP-to-UDT Conversion" on page 756.
Interval	UDT[a]	"INTERVAL-to-UDT Conversion" on page 658.
UDT	Predefined data types that are the target of implicit casts defined for the UDT[b]	• "UDT-to-Character Conversion" on page 761. • "UDT-to-DATE Conversion" on page 765. • "UDT-to-INTERVAL Conversion" on page 768. • "UDT-to-Numeric Conversion" on page 771. • "UDT-to-TIME Conversion" on page 774. • "UDT-to-TIMESTAMP Conversion" on page 777.
	Other UDTs that are the target of implicit casts defined for the UDT[b]	"UDT-to-UDT Conversion" on page 780.

a. The UDT must have an implicit cast that casts the predefined type to a UDT. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

b. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

For details on data types, see *SQL Data Types and Literals*.

Implicit Conversion of DateTime types

Teradata Database performs implicit conversion on DateTime data types in the following cases:

- When passing data using dynamic parameter markers, or the question mark (?) placeholder.
- With INSERT, INSERT...SELECT, and UPDATE statements.
- With MERGE INTO statements.
- When handling default values for the CREATE/ALTER TABLE statements. For details, see "DEFAULT Phrase" in *SQL Data Types and Literals*.
- During stored procedure execution, including the execution of the following statements: DECLARE, SELECT...INTO, and SET. See *SQL Stored Procedures and Embedded SQL*.

Implicit conversion is dependent on client-side support. For information about the client products which support implicit conversion of DateTime types, see the Teradata Tools and Utilities user documentation.

The following conversions are supported:

FROM...	TO...	For further details, see...
DATE	TIMESTAMP	"Implicit DATE-to-TIMESTAMP Conversion" on page 645.
TIME	TIMESTAMP	"Implicit TIME-to-TIMESTAMP Conversion" on page 713.
TIMESTAMP	DATE	"Implicit TIMESTAMP-to-DATE Conversion" on page 730.
TIMESTAMP	TIME	"Implicit TIMESTAMP-to-TIME Conversion" on page 744.
INTERVAL	INTERVAL	"Implicit INTERVAL-to-INTERVAL Conversion" on page 654.

Teradata Database performs implicit conversion on DateTime data types during assignment in the following cases:

FROM...	TO...	For further details, see...
DATE	TIMESTAMP	"Implicit DATE-to-TIMESTAMP Conversion" on page 645.

FROM...	TO...	For further details, see...
TIME	TIMESTAMP	"Implicit TIME-to-TIMESTAMP Conversion" on page 713.
TIMESTAMP	DATE	"Implicit TIMESTAMP-to-DATE Conversion" on page 730.
TIMESTAMP	TIME	"Implicit TIMESTAMP-to-TIME Conversion" on page 744.
Interval[a]	Exact Numeric	"Implicit INTERVAL-to-Numeric Conversion" on page 657.
Exact Numeric	Interval[a]	"Implicit Numeric-to-INTERVAL Conversion" on page 669.

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

Note: There is a general restriction that in Numeric-to-Interval conversions, the INTERVAL type must have only one DateTime field. However, this restriction is not an issue when implicitly converting the expression of an AT clause because the conversion is done with two CAST statements. See "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

For more information, see "ANSI DateTime and Interval Data Type Assignment Rules" on page 818.

Teradata Database performs implicit conversion on DateTime data types in single table predicates and join predicates in the following cases:

FROM...	TO...	For further details, see...
TIMESTAMP	DATE	"Implicit TIMESTAMP-to-DATE Conversion" on page 730.
Interval[a]	Exact Numeric	"Implicit INTERVAL-to-Numeric Conversion" on page 657.
Exact Numeric	Interval[a]	"Implicit Numeric-to-INTERVAL Conversion" on page 669.

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

For more information, see "Implicit Type Conversion of Comparison Operands" on page 502.

The following are not supported:

- Implicit conversion from TIME to TIMESTAMP and from TIMESTAMP to TIME are not supported in comparisons.
- Implicit conversion of DateTime types in set operations.

For details on data types, see *SQL Data Types and Literals*.

Implicit Conversion Rules

Teradata SQL performs implicit type conversions on expressions before any operation is performed.

The implementation of implicit type conversion follows the same rules as the implementation of explicit type conversion using Teradata conversion syntax. For details, see "Teradata Conversion Syntax in Explicit Data Type Conversions" on page 589.

For details on implicit type conversion of operands for comparison operations, see "Implicit Type Conversion of Comparison Operands" on page 502.

Truncation During Conversion

In some cases, implicit conversion can result in truncation of values without an error.

Recommendation: As a best practice, use an explicit CAST instead of relying on implicit conversions when possible.

Example 1

Consider the following table definition:

```
CREATE TABLE Test1 (c1 INT, c2 VARCHAR(1));
```

The following two INSERT statements complete without any errors.

```
INSERT INTO Test1 VALUES (1, '1');
INSERT INTO Test1 VALUES (2, 2);
```

The following query returns two rows.

```
SELECT * FROM Test1;

   c1      c2
-------------
    1       1
    2               <<<< Note that the value inserted in c2 is a blank
```

In the second INSERT statement, the number 2 was implicitly converted to CHAR using Teradata conversion syntax (that is, not using CAST). The process is as follows:

1. Convert the numeric value to a character string using the default or specified FORMAT for the numeric value. Leading and trailing pad characters are not trimmed.
2. Extend to the right with pad characters if required, or truncate from the right if required, to conform to the target length specification.

   If non-pad characters are truncated, no string truncation error is reported.

The conversion right-justifies the number, but takes the first byte of the result which is a single blank character. For more information about numeric to character conversions, see "Numeric-to-Character Conversion" on page 660.

Restrictions

Teradata Database does not perform implicit conversion on input arguments to UDFs, UDMs, or external stored procedures (external routines). Arguments do not necessarily have

to be exact matches to the parameter types, but they must be compatible. For example, you can pass a SMALLINT argument to an external routine that expects an INTEGER argument because SMALLINT and INTEGER are compatible. To pass a DATE type argument to an external routine that expects an INTEGER argument, you must explicitly cast the DATE type to an INTEGER type. For details, see *SQL External Routine Programming*.

Some SQL functions and operators require arguments that are exact matches to the parameter types. For details, refer to the documentation for the specific function or operator.

CAST in Explicit Data Type Conversions

Purpose

Converts an expression of a given data type to a different data type or the same data type with different attributes.

Teradata SQL supports two different syntaxes for CAST functionality, only one of which is ANSI SQL:2011 compliant.

Syntax

```
── CAST ─ ( expression ─ AS ┬─ ansi_sql_data_type ─┬─ ) ──────────────────────────
                            └─ data_definition_list ─┘
                                                                        1101A627
```

where:

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

ANSI Compliance

The form of CAST syntax that specifies *ansi_sql_data_type* is ANSI SQL:2011 compliant.

The form of CAST syntax that specifies *data_definition_list* is a Teradata extension to the ANSI SQL:2011 standard. Note that when *data_definition_list* consists solely of an ANSI data type declaration, then this form of the syntax is also ANSI-compliant.

Usage Notes

The ANSI SQL:2011 compliant form can be used to convert data types in either ANSI-compliant SQL statements or Teradata SQL statements.

The Teradata extended syntax is more general. It allows a type declaration *or* data attributes *or* both. For more information on data types and attributes, see *SQL Data Types and Literals*.

Avoid using the extended form of CAST for any application intended to be ANSI-compliant and portable.

CAST functions identically in both ANSI and Teradata modes.

When converting DateTime data types, you can use the AT clause to specify the time zone used for the CAST. You can specify the source time zone, a specific time zone displacement, or the current session time zone. For more information, see the section on converting the specific data type, for example, TIMESTAMP-to-DATE Conversion.

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 1234).
- Byte (BYTE, VARBYTE, and BLOB) to any data type other than UDT or byte, and data types other than byte or UDT to byte.
- CLOB to any data type other than UDT or character, and data types other than character or UDT to CLOB.

For information on casting to and from geospatial types, see *SQL Geospatial Types*.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Character Truncation Rules

The following rules apply to character strings:

IF the string is cast in this mode ...	THEN it is truncated of ...
ANSI	trailing pad character spaces to achieve the desired length. Truncation of other characters, or part of a multibyte character, returns an error.
Teradata	trailing characters to achieve the desired length. Truncation on Kanji1 character data types containing multibyte characters might result in truncating one byte of the multibyte character.

Server Character Set Rules

When *data_definition_list* specifies a data type of CHARACTER (CHAR) or CHARACTER VARYING (VARCHAR) and does not specify a CHARACTER SET clause to indicate which server character set to use, then the resulting server character set is as follows:

IF the data type of *expression* is ...	THEN the server character set of the resulting characters is ...
non-character	the user default server character set.
character	the server character set of *expression*.

Numeric Overflow, Field Mode, and CAST

Numeric overflows are handled differently depending on whether you are running ANSI or Teradata mode, and whether you are running in Field Mode or not.

Field Mode is not ANSI SQL:2011 compatible. In Field Mode, conversion to a numeric or decimal data type that results in a numeric overflow is returned as asterisks ('***') rather than an error message.

Record and Indicator Modes do not behave in this manner and return an error message.

Related Topics

For further rules that apply to the conversion between specific data types, for example, numeric-to numeric or character-to-numeric, see the appropriate succeeding topic in this chapter.

Examples

The following examples illustrate how to perform data type conversions using CAST.

Example 1

Using ANSI CAST syntax:

```
SELECT ID_Col, Name_Col
FROM T1
WHERE Int_Col = CAST(SUBSTRING(Char_Col FROM 3 FOR 3) AS INTEGER);
```

Example 2

Using ANSI CAST syntax:

```
SELECT CAST(SUBSTRING(Char_Col FROM 1 FOR 2) AS INTEGER),
   CAST(SUBSTRING (Char_Col FROM 3 FOR 3) AS INTEGER)
FROM T1;
```

Example 3

Using Teradata extensions to the ANSI CAST syntax:

```
CREATE TABLE t2 (f1 TIME(0) FORMAT 'HHhMIm');

INSERT t2 (CAST('15h33m' AS TIME(0) FORMAT 'HHhMIm'));

SELECT f1 FROM t2;
```

The result from the SELECT statement is:

```
    f1
------
15h33m
```

Teradata Conversion Syntax in Explicit Data Type Conversions

Teradata conversion syntax is defined as follows:

Syntax

where:

Syntax element ...	Specifies ...
expression	the data expression to be converted to the new definition specified by *data_type* and *data_attributes*.
data_type	a data type declaration such as INTEGER or DATE.
data_attribute	a data attribute such as FORMAT or TITLE.

ANSI Compliance

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

Using CAST Instead of Teradata Conversion Syntax

Using Teradata conversion syntax is strongly discouraged. It is an extension to the ANSI SQL:2011 standard and is retained only for backward compatibility with existing applications. Instead, use CAST to explicitly convert data types.

Usage Notes

When the conversion specifies *data_type*, then the data is converted at run time. At that time, a data conversion or range check error may occur.

For any kind of data type conversion using Teradata conversion syntax, where the item that includes a data type declaration is an operand of a complex expression, you must either enclose the appropriate entities in parentheses or use the CAST syntax.

You should always use the CAST function to perform conversions in new applications to ensure ANSI compatibility.

Related Topics

For further rules that apply to the conversion between specific data types, for example, numeric-to numeric or character-to-numeric, see the appropriate succeeding topic in this chapter.

Example 1

To evaluate an expression of the following form correctly:

```
column_name (INTEGER) + variable
```

You could enter the expression as follows:

```
(column_name (INTEGER)) + variable
```

or, preferably, as:

```
CAST (column_name AS INTEGER) + variable
```

For more information on using CAST, see "CAST in Explicit Data Type Conversions" on page 586.

Example 2

Here is an example that uses the Teradata conversion syntax, and specifies the FORMAT data attribute to convert the format of a DATE data type.

```
CREATE TABLE date1 (d1 DATE FORMAT 'E4,BM4BDD,BY4');
CREATE TABLE char1 (c1 CHAR(10));

INSERT date1 ('Saturday, March 16, 2002');

INSERT INTO char1 (c1)
SELECT ((d1 (FORMAT 'YYYY/MM/DD')))
FROM date1;

SELECT * FROM char1;
```

The result from the SELECT statement is:

```
         c1
----------
2002/03/16
```

If the second INSERT statement did not convert the DATE format to 'YYYY/MM/DD', the result from the SELECT statement is:

```
         c1
----------
Saturday,
```

Data Conversions in Field Mode

Field Mode: User Response Data

In Field Mode, a report format used in BTEQ, all data is returned in character form. The alignment and spacing of columns is controlled by data formats and title information. Each row returned is essentially a character string ready for display.

In Field Mode, it is unnecessary to explicitly convert numeric data to character format.

Conversions to Numeric Types

When in Field Mode, a numeric overflow returned for character to numeric data type conversion is not treated as an error. If the result exceeds the number of digits normally reserved for the numeric data type, the result appears as a set of asterisks in the report.

For example, the character to SMALLINT conversion in the following statement results in numeric overflow because the number of digits normally reserved for a SMALLINT is five:

```
SELECT '100000' (SMALLINT);
```

The result is:

```
'100000'
--------
  ******
```

Additionally, when in Field Mode, asterisks appear in the report for conversions to numeric types involving results that do not fit the specified output format.

For example, the DATE to INTEGER conversion in the following statement results in a value that does not fit the format specified by the FORMAT phrase:

```
SELECT CAST (CURRENT_DATE as integer format '9999');
```

The result is:

```
Date
----
****
```

The same query executed in Record or Indicator Variable Mode reports an error.

Byte Conversion

Purpose

Converts a byte expression to a different data definition.

CAST Syntax

1101B335

where:

Syntax element ...	Specifies ...
byte_expression	an expression in byte format to be cast to a different data definition.
byte_data_type	the new byte type to which *byte_expression* is to be converted.
UDT_data_type	a UDT that has a cast definition that casts the byte type to the UDT. To define a cast for a UDT, use the CREATE CAST statement. For details on CREATE CAST, see *SQL Data Definition Language*.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

CAST is ANSI SQL:2011 compliant, provided the syntax does not specify data attributes.

Teradata Conversion Syntax

1101A623

where:

Syntax element ...	Specifies ...
byte_expression	an expression in byte format to be cast to a different byte data definition.
byte_data_type	an optional byte type to which *byte_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

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

Conversions Where Source and Target Types Differ in Length

If the length specified by *byte_data_type* is less than the length of *byte_expression*, bytes beyond the specified length are truncated. No error is reported.

If *byte_data_type* is fixed-length and the length is greater than that of *byte_expression*, bytes of value binary zero are appended as required.

Supported Source and Target Data Types

Teradata Database supports byte data type conversions according to the following table.

Source Data Type	Target Data Type	Allowable Conversions
BYTE	• BYTE • VARBYTE • BLOB	• Implicit • Explicit using CAST and Teradata conversion syntax
VARBYTE		
BLOB		
BYTE	UDT[a]	• Implicit • Explicit using CAST
VARBYTE		
BLOB		
UDT[a]	• BYTE • VARBYTE • BLOB	• Implicit • Explicit using CAST and Teradata conversion syntax

a. Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Rules for Implicit Byte-to-UDT Conversions

Teradata Database performs implicit Byte-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit Byte-to-UDT data type conversion requires a cast definition (see "Usage Notes") that specifies the following:

- the AS ASSIGNMENT clause
- a BYTE, VARBYTE, or BLOB source data type

  The source data type of the cast definition does not have to be an exact match to the source of the implicit type conversion.

If multiple implicit cast definitions exist for converting different byte types to the UDT, Teradata Database uses the implicit cast definition for the byte type with the highest precedence. The following list shows the precedence of byte types in order from lowest to highest precedence:

- BYTE
- VARBYTE
- BLOB

Using HASHBUCKET to Convert a BYTE Type to an INTEGER Type

You can use the HASHBUCKET function to convert a BYTE(1) or BYTE(2) type to an INTEGER type. For details, see "Using HASHBUCKET to Convert a BYTE Type to an INTEGER Type" on page 891.

Example 1: Explicit Conversion of BLOB to VARBYTE

Consider the following table definition:

```
CREATE TABLE large_images
  (id INTEGER
  ,image BLOB);
```

The following statement casts the BLOB column to a VARBYTE type, and uses the result as an argument to the POSITION function:

```
SELECT POSITION('FFF1'xb IN (CAST(image AS VARBYTE(64000))))
FROM large_images
WHERE id = 5;
```

Example 2: Implicit Conversion of VARBYTE to BLOB

Consider the following table definitions:

```
CREATE TABLE small_images
  (id INTEGER
  ,image1 VARBYTE(30000)
  ,image2 VARBYTE(30000));

CREATE TABLE large_images
  (id INTEGER
  ,image BLOB);
```

Teradata Database performs a VARBYTE to BLOB implicit conversion for the following INSERT statement:

```
INSERT large_images
SELECT id, image1 || image2
FROM small_images;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-Character Conversion

Purpose

Shortens or expands output character strings.

CAST Syntax

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a different character data definition.
character_data_type	the new data type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE • CHARACTER SET

ANSI Compliance

CAST is ANSI SQL:2011 compliant, provided the syntax does not specify any data attributes.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a different character data definition.
character_data_type	an optional character type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE • CHARACTER SET If the syntax specifies *character_data_type*, CHARACTER SET can only appear after *character_data_type*.

ANSI Compliance

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

Implicit Character-to-Character Conversion

CLOB types can only be converted to or from CHAR or VARCHAR types. For example, implicit conversion is performed on CLOB data that is inserted into a CHAR or VARCHAR column.

Comparisons of strings (both fixed- and variable-length) require operands of equal length. The following table shows that the shorter string is converted by being padded on the right.

THIS expression ...	IS converted to ...	AND the result is ...
'x'='x '	'xΔ'='x '	TRUE
'x'='xx'	'xΔ'='xx'	FALSE

where Δ is a pad character.

If a character is not in the repertoire of the target character set, an error is reported.

For rules on the effect of server character sets on character conversion, see "Implicit Character-to-Character Translation" on page 599.

CAST Syntax Usage Notes

The server character set of *character_expression* must have the same server character set as the target data type.

If CAST is used to convert data to a character string and non-pad characters would be truncated, an error is reported.

Teradata Conversion Syntax Usage Notes

The server character set of *character_expression* can be changed to a different server character set specified as *data_attribute*, where *data_attribute* is the CHARACTER SET phrase.

This is not the recommended way to perform this translation. Instead, use the TRANSLATE function. For information, see "TRANSLATE" on page 1234.

General Usage Notes

If the source string (CHAR, VARCHAR, or CLOB) is longer than the target data type (CHAR, VARCHAR, or CLOB), excess characters are truncated.

IF the session doing an INSERT or UPDATE is in this mode ...	AND non-pad characters would be truncated to store character values in a table, THEN ...
ANSI	an error is reported.
Teradata	no error is reported.

Pad characters are trimmed or appended, according to the following rules:

IF the source string data type is ...	AND it is ...	AND the target data type is ...	THEN ...
CHAR	longer than the target	CLOB or VARCHAR	any trailing pad characters are trimmed.
CHAR, VARCHAR, or CLOB	shorter than the target	CHAR	trailing pad characters are appended to the target.
CHAR	all pad characters	CLOB or VARCHAR	the field is truncated to zero length.

Examples

Following are examples of character to character conversions:

Character String	String Length	Character Description	Conversion Result	Converted Length
'HELLO'	5	CHAR(3)	'HEL', if session is in Teradata mode	3
			Error, if session is in ANSI mode	
'HELLO'	5	CHAR(7)	'HELLO '	7
'HELLO'	5	VARCHAR(7)	'HELLO'	5
'HELLO '	7	VARCHAR(6)	'HELLO '	6

Character String	String Length	Character Description	Conversion Result	Converted Length
'HELLO '	7	VARCHAR(3)	'HEL', if session is in Teradata mode	3
			Error, if session is in ANSI mode	

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Implicit Character-to-Character Translation

Implicit string translation occurs when two character strings are incompatible within a given operation. For example,

```
SELECT *
FROM string_table
WHERE clatin < csjis;
```

where clatin represents a character column defined as CHARACTER SET LATIN and csjis represents a character column defined as CHARACTER SET KANJISJIS.

If an implicit translation of character string '*string*' to a UNICODE character string is required, it is equivalent to executing the TRANSLATE*(string* USING *source_repertoire_name*_TO_Unicode*)* function, where *source-repertoire-name* is the server character set of *string*.

More specifically, if as in the above example, *string* is of KANJISJIS type, then the translation is equivalent to executing the TRANSLATE*(string* USING KanjiSJIS_TO_Unicode) function.

ANSI Compliance

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

Character Literals

The following rules apply to implicit character-to-character translation involving character literals.

IF one operand is a ...	AND the other operand is a ...	THEN ...
literal	literal	both operands are translated to UNICODE.
	non-literal	the literal is translated to the type of the non-literal. If that fails, both are translated to UNICODE.
	literal expression	the literal is translated to the type of the literal expression. If that fails, both are translated to UNICODE.

IF one operand is a ...	AND the other operand is a ...	THEN ...
literal expression	literal expression	both operands are translated to UNICODE.
	non-literal	the literal expression is translated to the type of the non-literal. If that fails, both are translated to UNICODE.
non-literal	non-literal	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-literal> is first translated to Latin, and then the trim operation is performed.

```
...
TRIM(<unicode-literal> FROM <latin-value>)
```

The result is Latin.

Character-to-DATE Conversion

Purpose

Converts a character string to a date value.

CAST Syntax

```
─CAST─( character_expression ─AS─ DATE ──┬──────────────────────┬──)──────────────
                                         │ ┌──────────────────┐ │
                                         └─┴─ data_attribute ─┴─┘
```

1101B244

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a DATE value.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the date data.

Teradata Conversion Syntax

```
─ character_expression ─(──┬──────────────────────┬── DATE ──┬──────────────────────┬──)──────────────
                           │ ┌──────────────────┐ │          │ ┌──────────────────┐ │
                           └─┴─ data_attribute ,─┴─┘          └─┴─ , data_attribute ─┴─┘
```

1101B255

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a DATE value.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

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

Implicit Character-to-DATE Conversion

If the string does not represent a valid date, an error is reported.

In record or indicator mode, when the DateForm mode of the session is set to ANSIDate, the string must use the ANSI DATE format.

Usage Notes

The character expression is trimmed of leading and trailing pad characters and handled as if it was a string literal in the declaration of a DATE literal.

Character-to-DATE conversion is supported for CHAR and VARCHAR types only. The source character type cannot be CLOB.

If the string can be converted to a valid DATE, then it is. Otherwise, an error is returned.

Character String Format

If the dateform of the current session is INTEGERDATE, the date representation in the character string must match the DATE output format according to the rules in the following table:

IF the statement ...	THEN ...
specifies a FORMAT phrase for the DATE	the character string must match that DATE format.
does not specify a FORMAT phrase	If the DATE column definition: Specifies a FORMAT phrase, the character string must match the DATE format. Does not specify a FORMAT phrase, the character string must match 'YY/MM/DD', or the current setting of the default date format in the specification for data formatting (SDF) file.

For an example, see "Example 1: IntegerDate Dateform Mode" on page 604.

If the dateform of the current session is ANSIDATE, the date representation in the character string must match the DATE output format according to the rules in the following table:

IF the statement ...	THEN ...
specifies a FORMAT phrase for the DATE	the character string must match that DATE format.
does not specify a FORMAT phrase	*If in field mode*, then if the DATE column definition: • Specifies a FORMAT phrase, the character string must match that DATA format. • Does not specify a FORMAT phrase, the character string must match the ANSI format ('YYYY-MM-DD') *If in record or indicator mode*, the character string must match the ANSI format ('YYYY-MM-DD').

For an example, see "Example 2: ANSIDate Dateform Mode" on page 605.

Forcing a FORMAT on CAST for Converting Character to DATE

You can use a FORMAT phrase to convert a character string that does not match the format of the target DATE data type. A character string in a conversion that does not specify a FORMAT phrase uses the output format for the DATE data type.

For example, suppose the session dateform is INTEGERDATE and the default DATE format of the system is set to 'yyyymmdd' through the tdlocaledef utility. The following statement fails, because the character string contains separators, which does not match the default DATE format:

```
SELECT CAST ('2005-01-01' AS DATE);
```

To override the default DATE format, and convert a character string that contains separators, specify a FORMAT phrase for the DATE target type:

```
SELECT CAST ('2005-01-01' AS DATE FORMAT 'YYYY-MM-DD');
```

In character-to-DATE conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- EEEE
- E4
- EEE
- E3

For more information on default formats and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Character Strings That Omit Day, Month, or Year

If the character string and the format for a character-to-DATE conversion omits the day, month, or year, the system uses default values for the target DATE value.

IF the character string omits the ...	THEN the system uses the ...
day	value of 1 (the first day of the month).
month	value of 1 (the month of January).
year	current year (at the current session time zone).

Consider the following table:

```
CREATE TABLE date_log
   (id INTEGER
   ,start_date DATE
   ,end_date DATE
   ,log_date DATE);
```

The following INSERT statement converts three character strings to DATE values. The first character string omits the day, the second character string omits the month, and the third character string omits the year. Assume the current year is 1992.

```
INSERT date_log
   (1001
   ,CAST ('January 1992' AS DATE FORMAT 'MMMMBYYYY')
   ,CAST ('1992-01' AS DATE FORMAT 'YYYY-DD')
   ,CAST ('01/01' AS DATE FORMAT 'MM/DD'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM date_log;

         id  start_date  end_date  log_date
-----------  ----------  --------  --------
       1001    92/01/01  92/01/01  92/01/01
```

Example 1: IntegerDate Dateform Mode

For example, suppose the session dateform is INTEGERDATE, and the default DATE format of the system is set to 'yyyymmdd' through the tdlocaledef utility.

Consider the following table, where the start_date column uses the default DATE format and the end_date column uses the format 'YYYY/MM/DD':

```
CREATE TABLE date_log
   (id INTEGER
   ,start_date DATE
   ,end_date DATE FORMAT 'YYYY/MM/DD');
```

The following INSERT statement works because the character strings match the formats of the corresponding DATE columns and Teradata Database can successfully perform implicit character-to-DATE conversion:

```
INSERT INTO date_log (1099, '20030122', '2003/01/23');
```

To perform character-to-DATE conversion on character strings that do not match the formats of the corresponding DATE columns, you must use a FORMAT phrase:

```
INSERT INTO date_log
   (1047
   ,CAST ('Jan 12, 2003' AS DATE FORMAT 'MMMBDD,BYYYY')
   ,CAST ('Jan 13, 2003' AS DATE FORMAT 'MMMBDD,BYYYY'));
```

Example 2: ANSIDate Dateform Mode

Suppose the session dateform is ANSIDATE. The default DATE format of the system is 'YYYY-MM-DD'.

Consider the following table, where the start_date column uses the default DATE format and the end_date column uses the format 'YYYY/MM/DD':

```
CREATE TABLE date_log
   (id INTEGER
   ,start_date DATE
   ,end_date DATE FORMAT 'YYYY/MM/DD');
```

The following INSERT statement works because the character strings match the formats of the corresponding DATE columns and Teradata Database can successfully perform implicit character-to-DATE conversion:

```
INSERT INTO date_log (1099, '2003-01-22', '2003/01/23');
```

To perform character-to-DATE conversion on character strings that do not match the formats of the corresponding DATE columns, you must use a FORMAT phrase:

```
INSERT INTO date_log
   (1047
   ,CAST ('Jan 12, 2003' AS DATE FORMAT 'MMMBDD,BYYYY')
   ,CAST ('Jan 13, 2003' AS DATE FORMAT 'MMMBDD,BYYYY'));
```

Example 3: Implicit Character-to-DATE Conversion

Assume that the DateForm mode of the session is set to ANSIDate.

The following CREATE TABLE statement specifies a FORMAT phrase for the DATE data type column:

```
CREATE SET TABLE datetab (f1 DATE FORMAT 'MMM-DD-YYYY');
```

In field mode, the following INSERT statement successfully performs the character to DATE implicit conversion because the format of the string conforms to the format of the DATE column in the datetab table:

```
INSERT INTO datetab ('JAN-10-1999');
```

In record or indicator mode, when the DateForm mode of the session is set to ANSIDate, the following INSERT statement successfully performs the character to DATE implicit conversion because the format of the string is in the ANSI DATE format:

```
INSERT INTO datetab ('2002-05-10');
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-INTERVAL Conversion

Purpose

Converts a character string to an interval value.

CAST Syntax

```
─CAST─( character_expression ─AS─ interval_data_type ┌──────────────────┐ )──────────
                                                     │┌────────────────┐│
                                                     └┴─ data_attribute ┘┘
                                                                          1101B245
```

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to an INTERVAL value.
interval_data_type	an INTERVAL data type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI SQL, Teradata supports the specification of data attributes.

Teradata Conversion Syntax

```
─character_expression ─( ┌──────────────────┐ interval_data_type ┌──────────────────┐ )──────────
                         │┌────────────────┐│                   │┌────────────────┐│
                         └┴─ data_attribute ,┘                   └┴─ , data_attribute ┘┘
                                                                                        1101B256
```

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to an INTERVAL value.
data_attribute	one of the following optional data attributes: • NAMED • TITLE

Syntax element ...	Specifies ...
interval_data_type	an INTERVAL data type to which *character_expression* is to be converted.

ANSI Compliance

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

Usage Notes

The character value is trimmed of leading and trailing pad characters and handled as if it was a string literal in the declaration of an INTERVAL string literal.

Character-to-INTERVAL conversion is supported for CHAR and VARCHAR types only. The source character type cannot be CLOB.

If the contents of the character string can be converted to a valid INTERVAL, then they are; otherwise, an error is returned.

You cannot convert a character data type of GRAPHIC to an INTERVAL string literal.

Example 1

The following query returns ' -265-11'.

```
SELECT CAST('-265-11' AS INTERVAL YEAR(4) TO MONTH);
```

Example 2

If the source character string contains values not normalized in the INTERVAL form, but which nevertheless can be converted to a proper INTERVAL, the conversion is made.

For example, the following query returns '-267-06'

```
SELECT CAST('265-30' AS INTERVAL YEAR(4) TO MONTH);
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-Numeric Conversion

Purpose

Converts a character data string to a numeric value.

CAST Syntax

```
— CAST — (character_expression — AS — numeric_data_type ┬──────────────┬─ ) ──────────
                                                        └─ data_attribute ─┘
                                                                                 1101A628
```

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a numeric type.
numeric_data_definition	the numeric type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the numeric data.

Teradata Conversion Syntax

```
── character_expression ── ( ┬──────────────┬ numeric_data_type ┬──────────────┬ ) ──
                             └─ data_attribute , ─┘            └─ , data_attribute ─┘
                                                                                 1101A629
```

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a numeric type.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
numeric_data_type	the numeric type to which *character_expression* is to be converted.

ANSI Compliance

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

Implicit Character-to-Numeric Conversion

Implicit character to numeric conversion produces a valid result only if the character string represents a numeric value.

If a CHAR or VARCHAR character string is present in an expression that requires a numeric operand, it is read as a formatted numeric and is converted to a FLOAT value, using the default format for FLOAT.

To override the implicit format, use a FORMAT phrase.

Or, to change the default format for FLOAT, you can change the setting of the *REAL* element in the specification for data formatting (SDF) file. For information on default data type formats, the SDF file, and the FORMAT phrase, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

To use a CLOB type in an expression that requires a numeric operand, you must first explicitly convert the CLOB to CHAR or VARCHAR.

An empty character string (zero length) or a character string consisting only of pad characters is interpreted as having a numeric value of zero.

If the default format for FLOAT is -9.99E-99, then:

THIS expression ...	IS converted to ...	AND the result is ...
1.1*’$20.00’	1.10E 00*2.00E1	2.20E 01
’2’+’2’	2.00E 00+2.00E 00	4.00E 00
’A’ + 2	----------	error

If a column or parameter of numeric data type is specified with a string value, the string is again assumed to be a formatted numeric. For example, the following INSERT statement specifies the Salary as a numeric string:

```
INSERT INTO Employee (EmpNo, Name, Salary)
VALUES (10022, 'Clements D', '$38,000.00');
```

The conversion to numeric type removes editing symbols. When selected, the salary data contains only the special characters allowed by the FORMAT phrase for Salary in the CREATE TABLE statement. If the FORMAT phrase is 'G-(9)D9(2)', then the output looks like this:

```
   Salary
---------
38,000.00
```

If the FORMAT phrase is 'G-L(9)D9(2)', then the output looks like this:

```
    Salary
----------
$38,000.00
```

Supported Character Types

The character expression to be converted must be CHAR or VARCHAR. CLOBs cannot be explicitly converted to numeric types.

Usage Notes

Before processing begins, the numeric description is scanned for a FORMAT phrase, which is used to determine the radix separator, group separator, currency sign or string, signzone (S), or implied decimal point (V) formatting.

Conversion is performed positionally, character by character, from left to right, until the end of the number.

Only all-numeric character strings can be converted from character to numeric formats. For example, you can convert the character strings 'US Dollars 123456' or '123456' to the integer value 123456, but you cannot convert the string 'EX1AM2PL3E' to a numeric value.

The following list shows the steps for converting character type data to numeric. Note that you cannot convert a *character_expression* of GRAPHIC character type to numeric.

Conversion is performed stage by stage, without returning to a previous stage; however, stages can be skipped.

1. Leading pad characters are ignored. Trailing pad characters are ignored, except for signed zoned decimal input.

   Embedded spaces are only allowed according to the following rules:

   - If the current SDF file defines the group separator as a space, then the character string can include spaces to separate groups of digits to the left of the radix separator, according to the grouping rule defined by *GroupingRule* or *CurrencyGroupingRule*.
   - If the current SDF file defines the radix separator as a space, then the character string can include one space as the radix character.
   - If the FORMAT phrase contains a currency formatting character, such as N, and the matching currency string in the SDF file, such as *CurrencyName*, contains a space, the character string can include spaces as part of that currency string.

2. The sign (+ or -) is saved as part of the number. A mantissa sign may appear before the first digit in the string, or after the last digit in the string. An exponent sign may appear with a preceding mantissa sign.
3. The currency sign is ignored if it matches the FORMAT. A currency string is ignored if it matches the FORMAT. Only one currency is allowed in the string.
4. Digits are saved as the integral, fractional, or exponent part of the number, depending on whether the radix or the letter E has been parsed.
5. Separators are ignored, unless they match the radix specified in the FORMAT.

   If a separator matches the radix specified in the FORMAT, the location is saved as the beginning of the fractional part of the number. V marks the fractional component for implied decimals.

   The allowance of currency and separators is a non-ANSI Teradata extension of character to numeric conversion.
6. Embedded dashes (between digits) are allowed, unless the number is signed or includes a radix, currency, or exponent.
7. The letter E is saved as the beginning of the exponent part of the number. One space is allowed following an E.
8. The exponent sign (+ or -) is saved.
9. The exponent digits are saved. A sign character cannot appear after any exponent digit.

Numeric Overflow

In Field Mode, numeric overflow in character to numeric conversion is not treated as an error. If the result exceeds the number of digits normally reserved for the data type, asterisks are displayed.

In Record and Indicator Variable Modes, numeric overflow is reported as an error. This behavior applies to both the CAST and Teradata conversion syntax.

FORMAT Phrase Controls Parsing of the Data

A FORMAT phrase, by itself, cannot convert a character type value to a numeric type value. The phrase controls partially how the resultant value is parsed.

Some examples of character to numeric conversion appear in the following table. For FORMAT phrases that contain G, D, C, and N formatting characters, assume that the related entries in the specification for data formatting file (SDF) are:

```
RadixSeparator {"."}
GroupSeparator {","}
GroupingRule   {"3"}
Currency       {"$"}
ISOCurrency    {"USD"}
CurrencyName   {"US Dollars"}
```

Character String	Converted To	Resultant Numeric Value	Field Mode Display Result
'$20,000.00'	DECIMAL(10,2)	20000.00	20000.00
'$$$.50'	DECIMAL(10,2)	error[a]	error
'$.50'	DECIMAL(8,2)	.50	.50
'.345'	DECIMAL(8,3)	.345	.345
'-1.234E-02'	FLOAT	-.01234	-.01234
'-1E.-2'	FLOAT	error[b]	error
'00000000-.93'	DECIMAL(12,4)	error[c]	error
'- 55'	INTEGER	-55	-55
'E67'	FLOAT	0.0	0.00000000000000E 000
'9876'	DECIMAL(4,2) FORMAT '99V99'	98.76	9876
'-123'	INTEGER	-123	-123
'9876'	DECIMAL(4,2) FORMAT '9(2)V9(2)'	98.76	9876
'1-2-3'	INTEGER	123	123
'123-'	INTEGER	-123	-123
'123- '	INTEGER	-123	-123
'-1.234E 02'	FLOAT	-123.4	-1.23400000000000E 002
'111,222,333'	INTEGER FORMAT 'G9(I)'	111222333	0,111,222,333
'2.49US Dollars'	DECIMAL(10,2) FORMAT 'GZ(I)D9(F)BN'	2.49	2.49 US Dollars
'25000USD'	INTEGER FORMAT '9(I)C'	25000	0000025000USD

a. Only one currency is allowed in the character string.

b. The radix must precede the exponent part of the number.

c. Embedded dashes cannot appear in a string containing a radix.

A conversion that does not specify a FORMAT phrase uses the corresponding data type default format as defined in the SDF.

For more information on default data type formats, the SDF file, and the meaning of formatting characters in a FORMAT phrase, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Example: Implicit Conversion of Character to Numeric

The INSERT statement in the following example implicitly converts the character data type to the target numeric data type:

```
CREATE TABLE t1
(f1 DECIMAL(10,2) FORMAT 'G-U(9)D9(2)');

INSERT t1 ('USD12,345,678.90');
```

If a column definition in a CREATE TABLE statement does not specify a FORMAT phrase for the data type, the column uses the corresponding data type default format as defined in the specification for data formatting (SDF) file. For more information on the default format of data types and the definition of formatting characters in a FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals.*

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-Period Conversion

Purpose

Converts a character string to a Period value.

CAST Syntax

```
── CAST ── ( character_expression ── AS ── period_data_type ┬──────────────────┬ ) ──────
                                                           │┌─ data_attribute ─┐│
                                                           └┘                  └┘
                                                                                1101A587
```

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a Period value.
period_data_type	Period data type to which *character_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

Usage Notes

A character value expression can be cast as PERIOD(DATE), PERIOD(TIME), or PERIOD(TIMESTAMP) using the CAST function or implicit casting. A character input value can also be implicitly cast as a Period type.

After any leading and trailing pad characters in the source character value are trimmed, the resulting character string must conform to the format of the target type. Conversion of the beginning and ending portions of the character value expression to corresponding DateTime values follow the existing rules of CHARACTER/VARCHAR to DateTime data type conversions.

The existing rules include conversion of the source value with a TIME or TIMESTAMP format to UTC based on the specified time zone in the source or, if not specified, the current session time zone. The exception to conversion to UTC for Period data types is when the ending

portion of the source character is a TIMESTAMP value without a time zone and the value is equal to the maximum value that is used to represent UNTIL_CHANGED; in this case, the value is not changed to UTC.

If the target type has a TIME or TIMESTAMP element type and the beginning or ending bound portions of the character value expression contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.

If target type has a TIME or TIMESTAMP element type and the target precision is lower than either precision specified in the source character string, an error is reported. If the target precision is higher than a precision specified for a bound in the source character string, trailing zeros are added to the fractional seconds of the corresponding bound of the Period value resulting from the cast.

The target elements are set to the corresponding resulting values.

If the result beginning bound is not less than the result ending bound in their UTC forms, an error is reported.

If an ANSI DateTime format is used to interpret the character data during conversion, then enclosing the beginning and ending values inside apostrophes is optional. For details, see "Character Strings that Use ANSI DateTime Format" on page 616.

Implicit Character-to-Period Conversion

A CHARACTER or VARCHAR value is implicitly cast as a Period data type for an assignment, update, insert, merge, or parameter passing operation when the target site has a Period data type and for a comparison operation if the other operand has a Period data type. If any other non-Period value is directly assigned to a Period target site, an error is reported. In the same manner, if any other non-Period value is directly compared to a Period value, an error is reported.

Note: In some cases, a value may be explicitly cast as a Period data type in order to avoid this error.

During implicit conversion from CHARACTER or VARCHAR to Period data type, the ANSI DateTime format string is used to interpret the beginning and ending element values in the character string, if the response mode is other than the Field mode or if the character string data is parameterized. If the response mode is Field mode and if the character string data is not parameterized, then the target period format is used to interpret the beginning and ending element values in the character string. The following table describes this in detail.

Mode	Parameterized Data Present	Format for Implicit Cast Interpretation
Field	No	Target format
Field	Yes	ANSI format
Non-field	Yes	ANSI format
Non-field	No	ANSI format

When the ANSI DateTime format string is used to interpret the beginning and ending element values in the character string, enclosing the beginning and ending values inside the apostrophes is optional. This relaxation applies even during an explicit cast. For details, see "Character Strings that Use ANSI DateTime Format" on page 616.

Character Strings that Use ANSI DateTime Format

Here is a list of valid character string representations when the implicit or explicit character-to-period conversion uses the ANSI DateTime format to interpret the beginning and ending bound elements.

- `'('''beginning_element_value''',Δ''ending_element_value''')'`
- `'(beginning_element_value,Δending_element_value)'`
- `'('''beginning_element_value''',Δending_element_value)'`
- `'(beginning_element_value,Δ''ending_element_value''')'`

where formats of *beginning_element_value* and *ending_element_value* depend on the target data type.

Target Data Type	Format
PERIOD(DATE)	YYYY-MM-DD
PERIOD(TIME[(n)])	HH:MI:SS.S(F)
PERIOD(TIMESTAMP[(n)])	YYYY-MM-DDBHH:MI:SS.S(F)

For the meanings of the format characters, see the description of the FORMAT phrase in *SQL Data Types and Literals*.

Example

In the following example, two concatenated character literals are cast as PERIOD(TIMESTAMP(2)). The output is adjusted according to the current session time zone during display. Assume the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE and the format derived from SDF is 'YYYY-MM-DDBHH:MI:SS.S(2)Z'.

```
SELECT CAST('(''2005-02-02 12:12:12.34+08:00'', ' ||
       '''2006-02-03 12:12:12.34+08:00'')'
       AS PERIOD(TIMESTAMP(2)));
```

The following PERIOD(TIMESTAMP(2)) value is returned:

```
('2005-02-01 20:12:12.34', '2006-02-02 20:12:12.34')
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-TIME Conversion

Purpose

Converts a character data string to a TIME or TIME WITH TIME ZONE value.

CAST Syntax

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a TIME type.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative output formatting for the time data.

Note: TIME (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIME value to UTC based on the current session time zone or on a specified time zone.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a TIME type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

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

Implicit Character-to-TIME Conversion

In field mode, the string must conform to the format of the target TIME type.

In record or indicator mode, the string must use the ANSI TIME format.

Usage Notes

The character value is trimmed of leading and trailing pad characters and handled as if it were a string literal in the declaration of a TIME string literal.

If the contents of the string can be converted to a valid TIME, the conversion is made; otherwise, an error is returned to the application.

Character-to-TIME conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or GRAPHIC to TIME.

You can use a FORMAT phrase to specify an explicit format for the TIME target data type. A conversion that does not specify a FORMAT phrase uses the default format for the TIME data type.

IF the character string is converted to ...	THEN the default format ...
TIME	does not use the time zone formatting character and does not display a time zone.
TIME WITH TIME ZONE	uses the time zone formatting character to display the time zone.

For more information on default formats and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Conversions That Include Time Zone

The following rules apply to character-to-TIME conversions that include time zone information:

- If the target data type does not specify a time zone, for example, TIME(0), the source character string may contain a time zone of the format +hh:mi or -hh:mi, but only if it appears immediately before or immediately after the time.

  For example, the following conversion is successful:

  ```
  SELECT CAST ( '-02:0011:23:44'
  AS TIME(0) );
  ```

  The following conversion is not successful because of the blank separator character between the time zone and the time:

  ```
  SELECT CAST ( '+02:00 11:23:44.56'
  AS TIME(2) );
  ```

- If the source character string contains a time zone, and the target data type does not specify a time zone, for example, TIME(0), the conversion uses the time zone in the character string to convert the character string to Universal Coordinated Time (UTC). This is done regardless of whether the FORMAT phrase contains the time zone formatting character.

  ```
  SELECT CAST ('10:15:12+12:30'
  AS TIME(0));
  ```

- If the source character string does not contain a time zone, and the target data type specifies a time zone and a target FORMAT phrase that includes time zone formatting characters, the output includes the session time zone.

  ```
  SELECT CAST ('10:15:12'
  AS TIME(0) WITH TIME ZONE FORMAT 'HH:MI:SSBZ');
  ```

- If both the source character string and the target data type do not specify a time zone, the source character string is internally converted to UTC based on the current session time zone.

Conversions That Include Fractional Seconds

The following rules apply to conversions that include fractional seconds:

- The fractional seconds precision in the source character string must be less than or equal to the fractional seconds precision specified by the target type.

  ```
  SELECT CAST('12:30:25.44' AS TIME(3));
  ```

  If no fractional seconds appear in the source character string, then the fractional seconds precision is always less than or equal to the target data type fractional seconds precision, because the valid range for the precision is zero to six, where the default is six.

  ```
  SELECT CAST('12:30:25' AS TIME(3));
  ```

- If the target data type is defined by a FORMAT phrase, the fractional seconds precision formatting characters must be greater than or equal to the precision specified by the data type.

  ```
  SELECT CAST('12h:15.12s:30m'
  AS TIME(4) FORMAT 'HHh:SSDS(4)s:MIm');
  ```

  A FORMAT phrase must specify a fractional seconds precision of six if the target data type does not specify a fractional seconds precision, because the default precision is six.

  ```
  SELECT CAST ('12:30:25' AS TIME FORMAT 'HH:MI:SSDS(6)');
  ```

Character Strings That Omit Hour, Minute, or Second

If the character string in a character-to-TIME conversion omits the hour, minute, or second, the system uses default values for the target TIME value.

IF the character string omits the ...	THEN the system uses the ...
hour	value of 0.
minute	
second	

Consider the following table:

```
CREATE TABLE time_log
   (id INTEGER
   ,start_time TIME
   ,end_time TIME
   ,log_time TIME);
```

The following INSERT statement converts three character strings to TIME values. The first character string omits the hour, the second character string omits the minute, and the third character string omits the second.

```
INSERT time_log
   (1001
   ,CAST ('01:02.030405' AS TIME FORMAT 'MI:SS.S(6)')
   ,CAST ('01:02.030405' AS TIME FORMAT 'HH:SS.S(6)')
   ,CAST ('01:02' AS TIME FORMAT 'HH:MI'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM time_log;

          id       start_time         end_time         log_time
-----------  ---------------  ---------------  ---------------
        1001  00:01:02.030405  01:00:02.030405  01:02:00.000000
```

FORMAT Phrase Restrictions

In character-to-TIME conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- Z
- T

Example 1: Fractional Seconds

This query returns the value '12:23:39.999900' (with the fractional seconds extended to 6 places as requested by CASTing to a TIME(6) type).

```
SELECT CAST(' 12:23:39.9999 '
AS TIME(6));
```

Example 2: Truncation of Non-pad Character Data

This query returns an error because the requested conversion requires truncation of non-pad character data.

```
SELECT CAST(' 12:23:39.9999 '
AS TIME(3));
```

Example 3: Non Valid MINUTE Value

This query returns an error because the MINUTE value of 63 is not valid.

```
SELECT CAST(' 12:63:39.9999 '
AS TIME(6));
```

Example 4: FORMAT Phrase

This query returns the value '15h33m'.

```
SELECT CAST('15h33m'
AS TIME(0) FORMAT 'HHhMIm');
```

Example 5: Implicit Conversion of Character to TIME

The following CREATE TABLE statement specifies a FORMAT phrase for the TIME data type column:

```
CREATE SET TABLE timetab (f1 TIME(0) FORMAT 'TBHHhMImSSs');
```

In field mode, the following INSERT statement successfully performs the character to TIME implicit conversion because the format of the string conforms to the format of the TIME column in the timetab table:

```
INSERT INTO timetab ('AM 10h20m30s');
```

In record or indicator mode, the following INSERT statement successfully performs the character to TIME implicit conversion because the format of the string is in the ANSI TIME format:

```
INSERT timetab ('11:23:34');
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-TIMESTAMP Conversion

Purpose

Converts a character data string to a TIMESTAMP or TIMESTAMP WITH TIME ZONE value.

CAST Syntax

1101A247

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a TIMESTAMP type.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
timestamp_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative formatting for the time data.

Note: TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a TIMESTAMP type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

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

Implicit Character-to-TIMESTAMP Conversion

In field mode, the string must conform to the format of the target TIMESTAMP type.

In record or indicator mode, the string must use the ANSI TIMESTAMP format.

Usage Notes

The source expression is trimmed of leading and trailing pad characters and then handled as if it were a string literal in the declaration of a TIMESTAMP string literal.

Character-to-TIMESTAMP conversion is supported for CHAR and VARCHAR types only. You cannot convert a character data type of CLOB or GRAPHIC to TIMESTAMP.

If the contents of the string can be converted to a valid TIMESTAMP value, then the conversion is performed; otherwise an error is returned.

You can use a FORMAT phrase to specify an explicit format for the TIMESTAMP target data type. A conversion that does not specify a FORMAT phrase uses the default format for the TIMESTAMP data type.

IF the character string is converted to ...	THEN the default format ...
TIMESTAMP	does not use the time zone formatting character and does not display a time zone.
TIMESTAMP WITH TIME ZONE	uses the time zone formatting character to display the time zone.

For more information on default formats and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example

The following query returns '2007-12-31 23:59:59.999999-08:00'.

```
SELECT CAST('2007-12-31 23:59:59.999999'
AS TIMESTAMP(6) WITH TIME ZONE);
```

Notice that the source character string did not need to have explicit Time Zone fields for this conversion to work properly.

Conversions That Include Time Zone

The following rules apply to character-to-TIMESTAMP conversions that include time zone information:

- If the target data type does not specify a time zone, for example, TIMESTAMP(0), the source character string may contain a time zone of the format +hh:mi or -hh:mi, but only if it appears immediately before or immediately after the time.

  For example, the following conversion is successful:

  ```
  SELECT CAST ( '2008-09-19 11:23:44-02:00'
  AS TIMESTAMP(0) FORMAT 'Y4-MM-DDBHH:MI:SSBZ' );
  ```

  The following conversion is not successful because of the blank separator character between the time zone and the time:

  ```
  SELECT CAST ( '2008-01-19 +02:00 11:23:44'
  AS TIMESTAMP(0) FORMAT 'Y4-MM-DDBZBHH:MI:SS' );
  ```

- If the source character string contains a time zone, and the target data type does not specify a time zone, the conversion uses the time zone in the character string to convert the character string to Universal Coordinated Time (UTC). This is done whether or not the FORMAT phrase contains the time zone formatting character.

  ```
  SELECT CAST ('2002-02-20 10:15:12+12:30' AS TIMESTAMP(0));
  ```

- If the target FORMAT phrase includes time zone formatting characters, and the source character string does not contain a time zone, the output includes the session time zone. This is done whether or not the target data type specifies a time zone.

  ```
  SELECT CAST ('2002-02-20 10:15:12'
  AS TIMESTAMP(0) WITH TIME ZONE FORMAT 'Y4-MM-DDBHH:MI:SSBZ');
  ```
- If both the source character string and the target data type do not specify a time zone, the source character string is internally converted to UTC based on the current session time zone.

Conversions That Include Fractional Seconds

The following rules apply to conversions that include fractional seconds:

- The fractional seconds precision in the source character string must be less than or equal to the fractional seconds precision specified by the target type.

  ```
  SELECT CAST('2002-01-01 12:30:25.44' AS TIMESTAMP(3));
  ```

  If no fractional seconds appear in the source character string, then the fractional seconds precision is always less than or equal to the target data type fractional seconds precision, because the valid range for the precision is zero to six, where the default is six.

  ```
  SELECT CAST('2002-01-01 12:30:25' AS TIMESTAMP(3));
  ```

- If the target data type is defined by a FORMAT phrase, the fractional seconds precision formatting characters must be greater than or equal to the precision specified by the data type.

  ```
  SELECT CAST('12-02-07 12:30:25' AS TIMESTAMP(3)
  FORMAT 'DD-MM-YYBHH:MI:SSDS(3)');
  ```

  A FORMAT phrase must specify a fractional seconds precision of six if the target data type does not specify a fractional seconds precision, because the default precision is six.

  ```
  SELECT CAST('12-02-07 12h:15.12s:30m'
  AS TIMESTAMP FORMAT 'DD-MM-YYBHHh:SSDS(6)s:MIm');
  ```

Character Strings That Omit Day, Month, Year, Hour, Minute, or Second

If the character string in a character-to-TIMESTAMP conversion omits the day, month, year, hour, minute, or second, the system uses default values for the target TIMESTAMP value.

IF the character string omits the ...	THEN the system uses the ...
day	value of 1 (the first day of the month).
month	value of 1 (the month of January).
year	current year.
hour	value of 0.
minute	
second	

Consider the following table:

```
CREATE TABLE timestamp_log
(id INTEGER, start_ts TIMESTAMP, end_ts TIMESTAMP);
```

The following INSERT statement converts two character strings to TIMESTAMP values. Both strings omit the hour, minute, and second. Additionally, the first character string omits the day and the second character string omits the month.

```
INSERT timestamp_log
   (1001
   ,CAST ('January 2006' AS TIMESTAMP FORMAT 'MMMMBYYYY')
   ,CAST ('2006-01' AS TIMESTAMP FORMAT 'YYYY-DD'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM timestamp_log;

         id                    start_ts                      end_ts
-----------  --------------------------  --------------------------
       1001  2006-01-01 00:00:00.000000  2006-01-01 00:00:00.000000
```

Here is an INSERT statement where both character strings omit the year. Additionally, the first character string omits the hour and the second character string omits the minute. Assume the current year is 2003.

```
INSERT timestamp_log
   (1002
   ,CAST ('January 23 04:05' AS TIMESTAMP FORMAT 'MMMMBDDBMI:SS')
   ,CAST ('01-23 04:05' AS TIMESTAMP FORMAT 'MM-DDBHH:SS'));
```

The result of the INSERT statement is as follows:

```
SELECT * FROM timestamp_log WHERE id = 1002;

         id                    start_ts                      end_ts
-----------  --------------------------  --------------------------
       1001  2003-01-23 00:04:05.000000  2003-01-23 04:00:05.000000
```

Restrictions on FORMAT Phrase

In character-to-TIMESTAMP conversions, the FORMAT phrase must not consist solely of the following formatting characters:

- EEEE
- E4
- EEE
- E3
- T
- Z

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character-to-UDT Conversion

Purpose

Converts a character data string to a UDT.

CAST Syntax

```
── CAST ── (character_expression ── AS ── UDT_data_definition ) ──
```

1101A336

where:

Syntax element ...	Specifies ...
character_expression	a character expression to be cast to a UDT.
UDT_data_definition	the UDT type to which *character_expression* is to be converted.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit character-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Character-to-UDT Conversion

Teradata Database performs implicit Character-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

The source character type of the cast definition does not have to be an exact match to the source character type of the implicit conversion. Teradata Database can use an implicit cast definition that specifies a CHAR, VARCHAR, or CLOB source type.

If multiple implicit cast definitions exist for converting different character types to the UDT, Teradata Database uses the implicit cast definition for the character type with the highest precedence. The following list shows the precedence of character types in order from lowest to highest precedence:

- CHAR
- VARCHAR
- CLOB

For non-CLOB character types, if no Character-to-UDT implicit cast definitions exist, Teradata Database looks for other cast definitions that can substitute:

IF the following combination of implicit cast definitions exists ...				THEN Teradata Database ...
Numeric-to-UDT	DATE-to-UDT	TIME-to-UDT	TIMESTAMP-to-UDT	
X				uses the numeric-to-UDT implicit cast definition. If multiple numeric-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.
	X			uses the DATE-to-UDT implicit cast definition.
		X		uses the TIME-to-UDT implicit cast definition.
			X	uses the TIMESTAMP-to-UDT implicit cast definition.
X	X			reports an error.
X		X		
X			X	
	X	X		
	X		X	
		X	X	
X	X	X		
X	X		X	
X		X	X	
	X	X	X	
X	X	X	X	

Substitutions are valid because Teradata Database can implicitly cast the non-CLOB character type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Character Data Type Assignment Rules

Server Character Sets

LATIN, UNICODE, KANJISJIS, KANJI1, and GRAPHIC server character sets are generally mutually assignable.

Consider an assignment of an expression to a character string column. The assignment may be the result of the SQL UPDATE or INSERT statement, or it may be the result of a Load utility assignment.

The expression is converted to the server character set of the character column.

Exceptions to GRAPHIC Data

The following exceptions apply to GRAPHIC data:

- When you import GRAPHIC data and assign it to a character column, that column must be defined as GRAPHIC.
- When you import character data that is not GRAPHIC, you cannot assign it to a column defined as GRAPHIC.

  For more information, see the documentation on the USING row descriptor in *SQL Data Manipulation Language*.
- You cannot assign non-GRAPHIC data to a GRAPHIC column from BTEQ or load utilities.

  For more information, see the documentation on the USING row descriptor in *SQL Data Manipulation Language*.
- You cannot assign or export GRAPHIC data from a single byte character set like ASCII or EBCDIC.

DATE-to-Character Conversion

Purpose

Converts a DATE value to a character string.

CAST Syntax

1101A248

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be cast to a character string.
character_data_type	the character data type to which *date_expression* is to be converted.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

Teradata Conversion Syntax

1101B259

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be cast to a character string.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character data type to which *date_expression* is to be converted.
server_character_set	the server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default character set is used for the conversion.

ANSI Compliance

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

Usage Notes

When converting DATE to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the DATE value as represented by a character string literal.

IF the target data type is ...	AND *n* is ...	THEN ...
CHAR(*n*)	greater than the length of the DATE value as represented by a character string literal	trailing pad characters are added to pad the representation.
	too small	a string truncation error is returned.
VARCHAR(*n*)	greater than the length of the DATE value as represented by a character string literal	no blank padding is added to the character representation.
	too small	a string truncation error is returned.

Restrictions

DATE types cannot be implicitly or explicitly converted to character types if the server character set is GRAPHIC.

DATE to CLOB conversion is not supported.

Forcing a FORMAT on CAST for Converting DATE to Character

The default format for DATE to character conversion uses the format in effect for the DATE value.

To override the default format, you can convert a DATE value to a string using a FORMAT phrase. The resulting format, however, is the same as the DATE value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from DATE to CHAR and force an explicit FORMAT on the result regardless of the format associated with the DATE value. This is because of the rules for matching FORMAT phrases to data types.

Example 1

The dateform mode of the session is INTEGERDATE and column F1 in the table INTDAT is a DATE value with the explicit format 'YYYY,MMM,DD'.

```
SELECT F1 FROM INTDAT ;
```

The result (without a type change) is the following report:

```
F1
----------
1900,Dec,31
```

Assume that you want to convert this to a value of CHAR(12), and an explicit output format of 'MMMBDD,BYYYY'. Use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```
SELECT
CAST( (CAST (F1 AS FORMAT 'MMMBDD,BYYYY')) AS CHAR(12))
FROM INTDAT;
```

The result after the nested CASTs is the following report.

```
F1
------------
Dec 31, 1900
```

The inner CAST establishes the display format for the DATE value and the outer CAST indicates the data type of the desired result.

Example 2

Suppose you need to create a script to convert date values to the ANSI DATE format, regardless of the source of the DATE value or the DATEFORM mode of the session.

You can use nested CASTs and a FORMAT to do this as demonstrated by the example that follows.

```
SELECT
CAST( (CAST (F1 AS FORMAT 'YYYY-MM-DD')) AS CHAR(10))
FROM INTDAT;
```

The result after the nested CASTs is the following report.

```
F1
----------
1900-12-31
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-DATE Conversion

Use DATE-to-DATE conversion to convert the format or title of a DATE type.

CAST Syntax

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
date_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

CAST permits the use of data attributes, such as the FORMAT phrase that enables alternative output formatting of date data. This is a Teradata extension to CAST.

A DATE-to-DATE conversion involving a DATE type with a dateform of INTEGERDATE is a Teradata extension to the ANSI SQL:2011 standard. This is a Teradata extension to CAST

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

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

Example

Consider a table named employee that was created with a session dateform mode of INTEGERDATE where dob is a DATE column with a format of M3BDDBY4. To list employees who were born between January 30, 1938, and March 30, 1943, you could specify the date information as follows:

```
SELECT name, dob
FROM employee
WHERE dob BETWEEN 'Jan 30 1938' AND 'Mar 30 1943'
ORDER BY dob;
```

The result returns the date of birth information as specified for the Employee table:

```
Name                  DOB
----------    -----------
Inglis C      Mar 07 1938
Peterson J    Mar 27 1942
```

To change the date format to an alternate form, change the SELECT to:

```
SELECT name, dob (FORMAT 'yy-mm-dd')
FROM employee
WHERE dob BETWEEN 'Jan 30 1938' AND 'Mar 30 1943'
ORDER BY dob ;
```

The format specification changes the display to the following:

```
Name               DOB
----------    --------
Inglis C      38-03-07
Peterson J    42-03-27
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-Numeric Conversion

Introduction

DATE data may be converted to the following numeric types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- DECIMAL(*n*,*m*)
- NUMBER
- FLOAT

CAST Syntax

1101A250

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
numeric_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of numeric data attribute phrases.

Teradata Conversion Syntax

1101B261

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
numeric_data_type	the target numeric type to which the date expression is to be converted.

ANSI Compliance

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

Usage Notes

When a date is converted to a numeric, the value returned is the integer value for the internal stored date, which is encoded using the following formula:

```
(year - 1900) * 10000 + (month * 100) + day
```

Allowable date values range from AD January 1, 0001 to AD December 31, 9999.

For example, December 31, 1985 would be stored as the integer 851231; July 4, 1776 stored as -1239296; and March 30, 2041 stored as 1410330.

Conversion of DATE to DECIMAL(*n*,*m*) where the number of digits (*n*) is too small generates a numeric overflow error. Conversion of DATE to BYTEINT or SMALLINT generates a numeric overflow error if the value returned is outside the range of values that the data type can represent.

No error is generated on conversion of DATE to INTEGER or FLOAT.

FORMAT Phrase

A FORMAT phrase in DATE to numeric conversion may only contain the 9 or Z formatting character. For example:

```
SELECT CAST (DATE '2007-12-31' AS INTEGER FORMAT '9999999');
```

Implicit DATE-to-Numeric Conversion

Teradata Database performs implicit DATE-to-numeric type conversion when you assign a DATE type to a numeric type, compare a DATE type and numeric type, or pass a DATE type to a system function that takes a numeric type.

Example

The following example converts DATE data in the dob column of the employee table to a numeric format.

Note that the best practice is to define date data as a DATE type; do not define date data as a numeric type.

To change the display from date format to integer format, change the statement to:

```
SELECT name, dob (INTEGER)
FROM employee
WHERE dob BETWEEN 380307 AND 420825
ORDER BY dob ;
```

or

```
SELECT name, CAST (dob AS INTEGER)
FROM employee
WHERE dob BETWEEN 380307 AND 420825
ORDER BY dob ;
```

and the display becomes:

```
Name             DOB
----------    ------
Inglis C      380307
Peterson J    420327
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-Period Conversion

Casts as PERIOD(DATE) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]).

CAST Syntax

```
── CAST ── ( date_expression ─ AS ─ period_data_type ──┬──────────────────────┬──── ) ────
                                                       │ ┌─────────────────┐ │
                                                       └─┴─ period_data_attribute ─┴─
                                                                                  1101A589
```

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
period_data_type	the target Period type to which the date expression is to be converted.
period_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Usage Notes

A DATE value can be cast as PERIOD(DATE) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function. If an attempt is made to cast a DATE value as PERIOD(TIME[(n)] [WITH TIME ZONE]), an error is reported.

If the target type is PERIOD(DATE), the result beginning element is set to the source value. The result ending element is set to the result beginning bound plus one granule of the target type (that is, INTERVAL '1' DAY). If the result ending bound exceeds the maximum DATE value (that is, the source value is equal to the maximum DATE value), or the result ending bound equal to maximum DATE value (that is, the resulting ending bound value equal to value of UNTIL_CHANGED) an error is reported.

If the target type is PERIOD(TIMESTAMP[(n)]), the result beginning element is set to the UTC value obtained using the current session time zone and a timestamp value formed from

the source DATE value and a time portion of zero. The result ending element is set to the result beginning bound plus one granule of the target type (note that this cannot cause an error).

If the target type is PERIOD(TIMESTAMP[(n)] WITH TIME ZONE), the time portion of the result beginning element is set to the UTC value obtained using the current session time zone and a timestamp value formed from the source DATE value and a time portion of zero. The time zone of the result beginning element is set to the current session time zone displacement. The result ending element is set to the result beginning bound plus one granule of the target type (note that this cannot cause an error).

Note: The result has the same value for the beginning bound and last value.

Example 1

In the following example, a DATE literal is cast as PERIOD(DATE). The result beginning bound is obtained from the source. The result ending element is set to the result beginning bound plus INTERVAL '1' DAY.

```
SELECT CAST(DATE '2005-02-03' AS PERIOD(DATE));
```

The following PERIOD(DATE) value is returned:

```
('2005-02-03', '2005-02-04')
```

Example 2

In the following example, a DATE literal is cast as PERIOD(TIMESTAMP(4)). The result beginning bound is formed from the DATE literal and a time portion of zero. The result ending element is set to the result beginning bound plus INTERVAL '0.0001' SECOND.

```
SELECT CAST(DATE '2005-02-03' AS PERIOD(TIMESTAMP(4)));
```

The following PERIOD(TIMESTAMP(4)) value is returned:

```
('2005-02-03 00:00:00.0000', '2005-02-03 00:00:00.0001')
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-TIMESTAMP Conversion

Purpose

Converts a DATE value to a TIMESTAMP or TIMESTAMP WITH TIME ZONE value.

CAST Syntax

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used. This is the default.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used for the CAST. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

Syntax element ...	Specifies ...
timestamp_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting of timestamp data.

The AT clause is ANSI SQL:2011 compliant.

Note: As an extension to ANSI, the AT clause is supported when converting from DATE to TIMESTAMP using CAST. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
date_expression	a date expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

Syntax element ...	Specifies ...
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used. This is the default.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used for the conversion. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

Note: As an extension to ANSI, the AT clause is supported when converting from DATE to TIMESTAMP using Teradata Conversion Syntax. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Usage Notes

The following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than zero, trailing zeros are added in the result to adjust the precision.

IF you specify...	THEN...
AT LOCAL	a local timestamp value is formed from the source *date_expression* with the time portion set to '00:00:00'. Then, the result is formed from this local timestamp value adjusted to UTC by subtracting the time zone displacement based on the current session time zone. This is the same as not specifying the AT clause.

IF you specify...	THEN...
AT *expression* or AT TIME ZONE *expression*	a local timestamp value is formed from the source *date_expression* with the time portion set to '00:00:00'. Then, the result is formed from this local timestamp value adjusted to UTC by subtracting the time zone displacement defined by *expression*.
AT *time_zone_string* or AT TIME ZONE *time_zone_string*	a local timestamp value is formed from the source *date_expression* with the time portion set to '00:00:00'. The time zone displacement is determined based on *time_zone_string* and the local timestamp value. Then, the result is formed from the local timestamp value adjusted to UTC by subtracting the time zone displacement.

Implicit DATE-to-TIMESTAMP Conversion

Teradata Database performs implicit conversion from DATE to TIMESTAMP types in some cases. See "Implicit Conversion of DateTime types" on page 582.

The following conversions are supported:

From source type...	To target type...
DATE[a]	TIMESTAMP
	TIMESTAMP WITH TIME ZONE

a. ANSIDate dateform mode or IntegerDate dateform mode

The TIMESTAMP value is always converted to DATE in case of comparison. See "TIMESTAMP-to-DATE Conversion" on page 727.

Example 1

In this example, the result of the CAST is the timestamp formed from the source expression value '2008-05-14' and the default time '00:00:00' adjusted to UTC by the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE. Thus, the value of the CAST is '2008-05-13 23:00:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 00:00:00'.

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0));
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) AT LOCAL);
```

Example 2

In this example, the result of the CAST is the timestamp formed from the source expression value '2008-05-14' and the default time '00:00:00' adjusted to UTC by the current session time zone displacement, INTERVAL '06:00' HOUR TO MINUTE. Thus, the value of the CAST is

'2008-05-13 18:00:00' at UTC with the current session time zone displacement INTERVAL '06:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to its time zone displacement, INTERVAL '06:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 00:00:00+06:00'.

```
SET TIME ZONE INTERVAL '06:00' HOUR TO MINUTE;

SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) WITH TIME ZONE);
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) WITH TIME ZONE
   AT LOCAL);
```

Example 3

In the following SELECT statement, the result of the CAST is the timestamp formed from the date '2008-05-14' and the default time '00:00:00' adjusted to UTC by the specified time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. Thus, the value of the CAST is '2008-05-14 08:00:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '05:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 13:00:00'.

```
SET TIME ZONE INTERVAL '05:00' HOUR TO MINUTE;

SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) AT -8);
```

Consider the following SELECT statement:

```
SELECT CAST(DATE '2008-05-14' AS TIMESTAMP(0) WITH TIME ZONE AT -8);
```

In this case, the result of the CAST is the timestamp formed from the source expression value '2008-05-14' and the default time '00:00:00' adjusted to UTC by the specified time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. Thus, the value of the CAST is '2008-05-14 08:00:00' at UTC with the specified time zone displacement INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to its time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 00:00:00-08:00'. The current session time zone has no effect.

Example 4

In this example, the current timestamp is:

```
             Current TimeStamp(6)
--------------------------------
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the DATE value '2010-03-09' to a TIMESTAMP value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(DATE '2010-03-09' AS TIMESTAMP(0) AT 'America Pacific');
```

The result of the query is:

```
         2010-03-09
-------------------
2010-03-09 08:00:00
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

DATE-to-UDT Conversion

Purpose

Converts DATE data to UDT data.

CAST Syntax

```
── CAST ── ( date_expression ── AS ── UDT_data_definition ) ──
```

1101A337

where:

Syntax element ...	Specifies ...
date_expression	a DATE expression to be cast to a UDT.
UDT_data_definition	the UDT type to which *date_expression* is to be converted.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit DATE-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit DATE-to-UDT Conversion

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

Teradata Database performs implicit DATE-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

If no DATE-to-UDT implicit cast definition exists, Teradata Database looks for other cast definitions that can substitute:

IF the following combination of implicit cast definitions exists ...		THEN Teradata Database ...
Numeric-to-UDT	Character[a]-to-UDT	
X		uses the Numeric-to-UDT implicit cast definition. If multiple Numeric-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.
	X	uses the Character-to-UDT implicit cast definition. If multiple Character-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.
X	X	reports an error.

a. a non-CLOB character type

Substitutions are valid because Teradata Database can implicitly cast a DATE type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

INTERVAL-to-Character Conversion

Purpose

Use CAST syntax or Teradata explicit conversion syntax to convert an INTERVAL type to its canonical character string representation.

INTERVAL-to-Character conversion is supported for CHAR and VARCHAR types only. The target type cannot be CLOB.

CAST Syntax

where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.
character_data_type	the target character type to which the interval expression is to be converted.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the target character type to which the interval expression is to be converted.
server_character_set	which server character set to use for the conversion. If the CHARACTER SET clause is omitted, the user default character set is used to convert the INTERVAL expression.

ANSI Compliance

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

INTERVAL-to-Fixed CHARACTER Conversion

When the target data type is CHAR(*n*), then *n* must be equal to or greater than the length of the canonical form of the value as represented by a character string literal.

If *n* is greater than that length, trailing pad characters are added to pad the canonical representation.

If *n* is too small, then a string truncation error is returned.

INTERVAL-to-VARCHAR Conversion

When the target data type is VARCHAR(*n*), then *n* must be equal to or greater than the length of the canonical form of the value as represented by a varying character string literal.

If *n* is too small, then a string truncation error is returned.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

INTERVAL-to-INTERVAL Conversion

CAST Syntax

where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.
interval_data_type	the target INTERVAL type to which the interval expression is to be converted.
interval_data_attribute	one of the following optional data attributes: • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.

Syntax element ...	Specifies ...
interval_data_type	the optional target INTERVAL type to which the interval expression is to be converted.
data_attribute	one of the following optional data attributes: • NAMED • TITLE

ANSI Compliance

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

Compatible Types

Both data types must be from the same INTERVAL family: either Year-Month or Day-Time. Types cannot be mixed.

This INTERVAL data type ...	Belongs to this INTERVAL family ...
• INTERVAL YEAR • INTERVAL YEAR TO MONTH • INTERVAL MONTH	Year-Month
• INTERVAL DAY • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE • INTERVAL MINUTE TO SECOND • INTERVAL SECOND	Day-Time

Conversion of INTERVAL types is performed only when the fields and precisions are different.

Precision of Source and Target Types

A conversion can result in an overflow error if the precision of the target data type is smaller than the corresponding precision for the source data type.

If the least significant value of the source is lower than that of the target, then those source values having lower precision than the least significant field of the target are ignored. The result is truncation. Recovery from this action is installation-dependent.

If the most significant field in the source value has higher significance than the most significant field in the target value, then the higher order fields of the source are converted into

a scalar value of the precision of the most significant field in the target, using the factors of 12 months per year, 24 hours per day and so on.

If the compared scalar value overflows the defined precision for the target field, an error is returned.

Implicit INTERVAL-to-INTERVAL Conversion

Teradata Database performs implicit conversion from INTERVAL to INTERVAL data types in some cases. See "Implicit Conversion of DateTime types" on page 582.

Conversion of INTERVAL types is performed only when both data types are from the same INTERVAL family: either Year-Month or Day-Time. See "Compatible Types" on page 653.

Example 1: Least Significant Field in Source Lower Than Target

The following query converts '3-11' to '3'. Source is INTERVAL YEAR(2). The truncation completes the conversion.

```
SELECT CAST(INTERVAL '3-11' YEAR TO MONTH AS INTERVAL YEAR(2));
```

Example 2: Least Significant Field in Source Lower Than Target

The following query converts '135 12:37:25.26' to '3252'. Source is DAY(3) TO SECOND(2)

```
SELECT CAST(INTERVAL '135 12:37:25.26' DAY(3) TO SECOND(2) AS INTERVAL
HOUR(4));
```

Example 3: Least Significant Field in Source Higher Than Target

The following query converts '3' to '3-00'. Source is INTERVAL YEAR. The insertion of zeros completes the conversion.

```
SELECT CAST(INTERVAL '3' YEAR AS INTERVAL YEAR TO MONTH);
```

Example 4: Least Significant Field in Source Higher Than Target

The following query converts '135 00:00:00.0' to '3240:00:00.00' after you perform the additional conversion of multiplying 135 * 24 to obtain 3240, which is the final HOUR value. The source had a data type of DAY.

```
SELECT CAST(INTERVAL ' 135 00:00:00.0' DAY AS INTERVAL HOUR TO SECOND);
```

Example 5: Most Significant Field in Source Higher Than Target

The following query first treats the source INTERVAL value as '135 12' and then computes HOURS as (135*24)+12=3252. The result of the query is INTERVAL '3252' HOUR unless the precision for the target value is less than 4, in which case an error is returned. The source had a data type of DAY TO SECOND.

```
SELECT CAST(INTERVAL '135 12:37:25.26' DAY TO SECOND AS INTERVAL HOUR);
```

Example 6: Implicit Type Conversion During Assignment

Consider the following table which has an INTERVAL YEAR TO MONTH column:

```
CREATE TABLE TimeInfo
  (YrToMon INTERVAL YEAR TO MONTH);
```

If you insert data into the column using the following parameterized request, and you pass an INTERVAL YEAR or INTERVAL MONTH value to the request, Teradata Database implicitly converts the value to an INTERVAL YEAR TO MONTH value before inserting the value.

```
INSERT INTO TimeInfo
VALUES (?);
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

INTERVAL-to-Numeric Conversion

Purpose

Convert an INTERVAL with only one field to an exact numeric data type.

This numeric value is the value of the single numeric field in the INTERVAL record.

CAST Syntax

```
─CAST─(─ interval_expression ─AS─ numeric_data_type ┌──────────────────────┐─)─
                                                   └─ numeric_data_attribute ─┘
```

1101A254

where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.
numeric_data_type	the target numeric type to which the interval expression is to be converted.
numeric_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Teradata Conversion Syntax

```
─ interval_expression ─( ┌──────────────────────┐ numeric_data_type ┌──────────────────────┐─)──
                        └─ data_attribute , ─┘                     └─ , data_attribute ─┘
```

1101B265

where:

Syntax element ...	Specifies ...
interval_expression	an INTERVAL expression to be converted.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
numeric_data_type	the target numeric type to which the interval expression is to be converted.

ANSI Compliance

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

Implicit INTERVAL-to-Numeric Conversion

Teradata Database performs implicit conversion of an Interval data type to an exact numeric data type in some cases. See"Implicit Conversion of DateTime types" on page 582.

Example

Consider the following table definition:

```
CREATE TABLE sales_intervals
   ( sdate DATE
   , sinterval INTERVAL MONTH
   , stotals DECIMAL(5,0));
```

The following query uses CAST to convert INTERVAL MONTH values in the sinterval column to INTEGER.

```
SELECT stotals,
   (EXTRACT (MONTH FROM sdate)) + (CAST(sinterval AS INTEGER))
FROM sales_intervals;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

INTERVAL-to-UDT Conversion

Purpose

Converts interval data to UDT data.

CAST Syntax

```
── CAST ── ( interval_expression ── AS ── UDT_data_definition ) ──
```

1101A338

where:

Syntax element ...	Specifies ...
interval_expression	an interval expression to be cast to a UDT.
UDT_data_definition	the UDT type to which *interval_expression* is to be converted.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit INTERVAL-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit INTERVAL-to-UDT Conversion

Performing an implicit data type conversion requires a cast definition (see "Usage Notes" on page 658) that specifies the following:

- the AS ASSIGNMENT clause
- a source data type that is in the same INTERVAL family as the source of the implicit cast:

This INTERVAL data type ...	Belongs to this INTERVAL family ...
• INTERVAL YEAR • INTERVAL YEAR TO MONTH • INTERVAL MONTH	Year-Month
• INTERVAL DAY • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE • INTERVAL MINUTE TO SECOND • INTERVAL SECOND	Day-Time

The source data type of the cast definition does not have to be an exact match to the source of the implicit type conversion.

Teradata Database performs implicit INTERVAL-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Numeric-to-Character Conversion

Purpose

Converts a numeric data type to a character data type.

CAST Syntax

where:

Syntax element ...	Specifies ...
numeric_expression	the numeric data expression to be cast to a character type.
character_data_type	the character type to which the numeric data expression is to be converted.
data_attribute	one of the following optional data attributes: • CHARACTER SET • FORMAT • NAMED • TITLE If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
numeric_expression	the numeric data expression to be cast to a character type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character type to which the numeric data expression is to be converted. If *character_data_definition* does not specify a CHARACTER SET clause to indicate which server character set to use, the user default server character set is used.
server_character_set	which server character set to use. If the CHARACTER SET clause is not specified, the user default server character set is used.

ANSI Compliance

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

Implicit Numeric-to-Character Conversion

If a numeric argument in an SQL string function is implicitly converted to a CHAR or VARCHAR character type, and the format of the numeric argument includes any of the following formatting characters, the server character set of the character type is UNICODE:

- G
- F
- O
- A
- D
- L
- U
- I
- C
- N

For all other formats, the server character set is LATIN.

Numeric items cannot be converted to CLOB types or GRAPHIC characters.

For information on data type formats, formatting characters, and the FORMAT phrase, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

How CAST Differs from Teradata Conversion Syntax

The process for the CAST function is as follows:

1. Convert the numeric value to a character string using the default or specified format for the numeric value.
2. Trim leading and trailing pad characters.

3 Extend to the right as required by the target string length.

4 If truncation of non-pad characters is required to conform to the target string length, report string truncation error.

The CAST operation differs from the Teradata SQL conversion as follows:

- Results are left justified. Column displays are not aligned.
- Truncation of significant data generates a string truncation error.

Using Teradata conversion syntax (that is, not using CAST) for explicit conversion of *numeric-to-character* data requires caution.

The process is as follows:

1 Convert the numeric value to a character string using the default or specified FORMAT for the numeric value.

   Leading and trailing pad characters are not trimmed.

2 Extend to the right with pad characters if required, or truncate from the right if required, to conform to the target length specification.

   If non-pad characters are truncated, no string truncation error is reported.

For an example of numeric to character conversion that results in truncation of significant data, see "Example 1" on page 663.

Supported Character Types

Numeric to character conversion is supported for CHAR and VARCHAR types only. Numeric types cannot be converted to CLOB types.

Usage Notes

To convert a numeric type value to a character string, the character description must contain a data type declaration. A FORMAT phrase, by itself, cannot be used to convert a numeric type value to a character type value. The phrase only controls how to display the resultant value.

If the character description does not include a FORMAT phrase, then the format of the original numeric value determines how to display the data.

The Teradata conversion syntax form of numeric-to-character conversion uses explicit or default FORMATs to convert to a character representation. It then truncates or extends with pad characters, depending what length the character string dictates. This can lead to a loss of significance.

Attempting to convert from a numeric type to a character type that uses a GRAPHIC server character set generates an error.

As a general rule, you should store numbers as numeric data, not as character data. For example, a table is created with the following code:

```
CREATE TABLE job AS
   (job_code CHAR(6) PRIMARY KEY
   ,description CHAR(70) );
```

Subsequently, the following query is made:

```
SELECT job_code, description
FROM job
WHERE job_code = 1234;
```

The problem here is that '1234', '1234', '01234', '001234', '+1234', and so on, are all valid character representations of the numeric literal value, and the system cannot tell which value to use for hashing. Therefore, the system must do a full table scan to convert all job_code values to their numeric equivalents so that it can do the comparisons.

Example 1

T1.Field1 has a numeric INTEGER data type with the default format '-(10)9'. The user has values such as 123456, with no values of over 999999. The values, defined as being in INTEGER format, are to be converted to CHAR(8).

The following example illustrates the Teradata syntax for performing this numeric-to-character conversion.

```
SELECT Field1(CHAR(8)) FROM T1;
```

returns ' 123' for the value 123456, where the result includes 5 leading pad characters and truncates significant digits.

Example 2

Based on the following description of Salary, data is converted as illustrated in the following table (Δ = pad character):

```
Salary (DECIMAL(8,2), FORMAT '$$$,$$9.99')
```

Data	Conversion	Result
20000.00	Salary (CHAR(10))	'$20,000.00'
9000.00	Salary (CHAR(10))	'Δ$9,000.00'
20000.00	Salary (FORMAT'9(5)') (CHAR (5))	'20000'
9000.00	CAST (Salary AS CHAR(10))	'$9,000.00Δ'

The resultant character string is either extended with pad characters or truncated to conform to the given character description.

Example 3

Suppose EmpNo was defined as SMALLINT with the default format of '9(6)'. Suppose a value in EmpNo is 12501. The statement:

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

returns the '1250', with a leading pad character and the low order digit missing. The CAST function used for the same conversion, converts to the character representation of the

numeric value, trims leading pad characters, and finally truncates or pads on the right. For example, the following SELECT statement returns '12501'.

```
SELECT CAST (EmpNo AS CHAR(5)) FROM Employee;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Numeric-to-DATE Conversion

Purpose

Converts a numeric expression to a DATE data type.

CAST Syntax

```
— CAST — ( — numeric_expression — AS — DATE ┌──────────────────┐ ) ──────────────────
                                            │┌─ data_attribute ─┐│
                                            └└──────────────────┘┘
                                                                          1101B077
```

where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.
data_attribute	any of the following optional data attributes: • FORMAT • NAMED • TITLE A *date_data_definition* that specifies a FORMAT clause enables an alternative format. Specifying data attributes in CAST is a non-ANSI Teradata extension.

ANSI Compliance

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

Teradata Conversion Syntax

```
— numeric_expression — ( ┌──────────────────┐ DATE ┌──────────────────┐ ) ──────────
                         │┌─ data_attribute ,─┐│     │┌─ , data_attribute─┐│
                         └└──────────────────┘┘     └└──────────────────┘┘
                                                                          1101B385
```

where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE Specifying a FORMAT clause enables an alternative format.

ANSI Compliance

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

Translation of Numbers to Dates

Although not recommended, you can explicitly convert numbers to dates.

Teradata Database stores each DATE value as a four-byte integer using the following formula:

```
(year - 1900) * 10000 + (month * 100) + day
```

For example, December 31, 1985 would be stored as the integer 851231; July 4, 1776 stored as -1239296; and March 30, 2041 stored as 1410330.

The following table demonstrates how numeric dates are interpreted when inserted into a column. Note the translation of the third date, which was probably intended to be 1990-12-01.

This numeric value ...	Translates to this date value ...
901201	1990-12-01
1001201	2000-12-01
19901201	3890-12-01

Notice that this formula best fits two-digit dates in the 1900s. Because of the difficulty of using this format outside of the 1900s, dates are best specified as ANSI date literals instead.

Range of Allowable Values

Allowable date values range from AD January 1, 0001 (-18989899) to AD December 31, 9999 (80991231).

If the numeric value does not represent a valid date, an error is reported.

Numeric-to-DATE Implicit Type Conversion

Although not recommended, you can specify a numeric type in the assignment of a DATE type. Teradata Database performs implicit numeric-to-DATE type conversion prior to the assignment. The value of the numeric type must represent a valid date.

However, for comparison operations involving a numeric type operand and a DATE type operand, Teradata Database converts the DATE type to a numeric type. If you compare a numeric type and a DATE type and expect the comparison to be between two DATE types, you must explicitly convert the numeric type to a DATE type.

Example

This example casts the numeric integer expression to a date format.

```
SELECT CAST (1071201 AS DATE);
```

The result looks like this when the DateForm mode of the session is set to ANSIDate:

```
   1071201
----------
2007-12-01
```

Related Topics

FOR information on ...	SEE ...
implicit type conversion of operands for comparison operations	"Implicit Type Conversion of Comparison Operands" on page 502.
data type compatibility rules for assignments involving DateTime types	"ANSI DateTime and Interval Data Type Assignment Rules" on page 818.
data type compatibility rules for arithmetic operations involving DateTime types	"Arithmetic Operators and ANSI DateTime and Interval Data Types" on page 838.
data types and data attributes	*SQL Data Types and Literals*.

Numeric-to-INTERVAL Conversion

Purpose

Convert numeric data to an INTERVAL value with a single DateTime field.

CAST Syntax

```
─CAST─(─ numeric_expression ─AS─ interval_data_type ┌──────────────────────┐─)─
                                                   └─ interval_data_attribute ─┘
```

1101A281

where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.
interval_data_type	the target INTERVAL data type to which the numeric expression is being converted.
interval_data_attribute	one of the following optional data attributes: • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of interval data attribute phrases.

Teradata Conversion Syntax

```
─ numeric_expression ─( ┌──────────────────────┐ interval_data_type ┌──────────────────────┐─)──
                       └─ data_attribute ,─────┘                    └─ , data_attribute ────┘
```

1101B273

where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • NAMED • TITLE
interval_data_type	the target INTERVAL data type to which the numeric expression is being converted.

ANSI Compliance

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

Usage Notes

Numeric data is converted to an INTERVAL value with a single DateTime field.

If the numeric value is in the value range allowed for the INTERVAL, the value is used as the single field of the INTERVAL. Otherwise, an overflow error is returned.

Implicit Numeric-to-INTERVAL Conversion

Teradata Database performs implicit conversion of an exact numeric data type to an Interval data type in some cases. See "Implicit Conversion of DateTime types" on page 582.

Example

The following query returns ' -5' (with three leading pad characters).

```
SELECT CAST(-5 AS INTERVAL YEAR(4));
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Numeric-to-Numeric Conversion

Purpose

Converts a numeric expression defined with one data type to a different numeric data type.

CAST Syntax

where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.
numeric_data_type	the optional numeric data type to which *numeric_expression* is to be converted.
numeric_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI SQL, CAST permits data attributes such as the FORMAT phrase that enables an alternative format for *numeric_expression*.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
numeric_expression	an expression or existing field having a numeric data type.
numeric_data_type	the optional numeric data type to which *numeric_expression* is to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

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

Implicit Numeric-to-Numeric Conversion

Numeric items are converted to the same numeric type before any arithmetic or comparison operation is performed. The result returned is of this same underlying type.

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

For details on implicit type conversions for binary arithmetic expressions, see "Binary Arithmetic Result Data Types" on page 112.

For details on implicit type conversions for comparison operations, see "Implicit Type Conversion of Comparison Operands" on page 502.

Conversion to FLOAT/REAL/DOUBLE PRECISION

Because floating point numbers are not exact values, conversion of DECIMAL and integer values to FLOAT values might result in a loss of precision or produce a number that cannot be represented exactly. For example, a value like 0.1, when cast to FLOAT, no longer exactly equals to 0.1.

Truncation and Rounding During Conversion

Conversion of DECIMAL/NUMERIC to BIGINT, INTEGER, BYTEINT, or SMALLINT truncates any decimal portion. Conversion to DECIMAL produces a rounded result. If a range violation occurs, the operation may fail.

Conversion to FLOAT/REAL/DOUBLE PRECISION rounds to the nearest value available. Neither decimal fractions nor numbers greater than 9,007,199,254,740,992 can be guaranteed to be represented exactly, so the nearest representable value is chosen. If there are two representable values that qualify as the nearest value, then the representation with a '0' in the least significant bit is chosen. For example, 0.1, when stored in a FLOAT column, is rounded to a value slightly higher: 0.10000000000000000555111512312578270211815834045410156 25.

For details on rounding, see "Decimal/Numeric Data Types" in *SQL Data Types and Literals*.

Some examples of numeric conversions are:

Value	Converted To	Result
20000.99	INTEGER	20000
20000.99	DECIMAL(6,1)	20001.0
20000.99	DECIMAL(4, 1)	error
200000	SMALLINT	error

Using CAST in Applications With DECIMAL Type Size Restrictions

Some applications require DECIMAL types to have 15 digits or less.

Applications with this requirement may need to access DECIMAL columns that have more than 15 digits or use expressions that may produce DECIMAL results with more than 15 digits. To help with DECIMAL type size requirements, you can use CAST to convert DECIMAL types to a size of 15 or fewer digits.

For example, consider the following expression where A, B, and C are columns defined as DECIMAL(8,2):

```
SELECT (A*B)/C FROM table1;
```

The resulting value may be less than 15 digits, but A*B could be up to 18.

To ensure a result of less than 16 digits, use CAST:

```
SELECT CAST ((A*B)/C AS DECIMAL(15,2)) FROM table1;
```

Using CAST To Avoid Numeric Overflow

Because of the way the Teradata SQL compiler works, it is essential that you CAST the arguments of your expressions whenever large values are expected.

For example, suppose f1 is defined as DECIMAL(14,2) and you are going to multiply by an integer or get SUM(f1).

In this case, the following operations:

```
CAST(f1 AS DECIMAL(18,2))*100
```

or

```
SUM(CAST(f1 AS DECIMAL(18,2)))
```

are proper techniques for ensuring correct answer sets.

On the other hand, if you were to cast the *results* of the expressions, such as the following:

```
CAST(f1*100 AS DECIMAL(18,2))
```

or

```
CAST(SUM(f1) AS DECIMAL(18,2)
```

then you will likely experience overflow during the computations (and before the CAST is made)—not the desired result.

Example 1

This example casts the numeric integer expression named IntegerField to DECIMAL(7,2).

```
CAST (IntegerField AS DECIMAL (7,2))
```

Example 2

Although the FORMAT phrase cannot be used to change the underlying data type defined for a column, the phrase may be used to change the display for a numeric value.

For example, if the field values for columns Wholesale and Retail, both defined as DECIMAL(7,2), are 12467.75 and 21500.50, respectively, the result of the expression:

```
CAST (Wholesale - Retail AS FORMAT '-99999')
```

is:

```
-09033
```

A FORMAT phrase does not affect data that is returned to the client system in Record Mode (client system internal format).

In the previous example, the value returned to the client system is still in packed decimal format (for example, -9032.75).

The use of FORMAT in CAST is a Teradata extension to the ANSI standard.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Numeric-to-UDT Conversion

Purpose

Converts numeric data to UDT data.

CAST Syntax

```
── CAST ── ( numeric_expression ── AS ── UDT_data_definition ) ──
```

1101A334

where:

Syntax element ...	Specifies ...
numeric_expression	a numeric expression to be cast to a UDT.
UDT_data_definition	the UDT type, followed by any optional FORMAT, NAMED or TITLE data attribute phrases, to which *numeric_expression* is to be converted.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit numeric-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Numeric-to-UDT Conversion

Teradata Database performs implicit Numeric-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

The source numeric type of the cast definition does not have to be an exact match to the source numeric type of the implicit conversion. Teradata Database can use an implicit cast definition that specifies a BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, NUMBER, or REAL/FLOAT/DOUBLE target type.

If multiple implicit cast definitions exist for converting different numeric types to the UDT, Teradata Database uses the implicit cast definition for the numeric type with the highest precedence. The following list shows the precedence of numeric types in order from lowest to highest precedence:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- DECIMAL/NUMERIC
- NUMBER
- REAL/FLOAT/DOUBLE

If no numeric-to-UDT implicit cast definitions exist, Teradata Database looks for other cast definitions that can substitute:

IF the following combination of implicit cast definitions exists ...		THEN Teradata Database ...
DATE-to-UDT	Character-to-UDT	
X		uses the DATE-to-UDT implicit cast definition.
	X	uses the character-to-UDT implicit cast definition. The character type cannot be CLOB. If multiple character-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.
X	X	reports an error.

Substitutions are valid because Teradata Database can implicitly cast a numeric type to the substitute data type, and then use the implicit cast definition to cast from the substitute data type to the UDT.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-Character Conversion

Purpose

Converts a Period data type to its canonical character string representation.

Period-to-Character conversion is supported for CHAR and VARCHAR types only. The target type cannot be CLOB.

CAST Syntax

where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be cast to a character type.
character_data_type	the character type to which the Period data expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be cast to a character type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character type to which the Period data expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.

ANSI Compliance

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

Usage Notes

A Period expression can be cast as a character string representation using the CAST function or the Teradata cast syntax, or when forming the output for field mode. Assume *L* is the maximum length of the formatted character string for the format associated with the Period expression being cast. The resulting character string contains two strings representing the beginning and ending bounds of the Period expression, each up to length *L*, and each enclosed in apostrophes (' '), separated by comma and a space (,), and then enclosed within a left parenthesis and a right parenthesis [( )]. Thus, the maximum length of the resulting character string is 2**L*+8. Assume the actual length is *K* (which may be less than 2**L*+8, for example, if the format includes the full names of months and the specific month for a bound is July) and the target type is CHARACTER(*n*) or VARCHAR(*n*):

- If *n* is equal to *K*, the period is cast into the resulting character string of length *K*.
- If *n* is greater than *K* and the target is VARCHAR(*n*), the period is cast into the resulting character string with length *K*.

- If *n* is greater than *K* and the target is CHARACTER(*n*), the period is cast into the resulting character string and trailing pad characters are added to extend to length *n*.
- If *n* less than *K* and the session is in ANSI mode, a truncation error is reported.
- If *n* less than *K* and the session is in Teradata mode, a truncated string of length *n* is returned.

For data of Period data types with TIME and TIMESTAMP element types, the UTC value of the Period expression is adjusted to the time zone of the value or the current session time zone if the value does not have a time zone. The exception to conversion from UTC is for an ending bound of a PERIOD(TIMESTAMP(n)) value equal to the maximum value that is used to represent UNTIL_CHANGED; in this case, the value is not changed. Due to such adjustments, the ending bound may appear less than the beginning bound in the result, although in UTC the ending bound is greater than the beginning bound. This happens since the hour value for the TIME data type wraps over every 24 hours (that is, the hour value is obtained using 'module 24').

Example

Assume pts is a PERIOD(TIMESTAMP(2)) column in table t with a value of PERIOD '(2005-02-02 12:12:12.34, 2006-02-03 12:12:12.34)'.

In the following example, a PERIOD(TIMESTAMP(2)) column is cast as CHARACTER(52) using the CAST function.

```
SELECT CAST(pts AS CHARACTER(52)) FROM t;
```

The following is returned:

```
('2005-02-02 12:12:12.34', '2006-02-03 12:12:12.34')
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-DATE Conversion

Purpose

Converts Period data to a DATE value.

CAST Syntax

```
── CAST ── ( period_expression ─ AS ─ DATE ┬──────────────────────┬──────────────────────── ) ─
                                           │┌────────────────────┐│
                                           └┤ date_data_attribute ├┘
                                                                                  1101A600
```

where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be cast to a DATE type.
date_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of DATE data attribute phrases.

Usage Notes

A PERIOD(DATE) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as DATE using the CAST function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the source type is PERIOD(DATE), the result is the source beginning bound.

If the source type is PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), the result is the date portion of the source beginning bound after adjusting to the current session time zone.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]), an error is reported.

Example

Assume pd is a PERIOD(DATE) column in table t with a value of PERIOD '(2005-02-02, 2005-02-03)'.

In the following example, a PERIOD(DATE) column is cast as DATE. The result is the beginning bound of the column.

```
SELECT CAST(pd AS DATE) FROM t;
```

The following is returned:

```
2005-02-02
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-Period Conversion

CAST Syntax

where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be converted.
period_data_type	the optional Period type to which *period_expression* is to be converted.
period_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Compatible Types

The following table describes the allowed combinations of source and target types when both the source and the target types are Period data types.

Source Type	Target Type
PERIOD(DATE)	PERIOD(DATE)
	PERIOD(TIMESTAMP[(m)] [WITH TIME ZONE])

Source Type	Target Type
PERIOD(TIME[(n)] [WITH TIME ZONE])	PERIOD(TIME[(m)] [WITH TIME ZONE]) where *m* is the target precision, *m* must be greater than or equal to the source precision *n*. The default for m is 6.
	PERIOD(TIMESTAMP[(m)] [WITH TIME ZONE]) where *m* is the target precision, *m* must be greater than or equal to the source precision *n*. The default for m is 6.
PERIOD(TIMESTAMP[(n)] WITH TIME ZONE)	PERIOD(DATE)
	PERIOD(TIME[(m)] [WITH TIME ZONE]) where *m* is the target precision, *m* must be greater than or equal to the source precision *n*. The default for m is 6.
	PERIOD(TIMESTAMP[(m)] [WITH TIME ZONE]) where *m* is the target precision, *m* must be greater than or equal to the source precision *n*. The default for m is 6.

PERIOD(DATE) to PERIOD(TIMESTAMP)

A PERIOD(DATE) value can be cast as PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

The UTC value of the result elements are obtained after adjustment with respect to the current session time zone from the timestamps created by setting the date portion to the corresponding source elements and the time portions to 0. If the target type is PERIOD(TIMESTAMP[(n)] WITH TIME ZONE), both result time zone fields are set to the current session time zone displacement. An exception to this is if the source ending bound is the maximum DATE value; in that case, the result ending bound is set to the maximum TIMESTAMP value.

PERIOD(TIME) to PERIOD(TIME)

A PERIOD(TIME(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) using the CAST function.

The UTC value of the source is copied to the UTC value in the result. If the target type specifies WITH TIME ZONE and the source contains time zones, the time zone displacements from the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements. For example, assume the current session time zone displacement is INTERVAL - "08:00" HOUR TO MINUTE and the source PERIOD(TIME(0) WITH TIME ZONE) has the value PERIOD '(12:12:12+08:00, 12:12:13+08:00)'. The UTC value of this source is ('04:12:12', '04:12:13'). The UTC value of the result is set to this value. On output of this result, the UTC value is adjusted to the current session time zone and the result is ('20:12:12', '20:12:13').

Note: This value is actually for a previous day and, assuming that the CURRENT_DATE at UTC is DATE '2006-07-28', the output beginning bound would be '2006-07-27 20:12:12' if it was a timestamp element.

If the target precision is higher than the source precision, trailing zeros are appended to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

PERIOD(TIME) to PERIOD(TIMESTAMP)

A PERIOD(TIME(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

The source time values get adjusted with respect to the session time zone displacement from the corresponding UTC value. The date portion of each result element is set to CURRENT_DATE. The hour, minute, and, second are copied from the source after the above adjustment and the timestamp value is converted to corresponding UTC value.

If the target type specifies WITH TIME ZONE and the source contains time zones, the time zone displacements from the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements.

If the target precision is higher than the source precision, trailing zeros are appended to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

PERIOD(TIMESTAMP) to PERIOD(DATE)

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(DATE) using the CAST function.

The result elements are each set to the date portion of the corresponding source bound after the source bound is adjusted according to the current session time zone (the adjustment is not done for the source ending bound if it is the maximum value). If the adjustment for time zone changes the date, the changed value is used. If the result date portions are the same, an error is reported.

PERIOD(TIMESTAMP) to PERIOD(TIME)

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) using the CAST function.

The date portion in the beginning and ending UTC values of the source must have the same DATE value. Otherwise, an error is reported. The time portions of the result elements are copied from the corresponding source time portions. If the target type specifies WITH TIME ZONE and the source also contains time zones, the source time zone displacements are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements.

If the target precision is higher than the source precision, trailing zeros are added to the fractional seconds. If the target precision is lower than the source precision, an error is reported.

PERIOD(TIMESTAMP) to PERIOD(TIMESTAMP)

A PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

The result date and time portions are set to the corresponding source date and time portions. If the target type specifies WITH TIME ZONE and the source also contains time zones, the time zone displacements in the source are copied to the corresponding result elements. If the source does not contain time zones, the current session time zone displacement is copied to both result elements except if the source ending bound is the maximum value, the time zone for the result ending bound is +00:00.

If the target precision is higher than the source precision, trailing zeros are added in the fractional seconds. If the target precision is lower than the source precision, an error is reported.

Example 1: PERIOD(DATE) to PERIOD(TIMESTAMP)

Assume p is a PERIOD(DATE) column in table t1 with a value of PERIOD '(2005-02-02, 2006-02-03)' and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

In the following example, a PERIOD(DATE) column is cast as PERIOD(TIMESTAMP(6)). The date portion is obtained from the source for the corresponding result element and the time portions are set to zero.

```
SELECT CAST(p AS PERIOD(TIMESTAMP(6))) FROM t1;
```

The following is returned:

```
('2005-02-02 00:00:00.000000', '2006-02-03 00:00:00.000000')
```

Example 2: Least Significant Field in Source Lower Than Target

Assume p is a PERIOD(TIME(2)) column in table t with a value of PERIOD '(12:12:12.45, 13:12:12.67)' and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

In the following example, a PERIOD(TIME(2)) column is cast as PERIOD(TIME(6) WITH TIME ZONE). The time portion is obtained from the source with trailing zeros added to the fractional seconds to make the precision 6 for the corresponding result element and both result time zone fields are set to the current session time zone displacement.

```
SELECT CAST(p AS PERIOD(TIME(6)WITH TIME ZONE)) FROM t;
```

The following is returned:

```
('12:12:12.450000-08:00', '13:12:12.670000-08:00')
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-TIME Conversion

Purpose

Converts Period data to a TIME value.

CAST Syntax

where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of TIME data attribute phrases.

Usage Notes

A PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as TIME[(n)] [WITH TIME ZONE] using the CAST function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is reported.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), the result time portion is obtained from time portion of the source beginning bound. If both the source and target type are WITH TIME ZONE, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is WITH TIME ZONE, the result time zone field is set to the current session time zone displacement.

If the source type is PERIOD(DATE), an error is reported.

Example

Assume pt is a PERIOD(TIME(2)) column in table t with a value of PERIOD '(12:12:12.34, 12:12:12.35)'.

In the following example, a PERIOD(TIME(2)) column is cast as TIME(6). The TIME(6) result is obtained from the source beginning element with trailing zeros added to the fractional seconds to make the precision 6.

```
SELECT CAST(pt AS TIME(6)) FROM t;
```

The following is returned:

```
12:12:12.340000
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Period-to-TIMESTAMP Conversion

Purpose

Converts Period data to a TIMESTAMP value.

CAST Syntax

where:

Syntax element ...	Specifies ...
period_expression	the Period data expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
timestamp_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting of DateTime data.

Usage Notes

A PERIOD(DATE), PERIOD(TIME(n) [WITH TIME ZONE]), or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) value can be cast as TIMESTAMP[(n)] [WITH TIME ZONE] using

the CAST function. The source last value must be equal to the source beginning bound; otherwise, an error is reported.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]):

- If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision.
- If the target precision is lower than the source precision, an error is reported.

If the source type is PERIOD(DATE), the result is formed from the source beginning bound and a time portion of 0 adjusted with respect to the current session time zone, and, if the target type is WITH TIME ZONE, the current session time zone displacement.

If the source type is PERIOD(TIME(n) [WITH TIME ZONE]), the source beginning bound (in UTC) is adjusted with respect to the current session time zone displacement. The timestamp portion of the result is formed from CURRENT_DATE and the time portion of the source beginning bound obtained after the above adjustment. The resulting timestamp value is converted to UTC. If both the source and target type are WITH TIME ZONE, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is WITH TIME ZONE, the result time zone field is set to the current session time zone displacement.

If the source type is PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), the result timestamp portion is the timestamp portion of the source beginning bound. If both the source and target type are WITH TIME ZONE, the result time zone field is set to the time zone displacement of the source beginning bound. If only the target type is WITH TIME ZONE, the result time zone field is set to the current session time zone displacement.

Example

Assume pts is a PERIOD(TIMESTAMP(2)) column in table t with a value of PERIOD '(2005-02-03 12:12:12.34, 2005-02-03 12:12:12.35)'.

In the following example, column pts is cast as TIMESTAMP(6). The result is the source beginning bound with trailing zeros added to the fractional seconds to make the precision 6.

```
SELECT CAST(pts AS TIMESTAMP(6)) FROM t;
```

The following is returned:

```
2005-02-03 12:12:12.340000
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

Signed Zone DECIMAL Conversion

Introduction

Teradata SQL can convert input data that is in signed zone (external) DECIMAL format to a NUMERIC data type, thus allowing numeric operations to be performed on row values. The column in which the signed zone decimal data is to be stored may be any numeric data type.

A FORMAT phrase incorporating the S sign character filters the data as it passes in and out of Teradata Database.

The rightmost character of the input data string is assumed to contain the zone (overpunch) bit.

The following table shows the characters representing zone-numeric combinations.

Last Character (Input String)	Numeric Conversion	Last Character (Input String)	Numeric Conversion	Last Character (Input String)	Numeric Conversion
{	n … 0	}	-n … 0	0	n … 0
A	n … 1	J	-n … 1	1	n … 1
B	n … 2	K	-n … 2	2	n … 2
C	n … 3	L	-n … 3	3	n … 3
D	n … 4	M	-n … 4	4	n … 4
E	n … 5	N	-n … 5	5	n … 5
F	n … 6	O	-n … 6	6	n … 6
G	n … 7	P	-n … 7	7	n … 7
H	n … 8	Q	-n … 8	8	n … 8
I	n … 9	R	-n … 9	9	n … 9

The sign FORMAT phrase can be included in a CREATE TABLE or ALTER TABLE statement when the column is defined, or in the INSERT statement when the data is loaded. The chosen method depends on how the stored value is to be used.

When a sign FORMAT phrase is specified at column creation time, it is considered attached to the column because it translates data at the column level; that is, both when the data is loaded and when it is retrieved.

Using FORMAT in CREATE TABLE

When the FORMAT phrase is used in the CREATE TABLE statement, as follows:

```
CREATE TABLE Test1 (Col1 DECIMAL(4) FORMAT '9999S');
```

then zoned input character strings can be loaded with standard INSERT statements, whether the data is defined:

```
INSERT INTO Test1 (Col1) VALUES ('123J');
```

or read from a client system data record via the USING modifier:

```
USING Ext1 (CHAR(4))
INSERT INTO Test1 (Col1)
VALUES (:Ext1);
```

The data record contains the string '123J'.

Subsequently, a simple select, such as:

```
SELECT Col1 FROM Test1;
```

returns:

```
Col1
----

123J
```

Using Another FORMAT in the SELECT Statement

To override an attached format, another FORMAT phrase is needed in the retrieval statement. Using the preceding table, one of the two following statements must be used to retrieve the numeric value:

```
SELECT Col1 (FORMAT '+9999') FROM Test1;
```

or

```
SELECT CAST (Col1 AS INTEGER) FROM Test1;
```

The result is as follows.

```
Col1
-----

-1231
```

If FORMAT is Not Attached to the Column

If the format is not attached to the column, the sign FORMAT phrase must be used each time signed zoned decimal data is loaded and each time the row value is to be retrieved in signed zoned decimal format.

For example, if a table is defined using a CREATE TABLE statement like this:

```
CREATE TABLE Test2 (Col2 DECIMAL(5));
```

then the sign FORMAT phrase must be included whenever signed zoned decimal strings are inserted.

This is true whether the definition is explicitly defined, as it is in Examples 1 and 2, or defined implicitly by being read from a client system data record as it is in Examples 3 and 4.

Example 1

```
INSERT INTO Test2 (Col2)
VALUES ('5678B' (DECIMAL(5), FORMAT '99999S'));
```

Example 2

```
INSERT INTO Test2 (Col2)
VALUES ('9012L' (DECIMAL(5), FORMAT '99999S'));
```

Example 3

```
USING Ext2 (CHAR(5))
INSERT INTO Test2 (Col2)
VALUES (:Ext2 (DECIMAL(5), FORMAT '99999S'));
```

Example 4

```
USING Ext2 (CHAR(5))
INSERT INTO Test2 (Col2)
VALUES (:Ext2 (DECIMAL(5), FORMAT '99999S'));
```

where Ext2 contains the strings ‘5678B’ and ‘9012L’.

Because Col2 does not have an attached FORMAT phrase, a simple SELECT, such as the following example, returns the results as seen immediately following.

```
SELECT Col2 FROM Test2;

Col2
-------

 56782.
-90123.
```

A sign FORMAT phrase must be included in the SELECT statement in order to retrieve the values ‘5678B’ and ‘9012L’.

It is important to remember this rule when manipulating signed zoned decimal values, especially when using sophisticated facilities like subqueries.

Example 5

This example is based on the data from Example 4.

Consider a column created with a CHARACTER data type.

```
CREATE TABLE Test3 (Col3 CHAR(5));
```

The column is loaded by selecting, without a sign FORMAT phrase, values from an “unattached” column, as follows.

```
INSERT INTO Test3 (Col3)
SELECT Col2 FROM Test2 ;
```

The values that are inserted are the following:

```
Col3
-----

 5678
-9012
```

The sign FORMAT phrase *must* be included in the query specification in order to insert the values ‘5678B’ and ‘9012L’.

Related Topics

For information on data types, data type formats, formatting characters, and the FORMAT phrase, see *SQL Data Types and Literals*.

TIME-to-Character Conversion

Purpose

Convert TIME data to a character string.

CAST Syntax

where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be cast to a character type.
character_data_type	the character type to which the TIME expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting for the character representations of DateTime data.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be cast to a character type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character type to which the TIME expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.

ANSI Compliance

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

Usage Notes

When converting TIME to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the TIME value as represented by a character string literal.

IF the target data type is ...	AND *n* is ...	THEN ...
CHAR(*n*)	greater than the length of the TIME value as represented by a character string literal	trailing pad characters are added to pad the representation
	too small	a string truncation error is returned
VARCHAR(*n*)	greater than the length of the TIME value as represented by a character string literal	no blank padding is added to the character representation
	too small	a string truncation error is returned

TIME to CLOB conversion is not supported.

You cannot convert a TIME value to a character string when the server character set is GRAPHIC.

Forcing a FORMAT on CAST for Converting TIME to Character

The default format for TIME to character conversion is the format in effect for the TIME value.

You can convert a TIME value to a character string using a FORMAT phrase. The resulting format, however, is the same as the TIME value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from TIME to CHAR and force an explicit FORMAT on the result regardless of the format associated with the TIME value. This is because of the rules for matching FORMAT phrases to data types.

Example

Field T1 in the table INTTIME is a TIME(6) value with the explicit format 'HH:MI:SSDS(6)'. Assume that you want to convert this to a value of CHAR(6), and an explicit output format of 'HHhMIm'.

```
SELECT T1 FROM INTTIME ;
```

The result (without a type change) is the following report:

```
             T1
---------------
05:57:11.362271
```

Now use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```
SELECT
CAST( (CAST (T1 AS FORMAT 'HHhMim'))
AS CHAR(6))
FROM INTTIME;
```

The result after the nested CASTs is the following report.

```
T1
------
05h57m
```

The inner CAST establishes the display format for the TIME value and the outer CAST indicates the data type of the desired result.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIME-to-Period Conversion

Purpose

Converts TIME data as PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)][WITH TIME ZONE]).

CAST Syntax

```
── CAST ── ( time_expression ─ AS ── period_data_type ┬──────────────────┬── ) ──
                                                      └─ period_data_attribute ─┘
```

1101A610

where:

Syntax element ...	Specifies ...
time_expression	the TIME data expression to be converted.
period_data_type	the target Period type to which *time_expression* is to be converted.
period_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Usage Notes

A TIME(n) [WITH TIME ZONE] value can be cast as PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

If the target precision is higher than the source precision, trailing zeros are added in the result bounds to adjust the precision. If the target precision is lower than the source precision, an error is reported.

If the TIME source value contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.

If the target type is PERIOD(TIME[(n)] [WITH TIME ZONE]), the result beginning element is set to the source value (in UTC). If the target type is PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]), the source time value get adjusted with respect to the current session time zone displacement from the corresponding UTC value; the date portion in the result beginning element is set to CURRENT_DATE, the time portion is set to the source value obtained after the above adjustment, and the resulting timestamp value is converted to UTC. If both the source and target are WITH TIME ZONE, the time zone field of the result beginning element is set to the source time zone field. If only the target has WITH TIME ZONE, the time zone field of the result beginning element is set to the current session time zone displacement. The result ending element is set to the result beginning bound plus one granule of the target type. If the result ending bound has a lower value than the result beginning bound for a target type of PERIOD(TIME[(n)] [WITH TIME ZONE) or the result ending element value exceeds the maximum corresponding TIMESTAMP value for a target type of PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE), an error is reported.

Note: If the target type is WITH TIME ZONE, the result beginning and ending bounds have the same time zones.

Also, note that the result has the same value for the beginning bound and last value.

Example

Assume pt is a TIME(0) column in table t with a value of TIME '12:12:12' and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

In the following example, a TIME(0) column is cast as PERIOD(TIME(4) WITH TIME ZONE). The result beginning bound is formed form the source (in UTC) with trailing zeros added to make the precision 4 and the current session time zone displacement. The result ending element is set to the result beginning bound plus INTERVAL '0.0001' SECOND.

Note: The time zones of the result beginning and ending elements are the same.

```
SELECT CAST(pt AS PERIOD(TIME(4) WITH TIME ZONE)) FROM t;
```

Returns a PERIOD(TIME(4) WITH TIME ZONE) value as follows:

```
('12:12:12.0000-08:00', '12:12:12.0001-08:00')
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIME-to-TIME Conversion

Purpose

Converts TIME or TIME WITH TIME ZONE to TIME or TIME WITH TIME ZONE using optional data attributes.

CAST Syntax

where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used.

Syntax element ...	Specifies ...
AT SOURCE [TIME ZONE]	that the time zone associated with *time_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used for the CAST. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting for DateTime data.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIME (with or without time zone) to TIME WITH TIME ZONE. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIME (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIME value to UTC based on the current session time zone or on a specified time zone.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used.

Syntax element ...	Specifies ...
AT SOURCE [TIME ZONE]	that the time zone associated with *time_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement in the conversion. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used in the conversion. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata Conversion Syntax to convert from TIME (with or without time zone) to TIME WITH TIME ZONE. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIME (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIME value to UTC based on the current session time zone or on a specified time zone.

Usage Notes

If you specify an AT clause for a TIME[(n)] without time zone target data type, an error is returned.

If you specify an AT clause for a TIME[(n)] WITH TIME ZONE target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of *time_expression* is...	THEN...
AT LOCAL	with or without TIME ZONE	the result is formed from the source *time_expression* (in UTC) and the time zone displacement based on the current session time zone. If the data type of *time_expression* is without time zone, this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the result is formed from the time portion of the source *time_expression* (in UTC) and the time zone displacement associated with *time_expression*. Note that this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the result is formed from the time portion of the source *time_expression* (in UTC) and the time zone displacement associated with *time_expression*. Note that this is the same as not specifying the AT clause.
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT *expression* or AT TIME ZONE *expression*	with or without TIME ZONE	the result is formed from the time portion of the source *time_expression* (in UTC) and the time zone displacement defined by *expression*.
AT *time_zone_string* or AT TIME ZONE *time_zone_string*	with or without TIME ZONE	the result is formed from the time portion of the source *time_expression* (in UTC) and the time zone displacement based on *time_zone_string*. The time zone displacement is determined based on *time_zone_string*, CURRENT_TIMESTAMP AT '00:00', and the TIME value of *time_expression* at UTC.

Example 1

In this example, the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, is used to determine the UTC value, '07:30:00' of the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '07:30:00' at UTC and the current time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The result value of the CAST '07:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, and the result of the SELECT statements is: TIME '08:30:00+01:00'.

The result of the SELECT statements is equal to TIME '07:30:00+00:00' since values are compared based on their UTC values.

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(TIME '08:30:00' AS TIME(0) WITH TIME ZONE);
SELECT CAST(TIME '08:30:00' AS TIME(0) WITH TIME ZONE AT LOCAL);
```

Example 2

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '04:30:00' for the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '04:30:00' at UTC and the current session time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '20:30:00-08:00'.

The result of the SELECT statement is equal to TIME '04:30:00+00:00'.

```
SET TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;

SELECT CAST(TIME '08:30:00+04:00' AS TIME(0)
   WITH TIME ZONE AT LOCAL);
```

Example 3

The following SELECT statement returns an error because the source expression does not have a time zone displacement.

```
SELECT CAST(TIME '08:30:00' AS TIME(0)
   WITH TIME ZONE AT SOURCE TIME ZONE);
```

Example 4

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '04:30:00' for the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '04:30:00' at UTC, and the time zone displacement of the source expression, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the result of the SELECT statements is: TIME '08:30:00+04:00'.

The result of the SELECT statements is equal to TIME '04:30:00+00:00'. The current session time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, has no effect.

```
SET TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;

SELECT CAST(TIME '08:30:00+04:00' AS TIME(0) WITH TIME ZONE);
SELECT CAST(TIME '08:30:00+04:00' AS TIME(0)
   WITH TIME ZONE AT SOURCE);
```

Example 5

In this example, the current session time zone displacement, INTERVAL -'04:00' HOUR TO MINUTE, is used to determine the UTC value '12:30:00' for the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '12:30:00' at UTC, and the specified time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST '12:30:00' at UTC is adjusted to its time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '04:30:00-08:00'.

The result of the SELECT statement is equal to TIME '12:30:00+00:00'.

```
SET TIME ZONE INTERVAL -'04:00' HOUR TO MINUTE;

SELECT CAST(TIME '08:30:00' AS TIME(0) WITH TIME ZONE AT -8);
```

Example 6

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '04:30:00' for the TIME literal.

The result of the CAST is the time formed from the time portion of the source expression value '04:30:00' at UTC, and the specified time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '20:30:00-08:00'.

This result of the SELECT statement is equal to TIME '04:30:00+00:00'. The current session time zone displacement, INTERVAL '08:00' HOUR TO MINUTE, has no effect.

```
SET TIME ZONE INTERVAL '08:00' HOUR TO MINUTE;

SELECT CAST(TIME '08:30:00+04:00' AS TIME(0)
   WITH TIME ZONE AT -8);
```

Example 7

In this example, the current timestamp is:

```
             Current TimeStamp(6)
--------------------------------
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIME value '08:30:00' to a TIME WITH TIME ZONE value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIME '08:30:00' AS TIME(0) WITH TIME ZONE
   AT 'America Pacific');

      08:30:00
--------------
00:30:00-08:00
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIME-to-TIMESTAMP Conversion

Purpose

Converts TIME or TIME WITH TIME ZONE to TIMESTAMP or TIMESTAMP WITH TIME ZONE using optional data attributes.

CAST Syntax

where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used.

Syntax element ...	Specifies ...
AT SOURCE [TIME ZONE]	that the time zone associated with *time_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used for the CAST. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of the FORMAT phrase to enable alternative output formatting of DateTime data.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIME to TIMESTAMP. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIME (without time zone) and TIMESTAMP (without time zone) are not ANSI SQL:2011 compliant. Teradata Database internally converts a TIME or TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
time_expression	the TIME expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used.

Syntax element ...	Specifies ...
AT SOURCE [TIME ZONE]	that the time zone associated with *time_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement in the conversion. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used in the conversion. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata Conversion Syntax to convert from TIME to TIMESTAMP. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIME (without time zone) and TIMESTAMP (without time zone) are not ANSI SQL:2011 compliant. Teradata Database internally converts a TIME or TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Usage Notes

If you specify the AT clause for a TIMESTAMP[(n)] without time zone target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of *time_expression* is...	THEN...
AT LOCAL	with or without TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement based on the current session time zone. A local timestamp value is formed from CURRENT_DATE (at the above time zone displacement) and the time portion of *time_expression* obtained after the previous adjustment. The result is this local timestamp value adjusted to UTC by subtracting the above time zone displacement. This is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement of *time_expression*. A local timestamp value is formed from CURRENT_DATE (based on the time zone displacement of *time_expression)* and the time portion of *time_expression* obtained after the previous adjustment. The result is this local timestamp value adjusted to UTC by subtracting the time zone displacement of *time_expression*.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the source time_expression (in UTC) is adjusted by adding the time zone displacement of time_expression. A local timestamp value is formed from CURRENT_DATE (based on the time zone displacement of time_expression) and the time portion of time_expression obtained after the previous adjustment. The result is this local timestamp value adjusted to UTC by subtracting the time zone displacement of time_expression.
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT *expression* or AT TIME ZONE *expression*	with or without TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement defined by *expression.* A local timestamp value is formed from CURRENT_DATE at the above time zone displacement and the time portion of *time_expression* obtained after the above adjustment. The result is this local timestamp value adjusted to UTC by subtracting the above time zone displacement.

IF you specify...	AND the data type of *time_expression* is...	THEN...
AT *time_zone_string* or AT TIME ZONE *time_zone_string*	with or without TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement based on *time_zone_string*. The time zone displacement is determined based on *time_zone_string*, CURRENT_TIMESTAMP AT '00:00', and the TIME value of *time_expression* at UTC. A local timestamp value is formed from CURRENT_DATE at the above time zone displacement and the time portion of *time_expression* obtained after the above adjustment. The result is this local timestamp value adjusted to UTC by subtracting the above time zone displacement.

If you specify the AT clause for a TIMESTAMP[(n)] WITH TIME ZONE target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of *time_expression* is...	THEN...
AT LOCAL	with or without TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement based on the current session time zone. A local timestamp value is formed from CURRENT_DATE (at the above time zone displacement) and the time portion of *time_expression* obtained after the above adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the above time zone displacement. If the data type of *time_expression* is without time zone, this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement of *time_expression*. A local timestamp value is formed from CURRENT_DATE (based on the time zone displacement of *time_expression)* and the time portion of *time_expression* obtained after the previous adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the time zone displacement of *time_expression*.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.

IF you specify...	AND the data type of *time_expression* is...	THEN...
AT SOURCE TIME ZONE	WITH TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement of *time_expression*. A local timestamp value is formed from CURRENT_DATE (based on the time zone displacement of *time_expression*) and the time portion of *time_expression* obtained after the previous adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the time zone displacement of *time_expression*.
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT *expression* or AT TIME ZONE *expression*	with or without TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement defined by *expression*. A local timestamp value is formed from CURRENT_DATE (at the above time zone displacement) and the time portion of *time_expression* obtained after the above adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the above time zone displacement.
AT *time_zone_string* or AT TIME ZONE *time_zone_string*	with or without TIME ZONE	the source *time_expression* (in UTC) is adjusted by adding the time zone displacement based on *time_zone_string*. The time zone displacement is determined based on *time_zone_string*, CURRENT_TIMESTAMP AT '00:00', and the TIME value of *time_expression* at UTC. A local timestamp value is formed from CURRENT_DATE (at the above time zone displacement) and the time portion of *time_expression* obtained after the above adjustment. This resulting timestamp is adjusted to UTC, and the result value of the CAST at UTC is adjusted to the above time zone displacement.

Implicit TIME-to-TIMESTAMP Conversion

Teradata Database performs implicit conversion from TIME to TIMESTAMP data types in some cases. However, implicit conversion from TIME to TIMESTAMP is not supported for comparisons. See "Implicit Conversion of DateTime types" on page 582.

The following conversions are supported:

From source type...	To target type...
TIME	TIMESTAMP
	TIMESTAMP WITH TIME ZONE
TIME WITH TIME ZONE	TIMESTAMP
	TIMESTAMP WITH TIME ZONE

Example 1

Assuming the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, the following SELECT statements return the result: TIMESTAMP '2008-05-14 08:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;

SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0));
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT LOCAL);
```

The current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '23:30:00' of the literal.

For the CAST, the source expression value '23:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' TO MINUTE, to yield '08:30:00'. A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the time portion of the source expression value '08:30:00'. Then, this timestamp, '2008-05-14 08:30:00', at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 23:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 08:30:00'.

Example 2

Assuming the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, the following SELECT statements return the result: TIMESTAMP '2008-05-14 13:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;

SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0));
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) AT LOCAL);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE to yield '13:30:00'.

A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL HOUR '09:00' TO MINUTE, and the time portion of the source expression value '13:30:00'. Then this timestamp, '2008-05-14 13:30:00', at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 13:30:00'.

Example 3

An error is returned for the following SELECT statements because the source expression does not have a time zone.

```
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT SOURCE TIME ZONE);
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT SOURCE);
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE
   AT SOURCE TIME ZONE);
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE
   AT SOURCE);
```

Example 4

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '9:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-13 13:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0)
   AT SOURCE TIME ZONE);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the time zone displacement of the source, INTERVAL '04:00' HOUR TO MINUTE, to yield '08:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the time portion of the source expression value '08:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 08:30:00' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 04:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-13 13:30:00'.

Example 5

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone,

INTERVAL -'08:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-14 08:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT -8);
```

The current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '23:30:00' of the literal. For the CAST, the source expression value '23:30:00' at UTC is adjusted to the target time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, to yield '15:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the time portion of the source expression value '15:30:00' obtained after the above adjustment. Then this resulting timestamp '2008-05-13 15:30:00' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 23:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 08:30:00'.

Example 6

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-14 13:30:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) AT -8);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the target time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, to yield '20:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the time portion of the source expression value '20:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 20:30:00' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC.

The result value of the CAST at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 13:30:00'.

Example 7

Assuming the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, the following SELECT statements return the result: TIMESTAMP '2008-05-14 08:30:00+09:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE);
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE AT LOCAL);
```

The current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '23:30:00' of the literal. For the CAST, the source expression value '23:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, to yield '08:30:00'.

A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the time portion of the source expression value '08:30:00' obtained after the above adjustment. Then this timestamp '2008-05-14 08:30:00' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 23:30:00' at UTC with time zone displacement, INTERVAL '09:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statements is: TIMESTAMP '2008-05-14 08:30:00+09:00'.

Example 8

Assuming the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, the following SELECT statement returns the result: TIMESTAMP '2008-05-14 13:30:00+09:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0)
   WITH TIME ZONE AT LOCAL);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00 at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, to yield '13:30:00'.

A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the time portion of the source expression value '13:30:00' obtained after the above adjustment. Then this timestamp '2008-05-14 13:30:00' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC with time zone displacement, INTERVAL '09:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 13:30:00+09:00'.

Example 9

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone

displacement, INTERVAL '04:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-14 08:30:00+04:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) WITH TIME ZONE);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, to yield '13:30:00'.

A timestamp is formed from the current date '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the time portion of the source expression value '13:30:00' obtained after the above adjustment. Then this timestamp '2008-05-14 13:30:00' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC with time zone displacement, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '04:00' INTERVAL TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-14 08:30:00+04:00'.

Example 10

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-13 08:30:00+04:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) WITH TIME ZONE
   AT SOURCE);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the time zone displacement of the source expression, INTERVAL '04:00' HOUR TO MINUTE, to yield '08:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the time portion of the source expression value '08:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 08:30:00' at time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 04:30:00' at UTC with time zone displacement, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone, INTERVAL '04:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-13 08:30:00+04:00'. The current session time zone has no effect.

Example 11

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-13 15:30:00-08:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) WITH TIME ZONE AT -8);
```

The current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '23:30:00' of the literal. For the CAST, the source expression value '23:30:00' at UTC is adjusted to the target time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, to yield '15:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the time portion of the source expression value '15:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 15:30:00' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-13 23:30:00' at UTC with time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-13 15:30:00-08:00'.

Example 12

Assume that the current date is DATE '2008-05-14' at time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, but the current date is DATE '2008-05-13' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE. The following SELECT statement returns the result: TIMESTAMP '2008-05-13 20:30:00-08:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0) WITH TIME ZONE
   AT -8);
```

The time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, in the literal is used to determine the UTC value '04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal. For the CAST, the source expression value '04:30:00' at UTC is adjusted to the target time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, to yield '20:30:00'.

A timestamp is formed from the current date '2008-05-13' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the time portion of the source expression value '20:30:00' obtained after the above adjustment. Then this timestamp '2008-05-13 20:30:00' at time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, is adjusted to UTC so that the CAST result is '2008-05-14 04:30:00' at UTC with time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, so the result of the SELECT statement is: TIMESTAMP '2008-05-13 20:30:00-08:00'. The current session time zone has no effect.

Example 13

In this example, the current timestamp is:

```
             Current TimeStamp(6)
--------------------------------
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIME value '08:30:00' to a TIMESTAMP value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIME '08:30:00' AS TIMESTAMP(0) AT 'America Pacific');
```

The result of the query is:

```
           08:30:00
-------------------
2010-03-09 08:30:00
```

Example 14

In this example, the current timestamp is:

```
             Current TimeStamp(6)
--------------------------------
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIME value '08:30:00+04:00' to a TIMESTAMP value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIME '08:30:00+04:00' AS TIMESTAMP(0)
   AT 'America Pacific');
```

The result of the query is:

```
      08:30:00+04:00
-------------------
2010-03-10 04:30:00
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIME-to-UDT Conversion

Purpose

Converts TIME data to UDT data.

CAST Syntax

```
── CAST ── ( time_expression ── AS ── UDT_data_definition ) ──
```

1101A340

where:

Syntax element ...	Specifies ...
time_expression	a TIME expression to be cast to a UDT.
UDT_data_definition	the UDT type, followed by any optional FORMAT, NAMED, or TITLE data attribute phrases, to which *time_expression* is to be converted.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit TIME-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit TIME-to-UDT Conversion

Teradata Database performs implicit TIME-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

If no TIME-to-UDT implicit cast definition exists, Teradata Database looks for a CHAR-to-UDT or VARCHAR-to-UDT implicit cast definition that can substitute for the TIME-to-UDT implicit cast definition. Substitutions are valid because Teradata Database can implicitly cast a TIME type to the character data type, and then use the implicit cast definition to cast from the character data type to the UDT. If multiple character-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-Character Conversion

Purpose

Convert TIMESTAMP data to a character string.

CAST Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be cast to a character type.
character_data_type	the character type to which the TIMESTAMP expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.
character_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of character data attribute phrases.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be cast to a character type.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the character type to which the TIMESTAMP expression is to be converted.
server_character_set	the server character set to use for the conversion. If no CHARACTER SET clause is specified to indicate which server character set to use, the user default server character set is used.

ANSI Compliance

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

Usage Notes

When converting TIMESTAMP to CHAR(*n*) or VARCHAR(*n*), then *n* must be equal to or greater than the length of the TIMESTAMP value as represented by a character string literal.

IF the target data type is ...	AND *n* is ...	THEN ...
CHAR(*n*)	greater than the length of the TIMESTAMP value as represented by a character string literal	trailing pad characters are added to pad the representation.
	too small	a string truncation error is returned.

IF the target data type is ...	AND *n* is ...	THEN ...
VARCHAR(*n*)	greater than the length of the TIMESTAMP value as represented by a character string literal	no blank padding is added to the character representation.
	too small	a string truncation error is returned.

TIMESTAMP to CLOB conversion is not supported.

You cannot convert a TIME value to a character string if the server character set is GRAPHIC.

Forcing a FORMAT on CAST for Converting TIMESTAMP to Character

The default format for TIMESTAMP to character conversion is the format in effect for the TIMESTAMP value.

To override the format, you can convert a TIMESTAMP value to a string using a FORMAT phrase. The resulting format, however, is the same as the TIMESTAMP value. If you want a different format for the string value, you need to also use CAST as described here.

You must use nested CAST operations in order to convert values from TIMESTAMP to CHAR and force an explicit FORMAT on the result regardless of the format associated with the TIMESTAMP value. This is because of the rules for matching FORMAT phrases to data types.

Example

Field TS1 in the table INTTIMESTAMP is a TIMESTAMP value with the explicit format 'Y4-MM-DDBHH:MI:SSDS(6)'. Assume that you want to convert this to a value of CHAR(19), and an explicit output format of 'M3BDD,BY4BHHhMIm'.

```
SELECT TS1 FROM INTTIMESTAMP;
```

The result (without a type change) is the following report:

```
                        TS1
--------------------------
1900-12-31 08:25:37.899231
```

Now use nested CAST phrases and a FORMAT to obtain the desired result: a report in character format.

```
SELECT
CAST( (CAST (TS1 AS FORMAT 'M3BDD,BY4BHHhMIm'))
AS CHAR(19))
FROM INTTIMESTAMP;
```

The result after the nested CASTs is the following report.

```
TS1
-------------------
Dec 31, 1900 08h25m
```

The inner CAST establishes the display format for the TIMESTAMP value and the outer CAST indicates the data type of the desired result.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-DATE Conversion

Purpose

Convert TIMESTAMP data to a DATE value.

CAST Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted. *timestamp_expression* may include an AT clause.
AT LOCAL	that the time zone displacement based on the current session time zone is used. This is the default.
AT SOURCE [TIME ZONE]	that the time zone associated with *timestamp_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.

Syntax element ...	Specifies ...
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used for the CAST. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
date_data_attribute	any of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of DATE data attribute phrases, such as FORMAT that enables an alternative format.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIMESTAMP to DATE. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted. *timestamp_expression* may include an AT clause.
data_attribute	any of the following optional data attributes: • FORMAT • NAMED • TITLE
AT LOCAL	that the time zone displacement based on the current session time zone is used. This is the default.
AT SOURCE [TIME ZONE]	that the time zone associated with *timestamp_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement in the conversion. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used in the conversion. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata Conversion Syntax to convert from TIMESTAMP to DATE. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Usage Notes

The following table shows the result of the CAST function or Teradata conversion based on the various options specified. Note that the time zone adjustment may change the YEAR, MONTH, and DAY fields of the DATE value.

IF you specify...	AND the data type of *timestamp_expression* is...	THEN...
AT LOCAL	with or without TIME ZONE	the result is the date portion of the source *timestamp_expression* after adjusting its UTC value by adding the time zone displacement based on the current session time zone. This is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the result is the date portion of the source *timestamp_expression* after adjusting its UTC value by adding the time zone displacement associated with *timestamp_expression*.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the result is the date portion of the source *timestamp_expression* after adjusting its UTC value by adding the time zone displacement associated with *timestamp_expression*.
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT *expression* or AT TIME ZONE *expression*	with or without TIME ZONE	the result is the date portion of the source *timestamp_expression* after adjusting its UTC value by adding the time zone displacement defined by *expression*.
AT *time_zone_string* or AT TIME ZONE *time_zone_string*	with or without TIME ZONE	the result is the date portion of the source *timestamp_expression* after adjusting its UTC value by adding the time zone displacement based on *time_zone_string*. The time zone displacement is determined based on *time_zone_string* and the TIMESTAMP value of *timestamp_expression* at UTC.

Implicit TIMESTAMP-to-DATE Conversion

Teradata Database performs implicit conversion from TIMESTAMP types to DATE in some cases. See "Implicit Conversion of DateTime types" on page 582.

The following conversions are supported:

From source type...	To target type...
TIMESTAMP	DATE[a]
TIMESTAMP WITH TIME ZONE	

a. ANSIDate dateform mode or IntegerDate dateform mode

The TIMESTAMP value is always converted to DATE in case of comparison.

Example 1

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.

Example 2

Consider the following statements:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-05-31 22:30:00-08:00'
   AS DATE AT SOURCE TIME ZONE);

SELECT TIMESTAMP '2008-06-01 06:30:00+00:00' AT '-08:00'
   (DATE, AT SOURCE);

SELECT TIMESTAMP '2008-06-01 06:30:00+00:00' (DATE, AT -8);
```

```
SELECT TIMESTAMP '2008-06-01 07:30:00' (DATE, AT -8);
```

These SELECT statements return the date for time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE; that is, the statements return '08/05/31'. If the SELECT statements were specified without an AT clause or with an AT LOCAL clause, these statements would return '08/06/01' for the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The following shows the results of the SELECT statements if the AT clause was not specified:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-05-31 22:30:00-08:00' AS DATE);

2008-05-31 22:30:00-08:00
-------------------------
                 08/06/01

SELECT TIMESTAMP '2008-06-01 06:30:00+00:00'
   AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;

2008-06-01 06:30:00+00:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
---------------------------------------------------------------------
                                          2008-05-31 22:30:00-08:00

SELECT TIMESTAMP '2008-06-01 06:30:00+00:00'
   AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE (DATE);

2008-06-01 06:30:00+00:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
---------------------------------------------------------------------
                                                            08/06/01

SELECT TIMESTAMP '2008-06-01 06:30:00+00:00' (DATE);

2008-06-01 06:30:00+00:00
-------------------------
                 08/06/01

SELECT TIMESTAMP '2008-06-01 07:30:00' (DATE);

2008-06-01 07:30:00
-------------------
           08/06/01
```

The following shows the results of the SELECT statements if the AT clause was not specified, and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

```
SET TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-05-31 22:30:00-08:00' AS DATE);

2008-05-31 22:30:00-08:00
-------------------------
                 08/05/31

SELECT TIMESTAMP '2008-06-01 06:30:00+00:00'
   AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE (DATE);

2008-06-01 06:30:00+00:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
```

```
----------------------------------------------------------------------
                                                                08/05/31

SELECT TIMESTAMP '2008-06-01 06:30:00+00:00' (DATE);

2008-06-01 06:30:00+00:00
-------------------------
                 08/05/31

SELECT CAST(TIMESTAMP '2008-06-01 07:30:00+01:00'
   AS TIMESTAMP(0)) (DATE);

2008-06-01 07:30:00+01:00
-------------------------
                 08/05/31
```

Example 3

Consider the following statements:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-02 04:30:00+09:00'
   AS DATE AT SOURCE TIME ZONE);

SELECT TIMESTAMP '2008-06-01 20:30:00+01:00'
   AT TIME ZONE INTERVAL '09' HOUR (DATE, AT SOURCE);

SELECT TIMESTAMP '2008-06-01 20:30:00' (DATE, AT +9);
```

These SELECT statements return the date for time zone displacement, INTERVAL '09:00' HOUR TO MINUTE; that is, the statements return '08/06/02'. If the SELECT statements were specified without an AT clause or with an AT LOCAL clause, these statements would return '08/06/01' for the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The following shows the results of the SELECT statements if the AT clause was not specified:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-02 04:30:00+09:00' AS DATE);

2008-06-02 04:30:00+09:00
-------------------------
                 08/06/01

SELECT TIMESTAMP '2008-06-01 20:30:00+01:00'
   AT TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;

2008-06-01 20:30:00+01:00 AT TIME ZONE INTERVAL  9:00 HOUR TO MINUTE
----------------------------------------------------------------------
                                          2008-06-02 04:30:00+09:00

SELECT TIMESTAMP '2008-06-01 20:30:00+01:00'
   AT TIME ZONE INTERVAL '09:00' HOUR TO MINUTE (DATE);

2008-06-01 20:30:00+01:00 AT TIME ZONE INTERVAL  9:00 HOUR TO MINUTE
----------------------------------------------------------------------
```

```
                                                                  08/06/01

SELECT TIMESTAMP '2008-06-01 20:30:00' (DATE);

2008-06-01 20:30:00
-------------------
           08/06/01
```

The following shows the results of the SELECT statements if the AT clause was not specified, and the current session time zone displacement is INTERVAL '09:00' TO MINUTE.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-02 04:30:00+09:00' AS DATE);

2008-06-02 04:30:00+09:00
-------------------------
                 08/06/02

SELECT TIMESTAMP '2008-06-01 20:30:00+01:00'
   AT TIME ZONE INTERVAL '09:00' HOUR TO MINUTE (DATE);

2008-06-01 20:30:00+01:00 AT TIME ZONE INTERVAL  9:00 HOUR TO MINUTE
---------------------------------------------------------------------
                                                            08/06/02

SELECT CAST(TIMESTAMP '2008-06-01 20:30:00+01:00'
   AS TIMESTAMP(0)) (DATE);

2008-06-01 20:30:00+01:00
-------------------------
                 08/06/02
```

Example 4

Consider the following statements:

```
SET TIME ZONE INTERVAL '10:00' HOUR TO MINUTE;

SELECT CAST((TIMESTAMP '2008-06-01 18:30:00+01:00' AT '05:45')
   AS DATE AT SOURCE);

SELECT CAST((TIMESTAMP '2008-06-01 18:30:00+01:00' AT 5.75)
   AS DATE AT SOURCE);

SELECT TIMESTAMP '2008-06-01 23:15:00+05:45'
   (DATE, AT SOURCE TIME ZONE);

SELECT TIMESTAMP '2008-06-02 03:30:00' (DATE, AT '05:45');
SELECT TIMESTAMP '2008-06-02 03:30:00' (DATE, AT 5.75);
```

These SELECT statements return the date for time zone displacement, INTERVAL '05:45' HOUR TO MINUTE; that is, the statements return '08/06/01'. If the SELECT statements were specified without an AT clause or with an AT LOCAL clause, these statements would return '08/06/02' for the current session time zone displacement, INTERVAL '10:00' HOUR TO MINUTE.

The following shows the results of the SELECT statements if the AT clause was not specified:

```
SET TIME ZONE INTERVAL '10:00' HOUR TO MINUTE;

SELECT TIMESTAMP '2008-06-01 18:30:00+01:00'
   AT TIME ZONE INTERVAL '05:45' HOUR TO MINUTE;

2008-06-01 18:30:00+01:00 AT TIME ZONE INTERVAL  5:45 HOUR TO MINUTE
--------------------------------------------------------------------
                                          2008-06-01 23:15:00+05:45

SELECT CAST((TIMESTAMP '2008-06-01 18:30:00+01:00'
   AT TIME ZONE INTERVAL '05:45' HOUR TO MINUTE) AS DATE);

2008-06-01 18:30:00+01:00 AT TIME ZONE INTERVAL  5:45 HOUR TO MINUTE
--------------------------------------------------------------------
                                                             08/06/02

SELECT TIMESTAMP '2008-06-01 23:15:00+05:45' (DATE);

2008-06-01 23:15:00+05:45
-------------------------
                 08/06/02

SELECT TIMESTAMP '2008-06-02 03:30:00' (DATE);

2008-06-02 03:30:00
-------------------
           08/06/02
```

The following shows the results of the SELECT statements if the AT clause was not specified, and the current session time zone displacement is INTERVAL '05:45' HOUR TO MINUTE.

```
SET TIME ZONE INTERVAL '05:45' HOUR TO MINUTE;

SELECT CAST((TIMESTAMP '2008-06-01 18:30:00+01:00'
   AT TIME ZONE INTERVAL'05:45' HOUR TO MINUTE) AS DATE);

2008-06-01 18:30:00+01:00 AT TIME ZONE INTERVAL  5:45 HOUR TO MINUTE
--------------------------------------------------------------------
                                                             08/06/01

SELECT TIMESTAMP '2008-06-01 23:15:00+05:45' (DATE);

2008-06-01 23:15:00+05:45
-------------------------
                 08/06/01

SELECT CAST(TIMESTAMP '2008-06-02 03:30:00+10:00'
   AS TIMESTAMP(0)) (DATE);

2008-06-02 03:30:00+10:00
-------------------------
                 08/06/01
```

Example 5

Consider the following statements:

```
SET TIME ZONE +1;
SELECT CAST((TIMESTAMP '2008-06-01 08:30:00' AT TIME ZONE -8)
```

```
   AS DATE AT SOURCE TIME ZONE);
```

This SELECT statement returns the date for time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE; that is, the statement returns '08/05/31'. If the SELECT statement was specified without an AT clause or with an AT LOCAL clause, the statement would return '08/06/01' for the current session time zone displacement, INTERVAL HOUR '01:00' MINUTE.

The following shows the result of the SELECT statement if the AT clause was not specified:

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT TIMESTAMP '2008-06-01 08:30:00'
   AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;

2008-06-01 08:30:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
--------------------------------------------------------------
                                    2008-05-31 23:30:00-08:00

SELECT CAST((TIMESTAMP '2008-06-01 08:30:00'
   AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE) AS DATE);

2008-06-01 08:30:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
--------------------------------------------------------------
                                                      08/06/01
```

The following shows the result of the SELECT statement if the AT clause was not specified, and the current session time zone displacement is INTERVAL -'08:00' HOUR TO MINUTE.

```
SET TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;

SELECT CAST((CAST(TIMESTAMP '2008-06-01 08:30:00+01:00'
   AS TIMESTAMP(0)) AT TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE)
   AS DATE);

2008-06-01 08:30:00+01:00 AT TIME ZONE INTERVAL -8:00 HOUR TO MINUTE
---------------------------------------------------------------------
                                                           08/05/31
```

Example 6

In this example, the current timestamp is:

```
             Current TimeStamp(6)
--------------------------------
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIMESTAMP value '2010-03-09 22:30:00-08:00' to a DATE value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIMESTAMP '2010-03-09 22:30:00-08:00' AS DATE
   AT 'America Pacific');
```

The result of the query is:

```
2010-03-09 22:30:00-08:00
-------------------------
                 10/03/09
```

Example 7

The following SELECT statements return an error because the source expression does not have a TIMESTAMP WITH TIME ZONE data type.

```
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS DATE AT SOURCE);
SELECT CAST(TIME '08:30:00+03:00' AS DATE AT SOURCE TIME ZONE);
SELECT CAST(TIME '08:30:00' AS DATE AT SOURCE);
SELECT CAST(DATE '2008-06-01' AS DATE AT SOURCE TIME ZONE);
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-Period Conversion

Purpose

Converts a TIMESTAMP value as PERIOD(DATE), PERIOD(TIME[(n)][WITH TIME ZONE]), or PERIOD(TIMESTAMP[(n)][WITH TIME ZONE]).

CAST Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP data expression to be converted.
period_data_type	the target Period type to which *timestamp_expression* is to be converted.
period_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases.

Usage Notes

A TIMESTAMP(n) [WITH TIME ZONE] value can be cast as PERIOD(DATE), PERIOD(TIME[(n)] [WITH TIME ZONE]), or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) using the CAST function.

If the target type is PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]):

- If the target precision is higher than the source precision, trailing zeros are added in the result bounds to adjust the precision.
- If the target precision is lower than the source precision, an error is reported.

If the target type is PERIOD(DATE), the result beginning bound is the date portion of the source beginning bound adjusted to the current session time zone.

If the target type is PERIOD(TIME[(n)]), the result beginning bound is the time portion of the source value (in UTC).

If the target type is PERIOD(TIME[(n)] WITH TIME ZONE), the result beginning bound is formed from the time portion of the source value (in UTC) and, if the source type is WITH TIME ZONE, the source time zone displacement and, if not, the current session time zone displacement.

If the target type is PERIOD(TIMESTAMP[(n)]), the result beginning bound is the timestamp portion of the source value (in UTC).

If the target type is PERIOD(TIMESTAMP[(n)] WITH TIME ZONE), the result beginning bound is formed from the timestamp portion of the source value (in UTC) and, if the source type is WITH TIME ZONE, the source time zone displacement and, if not, the current session time zone displacement.

If the TIMESTAMP source value contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the target precision.

The result ending element is set to the result beginning bound plus one granule of the target type. If the result ending bound exceeds the maximum allowed DATE or TIMESTAMP value for a target type of PERIOD(DATE) or PERIOD(TIMESTAMP[(n)]), respectively, or the ending bound has a lower value than the result beginning bound in their UTC forms for a target type of PERIOD(TIME[(n)]), an error is reported.

Note: If the target type is WITH TIME ZONE, the result beginning and ending bounds have the same time zones.

Also, note that the result has the same value for the beginning bound and last value.

Example

In the following example, a TIMESTAMP(6) literal is cast as PERIOD(DATE). The result beginning element is set to the date portion of the source value. The result ending element is set to result beginning bound plus INTERVAL '1' DAY.

```
SELECT CAST(TIMESTAMP '2005-02-03 12:12:12.340000' AS PERIOD(DATE));
```

The following is returned:

```
('2005-02-03', '2005-02-04')
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-TIME Conversion

Purpose

Convert TIMESTAMP data to a TIME value.

CAST Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used.

Syntax element ...	Specifies ...
AT SOURCE [TIME ZONE]	that the time zone associated with *timestamp_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used for the CAST. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of TIME data attribute phrases.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIMESTAMP to TIME. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIME (without time zone) and TIMESTAMP (without time zone) are not ANSI SQL:2011 compliant. Teradata Database internally converts a TIME or TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used.

Syntax element ...	Specifies ...
AT SOURCE [TIME ZONE]	that the time zone associated with *timestamp_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement in the conversion. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used in the conversion. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata Conversion Syntax to convert from TIMESTAMP to TIME. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIME (without time zone) and TIMESTAMP (without time zone) are not ANSI SQL:2011 compliant. Teradata Database internally converts a TIME or TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Usage Notes

If you specify an AT clause for a TIME[(n)] without time zone target data type, an error is returned.

If you specify an AT clause for a TIME[(n)] WITH TIME ZONE target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of *timestamp_expression* is...	THEN...
AT LOCAL	with or without TIME ZONE	the result is formed from the source *timestamp_expression* (in UTC) and the time zone displacement based on the current session time zone. If the data type of *timestamp_expression* is without time zone, this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the result is formed from the time portion of the source *timestamp_expression* (in UTC) and the time zone displacement associated with *timestamp_expression*. Note that this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the result is formed from the time portion of the source *timestamp_expression* (in UTC) and the time zone displacement associated with *timestamp_expression*. Note that this is the same as not specifying the AT clause.
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT *expression* or AT TIME ZONE *expression*	with or without TIME ZONE	the result is formed from the time portion of the source *timestamp_expression* (in UTC) and the time zone displacement defined by *expression*.
AT *time_zone_string* or AT TIME ZONE *time_zone_string*	with or without TIME ZONE	the result is formed from the time portion of the source *timestamp_expression* (in UTC) and the time zone displacement based on *time_zone_string*. The time zone displacement is determined based on *time_zone_string* and the TIMESTAMP value of *timestamp_expression* at UTC.

Implicit TIMESTAMP-to-TIME Conversion

Teradata Database performs implicit conversion from TIMESTAMP to TIME data types in some cases. However, implicit conversion from TIMESTAMP to TIME is not supported for comparisons. See "Implicit Conversion of DateTime types" on page 582.

The following conversions are supported:

From source type...	To target type...
TIMESTAMP	TIME
	TIME WITH TIME ZONE

From source type...	To target type...
TIMESTAMP WITH TIME ZONE	TIME
	TIME WITH TIME ZONE

Example 1

In this example, the current session time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, is used to determine the UTC value, '2008-06-01 07:30:00', of the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 07:30:00' at UTC and the current time zone displacement, INTERVAL '01:00' HOUR TO MINUTE.

The result value of the CAST '07:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '01:00' HOUR TO MINUTE, and the result of the SELECT statements is: TIME '08:30:00+01:00'.

The result of the SELECT statements is equal to TIME '07:30:00+00:00' since values are compared based on their UTC values.

```
SET TIME ZONE INTERVAL '01:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIME(0)
   WITH TIME ZONE);

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIME(0)
   WITH TIME ZONE AT LOCAL);
```

Example 2

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' for the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 04:30:00' at UTC and the current session time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '20:30:00-08:00'.

The result of the SELECT statement is equal to TIME '04:30:00+00:00'.

```
SET TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
   AS TIME(0) WITH TIME ZONE AT LOCAL);
```

Example 3

The following SELECT statement return an error because the source expression does not have a time zone displacement.

```
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00'
   AS TIME(0) WITH TIME ZONE AT SOURCE);
```

Example 4

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' for the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 04:30:00' at UTC, and the time zone displacement of the source expression, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the result of the SELECT statements is: TIME '08:30:00+04:00'.

The result of the SELECT statements is equal to TIME '04:30:00+00:00'. The current session time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, has no effect.

```
SET TIME ZONE INTERVAL -'08:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
   AS TIME(0) WITH TIME ZONE);

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
   AS TIME(0) WITH TIME ZONE AT SOURCE TIME ZONE);
```

Example 5

In this example, the current session time zone displacement, INTERVAL -'04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 12:30:00' for the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 12:30:00' at UTC, and the specified time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST '12:30:00' at UTC is adjusted to its time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '04:30:00-08:00'.

The result of the SELECT statement is equal to TIME '12:30:00+00:00'.

```
SET TIME ZONE INTERVAL -'04:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00'
   AS TIME(0) WITH TIME ZONE AT -8);
```

Example 6

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' for the TIMESTAMP literal.

The result of the CAST is the time formed from the time portion of the source expression value '2008-06-01 04:30:00' at UTC, and the specified time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST '04:30:00' at UTC is adjusted to its time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIME '20:30:00-08:00'.

This result of the SELECT statement is equal to TIME '04:30:00+00:00'. The current session time zone displacement, INTERVAL '08:00' HOUR TO MINUTE, has no effect.

```
SET TIME ZONE INTERVAL '08:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
   AS TIME(0) WITH TIME ZONE AT -8);
```

Example 7

In this example, the current timestamp is:

```
             Current TimeStamp(6)
--------------------------------
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIMESTAMP value '2010-03-09 08:30:00' to a TIME WITH TIME ZONE value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIMESTAMP '2010-03-09 08:30:00' AS TIME(0) WITH TIME ZONE
   AT 'America Pacific');
```

The result of the query is:

```
2010-03-09 08:30:00
-------------------
     00:30:00-08:00
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-TIMESTAMP Conversion

Purpose

Convert TIMESTAMP data to a TIMESTAMP value with different precision information or WITH TIME ZONE definition.

CAST Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used.

Syntax element ...	Specifies ...
AT SOURCE [TIME ZONE]	that the time zone associated with *timestamp_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement for the CAST. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used for the CAST. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 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.

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using CAST to convert from TIMESTAMP to TIMESTAMP. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
timestamp_expression	the TIMESTAMP expression to be converted.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
AT LOCAL	that the time zone displacement based on the current session time zone is used.

Syntax element ...	Specifies ...
AT SOURCE [TIME ZONE]	that the time zone associated with *timestamp_expression* is used in the following cases: • AT SOURCE TIME ZONE is specified. • AT SOURCE is specified without TIME ZONE and there is no column named *source* in the scope. Otherwise, if AT SOURCE is specified without TIME ZONE and a column named *source* exists, then SOURCE references this column, and the value of the column is used as the time zone displacement in the conversion. If needed, the column value is implicitly converted to type INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If there are multiple columns named *source* in the scope, an error is returned.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement used in the conversion. For details, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823.

ANSI Compliance

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

The AT clause is ANSI SQL:2011 compliant.

As an extension to ANSI, the AT clause is supported when using Teradata Conversion Syntax to convert from TIMESTAMP to TIMESTAMP. In addition, you can specify the time zone displacement using additional expressions besides an INTERVAL expression.

Note: TIMESTAMP (without time zone) is not ANSI SQL:2011 compliant. Teradata Database internally converts a TIMESTAMP value to UTC based on the current session time zone or on a specified time zone.

Usage Notes

If you specify an AT clause for a TIMESTAMP[(n)] without time zone target data type, an error is returned.

If you specify an AT clause for a TIMESTAMP[(n)] WITH TIME ZONE target data type, the following table shows the result of the CAST function or Teradata conversion based on the various options specified. If the target precision is higher than the source precision, trailing zeros are added in the result to adjust the precision. If the target precision is lower than the source precision, an error is returned.

IF you specify...	AND the data type of *timestamp_expression* is...	THEN...
AT LOCAL	with or without TIME ZONE	the result is formed from the timestamp portion of the source *timestamp_expression* (in UTC) with the result time zone displacement based on the current session time zone. If the source data type is without time zone, this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	WITH TIME ZONE	the result is formed from the timestamp portion of the source *timestamp_expression* (in UTC) and the time zone displacement associated with *timestamp_expression*. Note that this is the same as not specifying the AT clause.
AT SOURCE (where SOURCE is a keyword and not a column reference)	without TIME ZONE	an error is returned.
AT SOURCE TIME ZONE	WITH TIME ZONE	the result is formed from the timestamp portion of the source *timestamp_expression* (in UTC) and the time zone displacement associated with *timestamp_expression*. Note that this is the same as not specifying the AT clause.
AT SOURCE TIME ZONE	without TIME ZONE	an error is returned.
AT *expression* or AT TIME ZONE *expression*	with or without TIME ZONE	the result is formed from the timestamp portion of the source *timestamp_expression* (in UTC) and the time zone displacement defined by *expression*.
AT *time_zone_string* or AT TIME ZONE *time_zone_string*	with or without TIME ZONE	the result is formed from the timestamp portion of the source *timestamp_expression* (in UTC) and the time zone displacement based on *time_zone_string*. The time zone displacement is determined based on *time_zone_string* and the TIMESTAMP value of *timestamp_expression* at UTC.

Example 1

The following SELECT statements return an error because the target data type does not have a TIMESTAMP WITH TIME ZONE data type.

```
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIMESTAMP(0)
   AT LOCAL);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+01:00' AS TIMESTAMP(0)
   AT LOCAL);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIMESTAMP(0)
   AT SOURCE TIME ZONE);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+01:00' AS TIMESTAMP(0)
```

```
   AT SOURCE);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIMESTAMP(0) AT +3);
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+01:00' AS TIMESTAMP(0)
   AT -6);
```

Example 2

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal.

The CAST result is the source expression value '2008-06-01 04:30:00' at UTC with the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIMESTAMP '2008-06-01 13:30:00+09:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
   AS TIMESTAMP(0) WITH TIME ZONE AT LOCAL);
```

Example 3

The following SELECT statements return an error because the source expression does not have a time zone displacement.

```
SELECT CAST(TIMESTAMP '2008-06-01 08:30:00'
   AS TIMESTAMP(0) WITH TIME ZONE AT SOURCE TIME ZONE);

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00'
   AS TIMESTAMP(0) WITH TIME ZONE AT SOURCE);
```

Example 4

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal.

The CAST result is source expression value '2008-06-01 04:30:00' at UTC with its time zone displacement, INTERVAL '04:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, and the result of the SELECT is: TIMESTAMP '2008-06-01 08:30:00+04:00'. The current session time zone has no effect.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
   AS TIMESTAMP(0) WITH TIME ZONE);

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
   AS TIMESTAMP(0) WITH TIME ZONE AT SOURCE TIME ZONE);
```

Example 5

In this example, the current session time zone displacement, INTERVAL '09:00' HOUR TO MINUTE, is used to determine the UTC value '2008-05-31 23:30:00' of the literal.

The CAST result is the source expression value '2008-05-31 23:30:00' at UTC with the target time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIMESTAMP '2008-05-31 15:30:00-08:00'.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00' AS TIMESTAMP(0)
   WITH TIME ZONE AT -8);
```

Example 6

In this example, the time zone displacement specified in the literal, INTERVAL '04:00' HOUR TO MINUTE, is used to determine the UTC value '2008-06-01 04:30:00' and time zone displacement, INTERVAL '04:00' HOUR TO MINUTE, of the literal.

The CAST result is the source expression value '2008-06-01 04:30:00' at UTC with the target time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE.

The result value of the CAST at UTC is adjusted to time zone displacement, INTERVAL -'08:00' HOUR TO MINUTE, and the result of the SELECT statement is: TIMESTAMP '2008-05-31 20:30:00-08:00'. The current session time zone has no effect.

```
SET TIME ZONE INTERVAL '09:00' HOUR TO MINUTE;

SELECT CAST(TIMESTAMP '2008-06-01 08:30:00+04:00'
   AS TIMESTAMP(0) WITH TIME ZONE AT -8);
```

Example 7

In this example, the current timestamp is:

```
             Current TimeStamp(6)
--------------------------------
2010-03-09 19:23:27.620000+00:00
```

The following statement converts the TIMESTAMP value '2010-03-09 08:30:00' to a TIMESTAMP WITH TIME ZONE value, where the time zone displacement is based on the time zone string, 'America Pacific'.

```
SELECT CAST(TIMESTAMP '2010-03-09 08:30:00' AS TIMESTAMP(0)
   WITH TIME ZONE AT 'America Pacific');
```

The result of the query is:

```
       2010-03-09 08:30:00
-------------------------
2010-03-09 00:30:00-08:00
```

Example 8

In this example, the tswz column in table1 contains the following timestamp data:

```
SELECT * FROM table1;
```

The result of the query is:

```
                                tswz
-----------------------------------
   2011-11-04 13:14:00.860000-07:00
```

You can use an AT clause with a time zone string in a SELECT query to get the same time in a different time zone. For example:

```
SELECT CAST(tswz AS TIMESTAMP WITH TIME ZONE AT 'gmt')
FROM table1;
```

The result of the query is:

```
                                tswz
-----------------------------------
   2011-11-04 20:14:00.860000+00:00
```

Similarly, you can use an AT clause with a time zone displacement to get the same result.

```
SELECT CAST(tswz AS TIMESTAMP WITH TIME ZONE AT TIME ZONE '00:00')
FROM table1;
```

The result of the query is:

```
                                tswz
-----------------------------------
   2011-11-04 20:14:00.860000+00:00
```

The following query produces the same result without an explicit CAST:

```
SELECT tswz AT 'gmt' FROM table1;
```

The result of the query is:

```
            tswz AT TIME ZONE 'gmt'
-----------------------------------
   2011-11-04 20:14:00.860000+00:00
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

TIMESTAMP-to-UDT Conversion

Purpose

Converts TIMESTAMP data to UDT data.

CAST Syntax

```
── CAST ── ( timestamp_expression ── AS ── UDT_data_definition ) ──
```

1101A341

where:

Syntax element ...	Specifies ...
timestamp_expression	a TIMESTAMP expression to be cast to a UDT.
UDT_data_definition	the UDT type, followed by any optional FORMAT, NAMED, or TITLE data attribute phrases, to which *timestamp_expression* is to be converted.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit TIMESTAMP-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit TIMESTAMP-to-UDT Conversion

Teradata Database performs implicit TIMESTAMP-to-UDT conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

If no TIMESTAMP-to-UDT implicit cast definition exists, Teradata Database looks for a CHAR-to-UDT or VARCHAR-to-UDT implicit cast definition that can substitute. Substitutions are valid because Teradata Database can implicitly cast a TIMESTAMP type to the character data type, and then use the implicit cast definition to cast from the character data type to the UDT. If multiple character-to-UDT implicit cast definitions exist, then Teradata Database returns an SQL error.

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-Byte Conversion

Purpose

Converts a UDT expression to a byte data type.

CAST Syntax

```
── CAST ── ( UDT_expression ── AS ── byte_data_definition ── ) ──
```

1101A344

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
byte_data_definition	the BLOB, BYTE or VARBYTE byte type followed by optional FORMAT, NAMED, or TITLE attribute phrases to which *UDT_expression* is to be converted.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax

```
─ UDT_expression ─ ( ┌──────────────────────┐ byte_data_type ┌──────────────────────┐ ) ──────
                    └─ data_attribute , ─┘                   └─ , data_attribute ─┘
```

1101B345

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
byte_data_type	the BLOB, BYTE or VARBYTE byte type to which UDT_expression is to be converted.

ANSI Compliance

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

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-byte conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit UDT-to-byte data type conversion requires a cast definition (see "Usage Notes") that specifies the following:

- the AS ASSIGNMENT clause
- a BYTE, VARBYTE, or BLOB target data type

  The target data type of the cast definition does not have to be an exact match to the target of the implicit type conversion.

If multiple implicit cast definitions exist for converting the UDT to different byte types, Teradata Database uses the implicit cast definition for the byte type with the highest precedence. The following list shows the precedence of byte types in order from lowest to highest precedence:

- BYTE
- VARBYTE
- BLOB

Example

Consider the following table definition, where image is a UDT:

```
CREATE TABLE history
   (id INTEGER
   ,information image );
```

Assuming an appropriate cast definition exists for the image UDT, the following statement converts the values in the information column to BYTE:

```
SELECT CAST (information AS BYTE(20))
FROM history
WHERE id = 100121;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-Character Conversion

Purpose

Converts a UDT expression to a character data type.

CAST Syntax

```
── CAST ─ ( UDT_expression ─ AS ─ character_data_definition ─ ) ──
```

1101A346

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
character_data_definition	the target character type, for example CHAR or VARCHAR, followed by optional FORMAT, NAMED, or TITLE attribute phrases.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax

```
─ UDT_expression ─ ( ┌──────────────────┐ character_data_type ┌──────────────────┐ ) ──
                    └─ data_attribute , ┘                     └─ , data_attribute ┘
```

1101B347

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
character_data_type	the target character type, for example CHAR or VARCHAR.

ANSI Compliance

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

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-character conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

The target character type of the cast definition does not have to be an exact match to the target character type of the implicit conversion. Teradata Database can use an implicit cast definition that specifies a CHAR, VARCHAR, or CLOB target type.

If multiple implicit cast definitions exist for converting the UDT to different character types, Teradata Database uses the implicit cast definition for the character type with the highest precedence. The following list shows the precedence of character types in order from lowest to highest precedence:

- CHAR
- VARCHAR
- CLOB

If no UDT-to-character implicit cast definitions exist, Teradata Database looks for other cast definitions that can substitute for the UDT-to-character implicit cast definition:

IF the following combination of implicit cast definitions exists ...				THEN Teradata Database ...
UDT-to-numeric	UDT-to-DATE	UDT-to-TIME	UDT-to-TIMESTAMP	
X				uses the UDT-to-numeric implicit cast definition. If multiple UDT-to-numeric implicit cast definitions exist, then Teradata Database returns an SQL error.
	X			uses the UDT-to-DATE implicit cast definition.
		X		uses the UDT-to-TIME implicit cast definition.
			X	uses the UDT-to-TIMESTAMP implicit cast definition.
X	X			reports an error.
X		X		
X			X	
	X	X		
	X		X	
		X	X	
X	X	X		
X	X		X	
X		X	X	
	X	X	X	
X	X	X	X	

Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a character type.

Example

Consider the following table definition, where euro is a UDT:

```
CREATE TABLE euro_sales_table
   (quarter INTEGER
   ,region VARCHAR(20)
   ,sales euro );
```

Assuming an appropriate cast definition exists for the euro UDT, the following statement converts the values in the sales column to CHAR(10):

```
SELECT region, CAST (sales AS CHAR(10))
FROM euro_sales_table
WHERE quarter = 1;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-DATE Conversion

Purpose

Converts a UDT expression to a DATE data type.

CAST Syntax

```
── CAST ── ( UDT_expression ── AS ── DATE ┬──────────────────────┬──────────────┬──────────────┬──── ) ────────────
                                          │┌────────────────────┐│              │              │
                                          └┤ date_data_attribute ├┘              │              │
                                                                                  │              │
                                                                                  └──────────────┘
                                                                                                  1101B348
```

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
date_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax

```
─ UDT_expression ─ ( ┬──────────────────────┬──────────────┬──────────────┬──────────────┬──── ) ────────────
                     │┌────────────────────┐│              │              │              │
                     └┤ data_attribute ,   ├┘              │              │              │
                                                           │              │              │
                                                           └──────────────┘              │
                                                                                         1101B349
```

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

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

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

Teradata Database performs implicit UDT-to-DATE conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

If no UDT-to-DATE implicit cast definition exists, Teradata Database looks for other cast definitions that can substitute for the UDT-to-DATE implicit cast definition:

IF the following combination of implicit cast definitions exists ...		THEN Teradata Database ...
UDT-to-Numeric	UDT-to-Character (non-CLOB)	
X		uses the UDT-to-numeric implicit cast definition. If multiple UDT-to-numeric implicit cast definitions exist, then Teradata Database returns an SQL error.
	X	uses the UDT-to-character implicit cast definition. If multiple UDT-to-character implicit cast definitions exist, then Teradata Database returns an SQL error.
X	X	reports an error.

Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a DATE type.

Example

Consider the following table definition, where datetime_record is a UDT:

```
CREATE TABLE support
   (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime_record UDT, the following statement converts the values in the information column to DATE:

```
SELECT id, CAST (information AS DATE) FROM support;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-INTERVAL Conversion

Purpose

Converts a UDT expression to an INTERVAL data type.

CAST Syntax

```
── CAST ── ( UDT_expression ── AS ── interval_data_definition ── ) ──
```

1101A350

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
interval_data_definition	the target predefined interval type followed by optional NAMED or TITLE attribute phrases.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax

```
─ UDT_expression ─ ( ┌──────────────────────┐ interval_data_type ┌──────────────────────┐ ) ──
                    └─ data_attribute , ─┘                      └─ , data_attribute ─┘
```

1101B351

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • NAMED • TITLE
interval_data_type	the target predefined interval type to which *UDT_expression* is to be converted.

ANSI Compliance

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

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Performing an implicit data type conversion requires a cast definition (see "Usage Notes") that specifies the following:

- the AS ASSIGNMENT clause
- a target data type that is in the same INTERVAL family as the target of the implicit cast:

This INTERVAL data type ...	Belongs to this INTERVAL family ...
• INTERVAL YEAR • INTERVAL YEAR TO MONTH • INTERVAL MONTH	Year-Month
• INTERVAL DAY • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE • INTERVAL MINUTE TO SECOND • INTERVAL SECOND	Day-Time

The target data type of the cast definition does not have to be an exact match to the target of the implicit type conversion.

Teradata Database performs implicit UDT-to-INTERVAL conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Example

Consider the following table definition, where datetime_record is a UDT:

```
CREATE TABLE support
   (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime_record UDT, the following statement converts the values in the information column to INTERVAL MONTH:

```
SELECT id, CAST (information AS INTERVAL MONTH) FROM support;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-Numeric Conversion

Purpose

Converts a UDT expression to a numeric data type.

CAST Syntax

```
── CAST ── ( UDT_expression ── AS ── numeric_data_definition ── ) ──
```

1101A352

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
numeric_data_definition	the target predefined numeric type followed by any optional FORMAT, NAMED, or TITLE attribute phrases.

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax

```
─ UDT_expression ─ ( ┌──────────────────────────┐ numeric_data_type ┌──────────────────────────┐ ) ──
                    └─┌──────────────────┐─┘                       └─┌──────────────────┐─┘
                      └─ data_attribute , ─┘                         └─ , data_attribute ─┘
```

1101B353

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.

Syntax element ...	Specifies ...
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
numeric_data_type	a predefined numeric type to which *UDT_expression* is to be converted.

ANSI Compliance

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

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-numeric conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes" on page 762) exists that specifies the AS ASSIGNMENT clause.

The target numeric type of the cast definition does not have to be an exact match to the target numeric type of the implicit conversion. Teradata Database can use an implicit cast definition that specifies a BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, NUMBER, or REAL/FLOAT/DOUBLE target type.

If multiple implicit cast definitions exist for converting the UDT to different numeric types, Teradata Database uses the implicit cast definition for the numeric type with the highest precedence. The following list shows the precedence of numeric types in order from lowest to highest precedence:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT

- DECIMAL/NUMERIC
- NUMBER
- REAL/FLOAT/DOUBLE

If no UDT-to-numeric implicit cast definitions exist, Teradata Database looks for other cast definitions that can substitute for the UDT-to-numeric implicit cast definition:

IF the following combination of implicit cast definitions exists ...		THEN Teradata Database ...
UDT-to-DATE	UDT-to-Character	
X		uses the UDT-to-DATE implicit cast definition.
	X	uses the UDT-to-character implicit cast definition. The character type cannot be CLOB. If multiple UDT-to-character implicit cast definitions exist, then Teradata Database returns an SQL error.
X	X	reports an error.

Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to the substitute data type, and then implicitly cast the substitute data type to a numeric type.

Example

Consider the following table definition, where euro is a UDT:

```
CREATE TABLE euro_sales_table
   (quarter INTEGER
   ,region VARCHAR(20)
   ,sales euro );
```

Assuming an appropriate cast definition exists for the euro UDT, the following statement converts the values in the sales column to DECIMAL(10,2):

```
SELECT SUM (CAST (sales AS DECIMAL(10,2))) FROM euro_sales_table;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-TIME Conversion

Purpose

Converts a UDT expression to a TIME data type.

CAST Syntax

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
time_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

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

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-TIME conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

If no UDT-to-TIME implicit cast definition exists, Teradata Database looks for a UDT-to-CHAR or UDT-to-VARCHAR cast definition that can substitute for the UDT-to-TIME implicit cast definition. Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to a character data type, and then implicitly cast the character data type to a DATE type. If multiple UDT-to-character implicit cast definitions exist, then Teradata Database returns an SQL error.

Example

Consider the following table definition, where datetime_record is a UDT:

```
CREATE TABLE support
   (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime_record UDT, the following statement converts the values in the information column to TIME WITH TIME ZONE:

```
SELECT id, CAST (information AS TIME WITH TIME ZONE) FROM support;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-TIMESTAMP Conversion

Purpose

Converts a UDT expression to a TIMESTAMP data type.

CAST Syntax

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.
timestamp_data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Teradata Conversion Syntax

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
data_attribute	one of the following optional data attributes: • FORMAT • NAMED • TITLE
fractional_seconds_precision	a single digit representing the number of significant digits in the fractional portion of the SECOND field. Values for *fractional_seconds_precision* range from 0 through 6 inclusive. The default precision is 6.

ANSI Compliance

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

Usage Notes

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-TIMESTAMP conversions for the following operations:

- UPDATE
- INSERT
- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs

- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

Performing an implicit data type conversion requires that an appropriate cast definition (see "Usage Notes") exists that specifies the AS ASSIGNMENT clause.

If no UDT-to-TIMESTAMP implicit cast definition exists, Teradata Database looks for a UDT-to-CHAR or UDT-to-VARCHAR cast definition that can substitute for the UDT-to-TIMESTAMP implicit cast definition. Substitutions are valid because Teradata Database can use the implicit cast definition to cast the UDT to a character data type, and then implicitly cast the character data type to a TIMESTAMP type. If multiple UDT-to-character implicit cast definitions exist, then Teradata Database returns an SQL error.

Example

Consider the following table definition, where datetime_record is a UDT:

```
CREATE TABLE support
   (id INTEGER
   ,information datetime_record );
```

Assuming an appropriate cast definition exists for the datetime_record UDT, the following statement converts the values in the information column to TIMESTAMP:

```
SELECT id, CAST (information AS TIMESTAMP) FROM support;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

UDT-to-UDT Conversion

Purpose

Converts a UDT expression to a different UDT type.

CAST Syntax

```
── CAST ── ( UDT_expression ── AS ── UDT_data_definition ── ) ──
```

1101A356

where:

Syntax element ...	Specifies ...
UDT_expression	an expression that results in a UDT data type. For details on expressions that can result in UDT data types, see "SQL UDF" on page 1290.
UDT_data_definition	a UDT type to which *UDT_expression* is to be converted, followed by any of the following optional attribute phrases: • FORMAT • NAMED • TITLE

ANSI Compliance

This is ANSI SQL:2011 compliant.

As an extension to ANSI, CAST permits the use of data attribute phrases such as FORMAT.

Usage Notes

Explicit UDT-to-UDT conversion using Teradata conversion syntax is not supported.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit Type Conversion

Teradata Database performs implicit UDT-to-UDT casts for the following operations:

- UPDATE
- INSERT

- Passing arguments to stored procedures, external stored procedures, UDFs, and UDMs
- Specific system operators and functions identified in other sections of this book, unless the DisableUDTImplCastForSysFuncOp field of the DBS Control Record is set to TRUE

An implicit data type conversion involving a UDT can only be performed if the cast definition specifies the AS ASSIGNMENT clause. For more information, see "CREATE CAST" in *SQL Data Definition Language*.

Example

Consider the following table definitions, where euro and us_dollar are UDTs:

```
CREATE TABLE euro_sales_table
   (euro_quarter INTEGER
   ,euro_region VARCHAR(20)
   ,euro_sales euro );

CREATE TABLE us_sales_table
   (us_quarter INTEGER
   ,us_region VARCHAR(20)
   ,us_sales us_dollar );
```

Assuming an appropriate cast definition exists for converting the euro UDT to a us_dollar UDT, the following statement performs a us_dollar UDT to euro UDT conversion:

```
INSERT INTO euro_sales_table
   SELECT us_quarter, us_region, CAST (us_sales AS euro)
   FROM us_sales_table;
```

Related Topics

For details on data types and data attributes, see *SQL Data Types and Literals*.

CHAPTER 14 Data Type Conversion Functions

This chapter describes functions that convert data from one data type to another.

TO_BYTES

Purpose

Decodes a sequence of characters in a given encoding into a sequence of bits. The following encodings are supported:

- BaseX
- BaseY
- Base64M (MIME)
- ASCII

where X is a power of 2 (for example, 2, 8, 16) and Y is not a power of 2 (for example, 10 and 36).

Syntax

```
┬──────────────────┬─ TO_BYTES ── (instring, in_encoding) ──────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A984

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
instring	the string value, that is, sequence of characters to be decoded.
in_encoding	the encoding TO_BYTES uses to return the sequence of characters specified by *instring*.

ANSI Compliance

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

Invocation

TO_BYTES is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *instring* = VARCHAR or CLOB
- *in_encoding* = VARCHAR(64)

If *in_encoding* is not one of the supported encodings, an error is returned.

If either *instring* or *in_encoding* is NULL, the result is NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The result data type is VARBYTE or BLOB.

Example 1

The following query:

```
SELECT CAST (TO_BYTES ('5A', 'base16') as BTYE (16));
TO_BYTES('5A','base16')
```

returns '5A-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 '

Example 2

The following query:

```
SELECT CAST (TO_BYTES ('-22EEVX', 'base36') as BYTE (16));
TO_BYTES('-22EEVX','base36')
```

returns 'F8-8D-33-23-00-00-00-00-00-00-00-00-00-00-00-00'

FROM_BYTES

Purpose

Encodes a sequence of bits into a sequence of characters representing its encoding. The following encodings are supported:

- BaseX
- BaseY
- Base64M (MIME)
- ASCII

where X is a power of 2 (for example, 2, 8, 16) and Y is not a power of 2 (for example, 10 and 36).

Syntax

```
┬──────────────────┬─ FROM_BYTES ── (instring, out_encoding) ──────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A985

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
instring	the binary string value, that is, the sequence of characters to be encoded.
out_encoding	the encoding FROM_BYTES uses to encode the sequence of characters specified by *instring*.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Invocation

FROM_BYTES is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *instring* = VARBYTE or BLOB
- *out_encoding* = VARCHAR(64)

If *out_encoding* is not one of the supported encodings, an error is returned.

If the resulting conversion overflows the result data type length, an error is generated.

If either *instring* or *out_encoding* is NULL, the result is NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The result data type is VARCHAR or CLOB. Note that the return value may be NULL.

Example 1

The following query:

```
SELECT from_bytes('5A1B'XB, 'base10');
```

returns '23067'.

Example 2

The following query:

```
SELECT from_bytes('5A3F'XB, 'ASCII');
```

returns 'Z\ESC '.

Example 3

The following query:

```
SELECT from_bytes('5A1B'XB, 'base16');
```

returns '5A1B'.

TO_NUMBER

Purpose

Converts *string_expr* to a NUMBER data type.

Syntax

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
string_expr	a character argument. *string_expr* contains a number in the format specified by the optional *format_arg*. If conversion fails, NULL is returned.
format_arg	an optional character argument. If *format_arg* is not valid, the function returns an error. If *format_arg* is NULL, TO_NUMBER returns a NULL.

Syntax element ...	Specifies ...
NLS_*param*	an optional character argument. NLS_*param* specifies characters that are returned by number format elements: • Decimal character • Group separator • Local currency symbol • Dual currency symbol • International currency symbol Valid values for *param* are: • NUMERIC_CHARACTERS = ' '*dg* ' ' • CURRENCY = ' ' *text* ' ' • DUAL_CURRENCY = ' ' *text* ' ' • ISO_CURRENCY= ' ' *text* ' ' The characters *d* and *g* represent the decimal character and group separator respectively. They must be different single-byte characters. *Text* must be enclosed in apostrophes. Ten characters are available for the currency symbol. The SDF (Specification for Data Formatting) file is used to determine any default formatting. If NLS_*param* is specified, it overrides any defaults specified in the SDF file. If NLS_*param* is NULL, NULL is returned.

format_arg Format Elements

Format Element...	Example...	Returns ...
, (comma)	9,999	a comma in the specified position. • A comma cannot begin a number format. • A comma cannot appear to the right of a decimal character or period in a number format.
. (period)	9.99	a decimal point. You can only specify one period in a number format.
$	$9999	a value with a leading dollar sign.
0	09999 9990	leading zeros. trailing zeros.
9	9999	a value with the specified number of digits with a leading space if positive or with a leading minus if negative.
B	B9999	blanks for the integer part of a fixed point number when the integer part is zero.

Format Element...	Example...	Returns ...
C	C999	the ISO currency symbol as specified in the ISOCurrency element in the SDF file.
D	99D99	the character that separates the integer and fractional part of non-monetary values. This is specified in the RadixSeparator element in the SDF file.
EEEE	9.9EEEE	a value in scientific notation.
G	9G999	(group separator) the character that separates groups of digits in the integer part of non-monetary values. This is specified in the GroupSeparator element in the SDF file.
L	L999	(local currency) the string representing the local currency as specified in the Currency element in the SDF file.
MI	9999MI	a trailing minus sign if the value is negative. The MI format element can appear only in the last position of a number format.
PR	9999PR	a negative value in <angle brackets>, or a positive value with a leading and trailing blank. The PR format element can appear only in the last position of a number format.
RN	RN rn	a value as Roman numerals in upper case. a value as Roman numerals in lower case. Valid values: a value between 1 and 3999.
S	S9999 9999S	a negative value with a leading or trailing minus sign. a positive value with a leading or trailing plus sign. The S format element can appear only in the first or last position of a number format.
TM	TM TM9 TME	(text minimum format) returns the smallest number of characters possible. This element is case insensitive. TM or TM9 return the number in fixed notation unless the output exceeds 64 characters. If the output exceeds 64 characters, the number is returned in scientific notation. TME returns the number in scientific notation with the smallest number of characters. You cannot precede this element with an other element. You can follow this element only with one 9 or one E (or e), but not with any combination of these.
U	U9999	(dual currency) the string that represents the dual currency as specified in the DualCurrency element in the SDF file.
V	999V99	a value multiplied by 10 to the n (and, if necessary, rounded up), where *n* is the number of 9's after the V.

Format Element...	Example...	Returns ...
X	XXXXX xxxxx	the hexadecimal value of the specified number of digits. If the specified number is not an integer, the function will round it to an integer. This element accepts only positive values or zero. Negative values return an error. You can precede this element only with zero (which returns leading zeroes) or FM. Any other elements return an error. If you do not specify zero or FM, the return always has one leading blank.

ANSI Compliance

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

Invocation

TO_NUMBER is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must be VARCHAR data types.

You can also pass arguments with data types that can be converted to the above data type using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

TO_NUMBER is a scalar function whose return value data type is NUMBER.

Usage Notes

The following are examples of the TO_NUMBER syntax that Teradata supports. The examples deviate from the TO_NUMBER syntax Oracle supports. The function:

- Does not supported *not* having a closing angle bracket for the PR format element. This example:

  ```
  SELECT TO_NUMBER ('<123', '999PR')
  ```

  returns NULL because the number is not in the correct format.
- Allows a space at the end of the number if the PR format element is used and the number is positive. This example:

  ```
  SELECT TO_NUMBER (' 123 ', '999PR')
  ```

  returns 123.
- Allows multiple integer digits in scientific notation. This example:

```
SELECT TO_NUMBER ('12E3', '99EEEE')
```

returns 12000.

- Allows commas in the integer portion of scientific notation. This example:

  ```
  SELECT TO_NUMBER ('1,234E3', '9,999EEEE')
  ```

  returns 1234000.
- Validates commas when there are two arguments, even when there is no decimal in the format. This example:

  ```
  SELECT TO_NUMBER ('1234', '9,999')
  ```

  returns NULL because the number is not in the correct format.
- Treats an empty string like any other string. This example:

  ```
  SELECT TO_NUMBER ('', 'BAD Format')
  ```

  returns an error, rather than NULL, because the format is not valid.
- Allows the use of period (.) and comma (,) with D and G are in the same format string. This example:

  ```
  SELECT TO_NUMBER ('1,234.99', '9,999D99')
  ```

  returns 1234.99

Example 1

The following query:

```
SELECT TO_NUMBER ('1789.96', '9999.99');
```

returns the result 1789.96.

Example 2

The following query:

```
SELECT TO_NUMBER('dollar123','L999','NLS_CURRENCY=''dollar''');
```

returns 123.

TO_CHAR(Numeric)

Purpose

Converts *numeric_expr* to a character string.

Syntax

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.

numeric_expr	a numeric argument.
	If the conversion fails, '#' characters are returned. When the integer portion of the number is less than the digits specified in the format string, a string with a variable number of '#' characters is returned. The number of '#' characters returned is equal to the length of the maximum possible character string result based on the format string or data type of *numeric_expr*. This includes a '#' character representing the optional plus/minus sign.
	For example:
	`SELECT TO_CHAR(12345678,'99')`
	returns '###'.
	`SELECT TO_CHAR(12345678,'99999')`
	returns '#####'.
	`SELECT TO_CHAR(12345678,'99V9')`
	returns '####'.

format_arg	an optional character argument. *format_arg* is used to format the numeric values. If *format_arg* is omitted, *numeric_expr* is converted to a string exactly long enough to hold its significant digits. If *format_arg* is not valid, an error is returned. For *format_arg* values, see "TO_NUMBER" on page 788.
NLS_*param*	an optional character argument to be used with *format_arg*. For NLS_*param* values, see "TO_NUMBER" on page 788.

ANSI Compliance

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

Invocation

TO_CHAR(Numeric) is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric_expr* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *format_arg* = CHAR or VARCHAR
- NLS_*param* = CHAR or VARCHAR

Result Type

TO_CHAR(Numeric) is a scalar function whose return data type is VARCHAR CHARACTER SET UNICODE.

Example

The following query:

```
SELECT TO_CHAR(50000, '$99,999.99');
```

returns '$50,000.00'.

TO_CHAR(DateTime)

Purpose

Converts a *date_timestamp_value* to a character string.

Syntax

```
┬──────────────────┬─ TO_CHAR ── ( date_timestamp_value ┬──────────────┬) ──────────
└─ TD_SYSFNLIB. ──┘                                    └, format_arg ┘
```

1101A988

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
date_timestamp_value	a datetime or interval argument.
format_arg	an optional character argument. For valid *format_arg* values, see "TO_DATE" on page 797. *format_arg* formats the date, time stamp, or interval value. If *format_arg* is omitted, a default format is used. If *format_arg* is not valid, an error is returned.

ANSI Compliance

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

Invocation

TO_CHAR(DateTime) is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *date_timestamp_value* = DATE, TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, or an Interval data type, such as INTERVAL YEAR TO MONTH.
- *format_arg* = CHAR or VARCHAR

Result Type

TO_CHAR(DateTime) is a scalar function whose return data type is VARCHAR CHARACTER SET UNICODE.

Example

The following query:

```
SELECT TO_CHAR(DATE '2003-12-23', 'DD-MON-YYYY');
```

returns a result of '23-DEC-2003'.

TO_DATE

Purpose

Converts *string_expr* to a DATE data type.

TO_DATE does not convert data to any of the other datetime data types. Do not use the TO_DATE function with a DATE value for *string_expr*. The first two digits of the returned DATE value can differ from the original *string_expr* depending on *format_arg* or the default date format.

Syntax

```
┬──────────────┬── TO_DATE ── ( string_expr, format_arg ) ──────────────────────
└── TD_SYSFNLIB. ┘
```

1101A906

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
string_expr	a character argument. If conversion fails, TO_DATE returns NULL.
format_arg	a character argument. *format_arg* is a data format specifying the format of *string_expr*. If *format_arg* is omitted, the following default date format is used: YYYY-MM-DD If format_arg is NULL, TO_DATE returns NULL. If *format_arg* is not valid, an error is returned.

format_arg Format Elements

Element	Description
- / , . ; : “text”	Punctuation characters are ignored and text enclosed in quotation marks is ignored.
AD A.D.	AD indicator.
AM A.M.	Meridian indicator.
BC B.C.	BC indicator.
CC SCC	Century. If the last 2 digits of a 4-digit year are between 01 and 99 inclusive, the century is 1 greater than the first 2 digits of that year. If the last 2 digits of a 4-digit year are 00, the century is the same as the first 2 digits of that year. Note: CC and SCC are only supported by the TO_CHAR function.
D	Day of week (1-7).
DAY	Name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DL	Date Long. Equivalent to the format string ‘FMDay, Month FMDD, YYYY’. Note: DL is only supported by the TO_CHAR function.
DS	Date Short. Equivalent to the format string ‘FMMM/DD/YYYYFM’. Note: DS is only supported by the TO_CHAR function.
DY	abbreviated name of day.
E	not supported.

Element	Description
EE	not supported.
FF [1..9]	Fractional seconds. Use [1..9] to specify the number of fractional digits. FF without any number following it prints a decimal followed by digits equal to the number of fractional seconds in the input data type. If the data type has no fractional digits, FF prints nothing. Note: Any fractional digits beyond 6 digits are truncated.
FM	Format Minimum mode. The value is returned with no leading or trailing blanks. This may be toggled on and off by adding an 'FM' before and after a section that should use minimum space. Note: FM is only supported by the TO_CHAR function.
FX	Requires the character data and the format model to be an exact match. Note: FX is not supported by the TO_CHAR function.
HH HH12	Hour of day (1-12).
HH24	Hour of the day (0-23).
IW	Week of year (1-52 or 1-53) based on ISO model. Note: IW is only supported by the TO_CHAR function.
IYY IY I	Last 3, 2, or 1 digits of ISO year. Note: I, IY, and IYY are only supported by the TO_CHAR function.
IYYY	4-digit year based on the ISO standard. Note: IYYY is only supported by the TO_CHAR function.
J	Julian day, the number of days since January 1, 4713 BC. Number specified with J must be integers. Teradata uses the Gregorian calendar in calculations to and from Julian Days.
MI	Minute (0-59).
MM	Month (01-12).

Element	Description
MON	Abbreviated name of month.
MONTH	Name of month.
PM P.M.	Meridian indicator.
Q	Quarter of year (1, 2, 3, 4). Note: Q is only supported by the TO_CHAR function.
RM	Roman numeral month (I - XII).
RR	Stores 20th century dates in the 21st century using only 2 digits. If the current year and the specified year are both in the range of 0-49, the date is in the current century. Note: RR is not supported by the TO_CHAR function.
RRRR	Round year. Accepts either 4-digit or 2-digit input. 2-digit input provides the same return as RR. Note: RRRR is not supported by the TO_CHAR function.
SP	Spelled. Any numeric element followed by SP is spelled in English words. The words are capitalized according to how the element is capitalized. For example: 'DDDSP' specifies all uppercase, 'DddSP' specifies that the first letter is capitalized, and 'dddSP' specifies all lowercase. Note: SP is only supported by the TO_CHAR function.
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).
TS	Time Short. Equivalent to the format string 'HH:MI:SS AM'. Note: TS is only supported by the TO_CHAR function.
TZD	not supported.
TZH	Time zone hour.
TZM	Time zone minute.

Element	Description
TZR	Time zone region. Equivalent to the format string 'TZH:TZM'. Note: TZR is only supported by the TO_CHAR function.
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the 7th day of the year. Note: WW is only supported by the TO_CHAR function.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh. Note: W is only supported by the TO_CHAR function.
X	Local radix character.
Y,YYY	Year with comma in this position.
YEAR SYEAR	Year, spelled out. S prefixes BC dates with a minus sign. Note: YEAR and SYEAR are only supported by the TO_CHAR function.
YYYY SYYYY	4-digit year. S prefixes BC dates with a minus sign.
YYY YY Y	Last 3, 2, or 1 digit of year. If the current year and the specified year are both in the range of 0-49, the date is in the current century.

ANSI Compliance

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

Invocation

TO_DATE is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must be a VARCHAR data type.

You can also pass arguments with data types that can be converted to the above type using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

TO_DATE is an overloaded scalar function whose return value data type is DATE.

Example

The following query:

```
SELECT TO_DATE ('January 15, 1989, 11:00 A.M.', 'Month dd, YYYY, HH: MI
A.M.');
```

returns the result 15-JAN-89.

TO_TIMESTAMP

Purpose

Converts *string_expr* or *integer_expr* to a TIMESTAMP data type.

Syntax

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
string_expr	a character argument. If the conversion fails, NULL is returned.
format_arg	a character argument. *format_arg* specifies the format of *string_expr*. For valid *format_arg* values, see "TO_DATE" on page 797. Any fractional seconds that are specified beyond 6 digits are truncated. If *format_arg* is omitted, TO_TIMESTAMP uses a default timestamp format of YYYY-MM-DD HH24:MI:SS.FF6. If *format_arg* is NULL, TO_TIMESTAMP returns NULL. If *format_arg* is not valid, an error is returned.
integer_expr	a number argument. POSIX epoch conversion is implicit in the TO_TIMESTAMP function when *integer_expr* is passed to the function. POSIX epoch is the number of seconds that have elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970.

ANSI Compliance

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

Invocation

TO_TIMESTAMP is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must be of VARCHAR data type or the following Numeric data types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

TO_TIMESTAMP is an overloaded scalar function whose return value data type is TIMESTAMP(6).

Example 1

The following query:

```
SELECT TO_TIMESTAMP ('January 15, 1989, 11:00 A.M.', 'Month dd, YYYY, HH:
MI A.M.');
```

returns the result 1989-01-15 11:00:00.000000.

Example 2

The following query, an example of POSIX epoch conversion:

```
SELECT TO_TIMESTAMP(242525);
```

returns the result 1970-01-03 19:22:05.000000

TO_TIMESTAMP_TZ

Purpose

Converts *string_expr* to a TIMESTAMP WITH TIME ZONE data type.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┴─ TO_TIMESTAMP_TZ ( string_expr ┌──────────────────────┐ ) ──
                                                       │                      │
                                                       └─ ( , format_arg ) ───┘
```

1145-009A

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
string_expr	a character argument. If the conversion fails, NULL is returned.
format_arg	a character argument. *format_arg* specifies the format of *string_expr*. *format_arg* does not convert data to any of the other datetime data types. Any fractional seconds that are specified beyond 6 digits are truncated. If *format_arg* is omitted, a default timestamp with time zone format of YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM is used. For valid *format_arg* values, see "TO_DATE" on page 797. If *formal_arg* is not valid, an error is returned. If *format_arg* is NULL, NULL is returned.

ANSI Compliance

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

Invocation

TO_TIMESTAMP_TZ is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must be a VARCHAR data type.

You can also pass arguments with data types that can be converted to the above type using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

TO_TIMESTAMP_TZ is an overloaded scalar function whose return value data type is TIMESTAMP(6) WITH TIME ZONE.

Example

The following query:

```
SELECT TO_TIMESTAMP_TZ ('January 15, 1989, 11:00 A.M.', 'Month dd, YYYY,
HH: MI A.M.');
```

returns the result 1989-01-15 11:00:00.000000+00:00.

TO_YMINTERVAL

Purpose

Converts *string_value* specified as either:

- SQL interval format
- ISO duration format

into an INTERVAL YEAR(4) TO MONTH value.

Any value that overflows the INTERVAL(4) YEAR TO MONTH value results in an error.

Syntax

```
┬──────────────────┬─ TO_YMINTERVAL ── ( string_value ) ──────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A910

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
string_value	a character argument. If *string_value* is an empty string or NULL, NULL is returned.

ANSI Compliance

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

Invocation

TO_YMINTERVAL is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

CHAR, VARCHAR

string_value Formats

- SQL Interval Format

The format for an interval year to month is [+ | -]YY-MM. YY is an integer between 0 and 9999, and MM is an integer between 0 and 11.

- ISO Duration Format

  The format for a duration is [-]P[n]Y[n]M[n] where *n* specifies the value of the element (for example, 4Y is 4 years).

  This represents a subset of the ISO duration format and the use of any arguments other than P, Y, and M are ignored. Leading and trailing zeroes are optional. Spaces are not allowed within the element format. Elements may be omitted if they are zero, but at least one element and its number is required.

  A leading negative sign denotes a negative interval. If omitted, the interval is positive. A positive sign is not allowed.

  Element values are integers between 0 and any value that will not cause an interval overflow. The values can exceed their normal time limit. For example, P14M represents 14 months, even though there are 12 months in a year. The ISO duration letter designators have the following meaning (only the P, Y, and M designators are allowed for the YEAR to MONTHS interval):

ISO Designator	Definition
P	Duration. It is always at the beginning.
Y	Year. It follows the number of years.
M	Month. It follows the number of months.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The return value data type is INTERVAL YEAR(4) TO MONTH.

Example 1

The following query, which specifies an SQL interval format:

```
SELECT TO_YMINTERVAL('04-10');
```

returns the result '04-10'.

Example 2

The following query, which specifies an ISO duration:

```
SELECT TO_YMINTERVAL('P100Y4M');
```

returns the result '100-04', which is 100 years, 4 months.

Example 3

The following query:

```
SELECT TO_YMINTERVAL('P20Y3M4DT12H23M34.234S');
```

returns an error because only the P, Y, and M designators are allowed.

Example 4

The following query, which specifies an ISO duration of -14 months:

```
SELECT TO_YMINTERVAL('-P14M');
```

returns the result '-01-02', which is negative 1 year, 2 months.

TO_DSINTERVAL

Purpose

Converts *string_value* specified as either of the following:

- SQL interval format
- ISO duration format

into an INTERVAL DAY(4) TO SECOND(6) value.

Any value that overflows the INTERVAL DAY(4) TO SECOND(6) value results in an error.

Syntax

```
┬──────────────────┬─ TO_DSINTERVAL ( string_value ) ──────────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A911

where

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
string_value	a character argument. If *string_value* is an empty string or NULL, NULL is returned.

ANSI Compliance

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

Invocation

TO_DSINTERVAL is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- CHAR
- VARCHAR

string_value Formats

- SQL Interval Format

  The format for an interval day to second is [+ | -]DD HH:MM:SS[.xxxxxx]. One or more spaces separate the day from hour element. There can be any number of spaces between elements. DD is an integer between 0 and 9999, HH is an integer between 0 and 23, MM is an integer between 0 and 59, and SS is an integer between 0 and 59. The optional fractional seconds can be from .0 to .999999.

- ISO Duration Format

  The format for a duration is [-]P[n]DT[n]H[n]M[n][.frac_secs]S where n specifies the value of the element (for example, 4H is 4 hours). This represents a subset of the ISO duration format and any other duration letters (including valid ISO letters such as Y and M) result in an error.

  Leading and trailing zeroes are optional. Spaces are not allowed within the element format. Elements may be omitted if they are zero, but at least one element and its number is required.

  A leading negative sign is used to denote a negative interval. If omitted, the interval is positive. A positive sign is not allowed. Element values are integers between 0 and any value that will not cause an interval overflow. Fractional seconds are optional and can be between .0 and .999999.

  The T designator always precedes the time elements (and removes the ambiguity of M between months and minutes within the full ISO duration syntax). If there are no H, M, or S time elements, T is omitted.

  Values can exceed their normal time limit. For example, PT72H represents 72 hours even though there are 24 hours in a day. The ISO duration letter designators have the following meaning (only the P, D, T, H, M, and S designators are allowed for the DAY to SECONDS interval):

ISO Designator	Definition
P	Duration. It is always at the beginning.
D	Day. It follows the number of days.
T	Time. It precedes the time elements H, M, and S.
H	Hour. It follows the number of hours.
M	Minutes. It follows the number of minutes.
S	Second. It follows the number of seconds.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The return value data type is INTERVAL DAY(4) TO SECOND(6).

Example 1

The following query, which specifies an SQL interval format:

```
SELECT TO_DSINTERVAL('100 04:23:59');
```

returns the result '100 04:23:59.000000'.

Example 2

The following query, which specifies an SQL interval format:

```
SELECT TO_DSINTERVAL('100 04:23:59.543');
```

returns the result '100 04:23:59.543000'.

Example 3

The following query, which specifies an ISO duration:

```
SELECT TO_DSINTERVAL('P100DT4H23M59S');
```

returns the result '100 04:23:59.000000'.

Example 4

The following query, which specifies an ISO duration of 73 hours:

```
SELECT TO_DSINTERVAL('PT73H');
```

returns the result '3 01:00:00.000000', which is 3 days, 1 hours.

Example 5

The following query:

```
SELECT TO_DSINTERVAL('P2MT12H');
```

returns an error because the month designator, '2M', isn't allowed.

NUMTODSINTERVAL

Purpose

Converts *numeric_value* into an INTERVAL DAY(4) TO SECOND(6) value.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘ NUMTODSINTERVAL ── ( numeric_value, interval_unit ) ──────────────
```

1101A912

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
numeric_value	a numeric argument. If *numeric_value* is NULL, NULL is returned
interval_unit	a character argument. *interval_unit* specifies the unit of value for *numeric_value*. It must be one of the following case-insensitive values: • 'DAY' • 'HOUR' • 'MINUTE' • 'SECOND' Any value that overflows the INTERVAL DAY(4) TO SECOND(6) value results in an error. If *interval_unit* is NULL, NULL is returned.

ANSI Compliance

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

Invocation

NUMTODSINTERVAL is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *interval_unit* = CHAR or VARCHAR

Result Type

The return value data type is INTERVAL DAY(4) TO SECOND(6).

Example 1

The following query:

```
SELECT NUMTODSINTERVAL(100, 'DAY');
```

returns the result '100 00:00:00.000000', which is 100 days.

Example 2

The following query:

```
SELECT NUMTODSINTERVAL(100, 'HOURS');
```

returns the result '4 04:00:00.000000', which is 4 days, 4 hours.

Example 3

The following query:

```
SELECT NUMTODSINTERVAL(1600, 'MINUTES');
```

returns the result '1 02:40:00.000000', which is 1 day, 2 hours, 40 minutes.

Example 4

The following query:

```
SELECT NUMTODSINTERVAL(3.5, 'HOURS');
```

returns the result 3 hours, 30 minutes.

NUMTOYMINTERVAL

Purpose

Converts *numeric_value* into an INTERVAL YEAR(4) TO MONTH value.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── NUMTOYMINTERVAL ── ( numeric_value, interval_unit ) ──────────
```

1101A913

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
numeric_value	a numeric argument. If *numeric_value* is NULL, NULL is returned.
interval_unit	a character argument. *interval_unit* specifies the unit of value for *numeric_value*. It must be one of the following case-insensitive values: • 'YEAR' • 'MONTH' Any value that overflows the INTERVAL YEAR(4) TO MONTH value results in an error. If *interval_unit* is NULL, NULL is returned.

ANSI Compliance

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

Invocation

NUMTOYMINTERVAL is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *numeric_value* = BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, or NUMBER
- *exponent_value* = CHAR OR VARCHAR

Result Type

The return value data type is INTERVAL YEAR(4) TO MONTH.

Example 1

The following query:

```
SELECT NUMTOYMINTERVAL(40, 'YEAR');
```

returns the result '40-00', which is 40 years, 0 months.

Example 2

The following query:

```
SELECT NUMTOYMINTERVAL(100, 'MONTHS');
```

returns the result '08-04', which is 8 years, 4 months.

CHAPTER 15 DateTime and Interval Functions and Expressions

This chapter describes functions and expressions that operate on ANSI DateTime and Interval values, as well as functions and expressions that operate on Teradata DATE values, which are extensions to the ANSI SQL:2011 standard.

Overview

ANSI DateTime Data Types

ANSI DateTime data types include:

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

Interval Data Types

There are two categories of ANSI Interval data types:

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

ANSI DateTime and Interval Data Type Assignment Rules

Data Type Compatibility and Conversion

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

IF the source type is ...	AND the target type is ...	THEN ...
DATE	DATE	the types are compatible and assignments do not require conversion. For compatibility with existing Teradata assignments, non-ANSI operations such as assigning a DATE to an INTEGER or an INTEGER to a DATE (with validity checking) follow existing Teradata assignment rules.
TIME	TIME	the types are compatible and assignments do not require conversion. The Teradata system value TIME is encoded as a REAL and is not compatible with ANSI TIME or TIME WITH TIME ZONE.
TIMESTAMP	TIMESTAMP	the types are compatible and assignments do not require conversion.
Year-Month INTERVAL	Year-Month INTERVAL	
Day-Time INTERVAL	Day-Time INTERVAL	

IF the source type is ...	AND the target type is ...	THEN ...
Numeric	DATE	Teradata Database performs implicit type conversion before the assignment. See "Implicit Type Conversions" on page 579 for details.
DATE	• Character • Numeric • TIMESTAMP	
Character	• DATE • TIME • TIMESTAMP	
TIME	TIMESTAMP	
TIMESTAMP	• DATE • TIME	
Interval Note: The INTERVAL type must have only one field. For example, INTERVAL YEAR.	Exact Numeric	
Exact Numeric	Interval[a]	

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

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

```
── CAST ─ ( expression ─ AS ─┬─ ansi_sql_data_type ─┬─ ) ──────────────────────
                             └─ data_definition_list ─┘
                                                                  1101A627
```

where:

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

For more information, see "CAST in Explicit Data Type Conversions" on page 586.

Interval Data Type Assignment Rules

The following rules apply to Year-Month INTERVAL assignments.

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

The following rules apply to Day-Time INTERVAL assignments.

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

Scalar Operations on ANSI SQL:2011 DateTime and Interval Values

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

Scalar operations include:

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

Data Type Compatibility

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

To convert ANSI DateTime or Interval expressions, use the CAST function. See "CAST in Explicit Data Type Conversions" on page 586.

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 AT LOCAL, AT [TIME ZONE] *expression*, or AT [TIME ZONE] *time_zone_string*.

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 830.
date_time_primary	one of the following elements, any of which must have the appropriate DateTime type: • Column reference • DateTime literal value For details on DateTime literals, see *SQL Data Types and Literals*. • DateTime function reference For example, the result of a CASE expression or CAST function or DateTime built-in function such as CURRENT_DATE or CURRENT_TIME. • Scalar function reference • Aggregate function reference • (*table_expression*) A scalar subquery. • (*date_time_timestamp_expression*)
AT LOCAL	that the default time zone displacement based on the current session time zone is used. The current session time zone may be specified as a time zone string or a time zone displacement expressed as an Interval data type that defines the local time zone offset.
AT [TIME ZONE] *expression*	that the time zone displacement defined by *expression* is used. The data type of *expression* should be INTERVAL HOUR(2) TO MINUTE or it must be a data type that can be implicitly converted to INTERVAL HOUR(2) TO MINUTE.
AT [TIME ZONE] *time_zone_string*	that *time_zone_string* is used to determine the time zone displacement.

AT LOCAL and AT TIME ZONE Time Zone Specifiers

A *date_time_primary* can include an AT LOCAL or AT [TIME ZONE] clause only if the *date_time_primary* evaluates to a TIME or TIMESTAMP value or is the built-in function CURRENT_DATE or DATE.

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

The *expression* that specifies the time zone displacement in an AT [TIME ZONE] clause is implicitly converted, as needed and if allowed, to a time zone displacement or time zone string depending on its data type as defined in the following table:

Data type of *expression*	Implicit Conversion
INTERVAL HOUR(*n*) TO MINUTE where *n* is not 2	`CAST(`*`expression`*` AS INTERVAL HOUR(2) TO MINUTE)`
INTERVAL HOUR INTERVAL DAY INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL HOUR INTERVAL HOUR TO SECOND INTERVAL MINUTE INTERVAL MINUTE TO SECOND INTERVAL SECOND	`CAST(`*`expression`*` AS INTERVAL HOUR(2) TO MINUTE)`
BYTEINT SMALLINT INTEGER BIGINT DECIMAL/NUMERIC if the fractional precision is 0	`CAST(CAST(`*`expression`*` AS INTERVAL HOUR(2)) AS INTERVAL HOUR(2) TO MINUTE)`
DECIMAL/NUMERIC if the fractional precision is greater than 0	`CAST(CAST((`*`expression`*`)*60 AS INTERVAL MINUTE(4)) AS INTERVAL HOUR(2) TO MINUTE)`

Character with CHARACTER SET UNICODE	`CAST(CAST(`*`expression`*` AS INTERVAL HOUR(2)) AS INTERVAL HOUR(2) TO MINUTE)`
	If an error occurs for the above CAST statement, Teradata Database attempts the following:
	`CAST(`*`expression`*` AS INTERVAL HOUR(2) TO MINUTE)`
	If an error occurs for this CAST statement also, Teradata Database treats the character value as a time zone string.

Character that is not CHARACTER SET UNICODE	`TRANSLATE(`*`expression`*` USING `*`source_repertoire_name_TO_Unicode`*`)` where *source_repertoire_name* is the server character set of *expression*. The translated value is then processed as above for a character value with CHARACTER SET UNICODE.
other	An error is returned.

Note: There is a general restriction that in Numeric-to-Interval conversions, the INTERVAL type must have only one DateTime field. However, this restriction is not an issue when implicitly converting the *expression* of an AT clause because the conversion is done with two CAST statements.

If the conversion to INTERVAL HOUR(2) TO MINUTE results in a value that is not between INTERVAL -'12:59' HOUR TO MINUTE and INTERVAL '14:00' HOUR TO MINUTE, an error is returned.

You can specify two kinds of time zone strings in the AT [TIME ZONE] *time_zone_string* clause:

- Time zone strings that do not follow separate daylight savings time (DST) and standard time zone displacements from Coordinated Universal Time (UTC) time.
- Time zone strings that follow different DST and standard time zone displacements from UTC time.

The following time zone strings are supported:

Strings that do not follow separate DST and standard time zone displacements		
• 'GMT' • 'GMT+1' • 'GMT+10' • 'GMT+11' • 'GMT+11:30' • 'GMT+12' • 'GMT+13' • 'GMT+14' • 'GMT+2' • 'GMT+3' • 'GMT+3:30'	• 'GMT+4' • 'GMT+4:30' • 'GMT+5' • 'GMT+5:30' • 'GMT+5:45' • 'GMT+6' • 'GMT+6:30' • 'GMT+7' • 'GMT+8' • 'GMT+8:45' • 'GMT+9' • 'GMT+9:30'	• 'GMT-1' • 'GMT-10' • 'GMT-11' • 'GMT-2' • 'GMT-3' • 'GMT-4' • 'GMT-5' • 'GMT-6' • 'GMT-6:30' • 'GMT-7' • 'GMT-8'

Strings that follow different DST and standard time zone displacements		
• 'Africa Egypt' • 'Africa Morocco' • 'Africa Namibia' • 'America Alaska' • 'America Aleutian' • 'America Argentina' • 'America Atlantic' • 'America Brazil' • 'America Central' • 'America Chile' • 'America Cuba' • 'America Eastern' • 'America Mountain' • 'America Newfoundland' • 'America Pacific' • 'America Paraguay' • 'America Uruguay'	• 'Asia Gaza' • 'Asia Iran' • 'Asia Iraq' • 'Asia Irkutsk' • 'Asia Israel' • 'Asia Jordan' • 'Asia Kamchatka' • 'Asia Krasnoyarsk' • 'Asia Lebanon' • 'Asia Magadan' • 'Asia Omsk' • 'Asia Syria' • 'Asia Vladivostok' • 'Asia West Bank' • 'Asia Yakutsk' • 'Asia Yekaterinburg'	• 'Australia Central' • 'Australia Eastern' • 'Australia Western' • 'Europe Central' • 'Europe Eastern' • 'Europe Kaliningrad' • 'Europe Moscow' • 'Europe Samara' • 'Europe Western' • 'Indian Mauritius' • 'Mexico Central' • 'Mexico Northwest' • 'Mexico Pacific' • 'Pacific New Zealand' • 'Pacific Samoa'

Teradata Database resolves the time zone string and calculates the time zone displacement for the session or requested query.

Note: Teradata Database will automatically adjust the time zone displacement to account for the start or end of daylight saving time only if you specify a time zone using a time zone string that follows different DST and standard time zone displacements. GMT format strings represent time zone strings that follow only one standard time and does not have a separate daylight saving time. For example, the time zone string 'GMT+5:30' can be used for India in order to use the displacement interval 5:30, which is applicable all year around.

Teradata Database resolves the time zone string based on the rules and time zone displacement information stored in the system UDF (user-defined function), GetTimeZoneDisplacement.

If the time zone strings provided by Teradata do not meet your requirements, you may add new time zone strings or modify the existing time zone strings by modifying or adding new rules to the GetTimeZoneDisplacement UDF. For details, see "GetTimeZoneDisplacement" on page 868.

You can also use the AT clause to explicitly specify a time zone in the following cases:

- With the following built-in functions:
  - "CURRENT_DATE" on page 351.
  - "CURRENT_TIME" on page 357.
  - "CURRENT_TIMESTAMP" on page 361.
  - "DATE" on page 367.
  - "TIME" on page 378.

  Note: If you specify these built-in functions with an AT LOCAL clause, the value returned depends on the setting of the DBS Control flag TimeDateWZControl.
- When converting DateTime data types using the CAST function or Teradata conversion syntax. You can specify the time zone used for the CAST or conversion as the source time zone, a specific time zone displacement or time zone string, or the current session time zone. For more information, see Chapter 13: "Data Type Conversions."
- With the EXTRACT function to specify a time zone for the source expression before extracting the fields.

For more information about time zones, see "DateTime and Interval Data Types" in *SQL Data Types and Literals*.

Related Topics

For more information on...	See...
Setting session time zones	SET TIME ZONE, CREATE USER, MODIFY USER in *SQL Data Definition Language*.
System time zone settings	"System TimeZone Hour" and "System TimeZone Minute" in *Utilities*.
Automatic adjustment of the system time to account for DST	"SDF file" and "Teradata Locale Definition Utility (tdlocaledef)" in *Utilities*.

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.

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.

Example 1: *date_time_primary*

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

```
SELECT CURRENT_TIME;
```

Example 2: *date_time_term* With an integer numeric Time Zone Specifier

In this example, the *date_time_primary* is a built-in time function and the time zone displacement is specified by the negative integer numeric value for Pacific Standard Time.

```
SELECT CURRENT_TIME AT TIME ZONE -8;
```

Example 3: *date_time_term* With a scaled decimal Time Zone Specifier

In this example, the *date_time_primary* is a built-in time function and the time zone displacement is specified by the scaled decimal value for Venezuela's time zone.

```
SELECT CURRENT_TIME AT TIME ZONE -4.5;
```

Example 4: *date_time_term* With an 'hh:mm' string Time Zone Specifier

In this example, the *date_time_primary* is a built-in time function and the time zone displacement is specified by the negative 'hh:mm' string for Venezuela's time zone.

```
SELECT CURRENT_TIME AT TIME ZONE -'4:30';
```

Example 5: *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(2) TO MINUTE.

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

Example 6: *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 7: *date_time_term* With a Time Zone String 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 the time zone displacement is based on the time zone string 'America Pacific'.

```
SELECT f1 AT TIME ZONE 'America Pacific'
FROM TS;
```

Example 8: *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 9: *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 10: *date_time_expression* With Subtraction

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

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

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

```
EXPIRATION_DATE - SUBSCRIPTION_INTERVAL
```

Time Zone Sort Order

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

Examples

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

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

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

This statement returns the following results table.

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

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

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

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

This statement returns the following results table.

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

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

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

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

This statement returns the following results table.

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

ANSI Interval Expressions

Purpose

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

Definition

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

interval_expression Syntax

1101A010

interval_term Syntax

```
┬─ ± ─ interval_primary ──────────────────┬
│ interval_term ─┬─ * ─┬─ numeric_factor ─┤
│                └─ / ─┘                  │
└ numeric_term ─ * ─ interval_factor ─────┘
```

FF07D268

numeric_term Syntax

```
┬─ numeric_factor ──────────────────────┬
└ numeric_term ─┬─ * ─┬─ numeric_factor ─┘
                └─ / ─┘
```

FF07D270

numeric_factor Syntax

```
─── ± ─ numeric_primary ───
```

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 822.

Syntax element ...	Specifies ...
date_time_term	a single *date_time_primary* or a *date_time_primary* with a time zone specifier of AT LOCAL, AT [TIME ZONE] *expression*, or AT [TIME ZONE] *time_zone_string*. For more information on *date_time_term*, see "ANSI DateTime Expressions" on page 822.
start	a DateTime value with the following syntax that defines the beginning of a date or time interval: 1101A018 where: • *precision* specifies the permitted range of digits, ranging from one to four. The default precision is two. • *fractional_seconds_precision* specifies the fractional precision for values of SECOND, ranging from zero to six. The default is six. MONTH and SECOND values are only permitted when used without TO *end*.
TO *end*	a DateTime value with the following syntax that defines the end of a date or time interval: 1101A017 where *fractional_seconds_precision* specifies the fractional precision for values of SECOND, ranging from zero to six. The default is six. The value for *end* must be less significant than the value for *start*. If *start* is a YEAR value, then *end* must be a MONTH value.
numeric_factor	a signed *numeric_primary*.
numeric_term	a *numeric_factor* or a *numeric_term* multiplied or divided by a *numeric_factor*.

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

Examples of Interval Expression Components and Their Processing

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

Example of *interval_term*

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

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

This example uses the second definition.

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

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

The *numeric_factor* is 4.

The processing involves the following stages:

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

3 The result is converted to '14-4'.

Example of *numeric_factor*

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

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

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

The processing involves the following stages:

1 The interval is multiplied by 12, giving the result as an interval.

2 The interval result is divided by 5, giving '24-04'.

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

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

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

The processing involves the following stages:

1 The *numeric_factor* is computed, giving the result 2.4, which is truncated to 2 because the value is an integer by default.

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

Example of *interval_term / numeric_factor*

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

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

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

The *numeric_factor* is 3.

The processing involves the following stages:

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

Ten years and three months evaluate to 123 months.

2 The interval total is divided by the *numeric_factor* 3, giving '3-05'.

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

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

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

The *numeric_factor* is 3.

The processing involves the following stages:

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

Ten years and two months evaluate to 122 months.

2. The interval total is divided by the *numeric_factor* 3, giving 40.67 months, which is truncated to 40 because the value is an integer.
3. The interval total is converted back to the appropriate format, giving INTERVAL '3-04'.

Example of *numeric_term * interval_primary*

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

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

The *numeric_term* is 12/5.

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

The processing involves the following stages:

1. The *numeric_term* 12/5 is evaluated, giving 2.4, which is truncated to 2 because the value is an integer by default.
2. The *interval_primary* is multiplied by 2, giving '20-04'.

Example of *numeric_term * ± interval_primary*

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

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

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

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

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

The processing involves the following stages:

1. The *interval_primary* is converted to an exact numeric, or 60 days.
2. The operations indicated in the arithmetic are performed on the operands (which are both numeric at this point), producing an exact numeric result having the appropriate scale and precision.

   In this example, 60 days are subtracted from RACE_DURATION, which is an INTERVAL type of INTERVAL DAY TO SECOND.
3. The numeric result is converted back into the indicated INTERVAL type, DAY TO SECOND.

Example of *interval_expression*

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

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

This example uses the second definition.

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

The *interval_expression* is INTERVAL '125' MONTH.

The *interval_term* is INTERVAL '12' YEAR.

The processing involves the following stages:

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

This example uses the third definition for *interval_expression*.

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

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

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

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

The processing involves the following stages:

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

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

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

Now, determine the difference in minutes:

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

The processing involves the following stages:

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

Rules

The following rules apply to Interval expressions.

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

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

Normalization of Intervals with Multiple Fields

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

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

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

First define the table.

```
CREATE TABLE BillDateTime
(column_1 INTERVAL YEAR
,column_2 INTERVAL YEAR(1) TO MONTH
,column_3 INTERVAL YEAR(2) TO MONTH
,column_4 INTERVAL YEAR(3) TO MONTH );
```

Now insert the value INTERVAL '999-12' YEAR TO MONTH using this INSERT statement.

```
INSERT BillDateTime (column_1, column_4)
VALUES ( INTERVAL '40' YEAR, INTERVAL '999-12' YEAR TO MONTH );
```

The result is an overflow error because the valid range for INTERVAL YEAR(3) TO MONTH values is -'999-11' to '999-11'.

You might expect the value '999-12' to work, but it fails because the Parser normalizes it to a value of '1000-00' YEAR TO MONTH. Because the value for year is then four digits, an overflow occurs and the operation fails.

Arithmetic Operators and ANSI DateTime and Interval Data Types

Operations on ANSI DateTime and Interval values can include the scalar arithmetic operators +, -, *, and /. However, the operators are only valid on specific combinations of DateTime and Interval values.

Arithmetic Operators and Result Types

The following arithmetic operations are permitted for DateTime and Interval data types:

First Value Type	Operator	Second Value Type	Result Type
DateTime	-	DateTime	Interval
DateTime	+	Interval	DateTime
DateTime	-	Interval	DateTime
Interval	+	DateTime	DateTime
Interval	+	Interval	Interval
Interval	-	Interval	Interval
Interval	*	Number	Interval
Interval	/	Number	Interval
Number	*	Interval	Interval

Adding or Subtracting Numbers from DATE

Teradata SQL extends the ANSI SQL:2011 standard to allow the operations of adding or subtracting a number of days from an ANSI DATE value.

Teradata SQL treats the number as an INTERVAL DAY value.

For more information, see "DATE and Integer Arithmetic" on page 842.

Calculating the Difference Between Two DateTime Values

Teradata Database calculates the interval difference between two DATE, TIME or TIMESTAMP values according to the ANSI SQL standard. Units smaller than the unit of the result are ignored when calculating the interval value.

For example, when computing the difference in months for two DATE values, the day values in each of the two operands are ignored. Similarly when computing the difference in hours for two TIMESTAMP values, the minutes and the seconds values of the operands are ignored.

Example 1

The following query calculates the difference in days between the two DATE values.

```
SELECT (DATE '2007-05-10' - DATE '2007-04-28') DAY;
```

The result is the following:

```
(2007-05-10 - 2007-04-28) DAY
-----------------------------
12
```

The following query calculates the difference in months between the two DATE values.

```
SELECT (DATE '2007-05-10' - DATE '2007-04-28') MONTH;
```

The result is the following:

```
(2007-05-10 - 2007-04-28) MONTH
-------------------------------
1
```

There is a difference of 12 days between the two dates, which does not constitute one month. However, Teradata Database ignores the day values during the calculation and only considers the month values, so the result is an interval of one month indicating the difference between April and May.

Example 2: Add Interval to DATE

The following example adds an Interval value to a DateTime value:

```
CREATE TABLE Subscription
(id CHARACTER(13)
,subscribe_date DATE
,subscribe_interval INTERVAL MONTH(4));

INSERT Subscription (subscribe_date, subscribe_interval)
VALUES (CURRENT_DATE, INTERVAL '24' MONTH);

SELECT subscribe_date + subscribe_interval FROM Subscription;
```

The result is a DateTime value.

Aggregate Functions and ANSI DateTime and Interval Data Types

DateTime Data Types

The following aggregate functions are valid for ANSI SQL:2011 DateTime types.

For this function ...	The result is ...	For more information, see ...
AVG(*arg*)	the type of the argument.	"AVG" on page 36.

For this function ...	The result is ...	For more information, see ...
MAX(*arg*)	the type of the argument, based on the comparison rules for DateTime types.	"MAX" on page 57.
MIN(*arg*)		"MIN" on page 60.
COUNT(*arg*)	INTEGER, if the mode is Teradata.	"COUNT" on page 42.
	DECIMAL(*n*,0), if the mode is ANSI, where: • *n* is 15 if MaxDecimal in DBSControl is 0, 15, or 18. • *n* is 38 if MaxDecimal in DBSControl is 38.	

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 36.
COUNT(*arg*)	INTEGER, if the mode is Teradata.	"COUNT" on page 42.
	DECIMAL(*n*,0), if the mode is ANSI, where: • *n* is 15 if MaxDecimal in DBSControl is 0, 15, or 18. • *n* is 38 if MaxDecimal in DBSControl is 38.	
MAX(*arg*)	the type of the argument, based on the comparison rules for DateTime types.	"MAX" on page 57.
MIN(*arg*)		"MIN" on page 60.
SUM(*arg*)	the type of the argument.	"SUM" on page 102.

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

The values reflect the time when the request started and do not change during the duration of the request.

Example

The following example uses the CURRENT_DATE DateTime function:

```
SELECT INTERVAL '20' YEAR + CURRENT_DATE;
```

Related Topics

For more information on ...	See ...
CURRENT_DATE	"CURRENT_DATE" on page 351
CURRENT_TIME	"CURRENT_TIME" on page 357
CURRENT_TIMESTAMP	"CURRENT_TIMESTAMP" on page 361
EXTRACT	"EXTRACT" on page 864

Teradata Date and Time Expressions

Teradata SQL provides a data type for DATE values and stores TIME values as encoded numbers with type REAL. This is a Teradata extension of the ANSI SQL:2011 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 818 and "Scalar Operations on ANSI SQL:2011 DateTime and Interval Values" on page 821.

DATE and Integer Arithmetic

The following arithmetic functions can be performed with date and an integer (INTEGER is interpreted as a number of days):

- `DATE + INTEGER`
- `INTEGER + DATE`
- `DATE - INTEGER`

These expressions are not processed as simple addition or subtraction, but rather as explained in the following process:

1. The encoded date value is converted to an intermediate value which is the number of days since some system-defined fixed date.
2. The integer value is then added or subtracted, forming another value as number of days, since the fixed base date.
3. The result is converted back to a date, valid in the Gregorian calendar.

DATE and Date Arithmetic

The DATE - DATE expression is not processed as a simple subtraction, but rather as explained in the following process:

1. The encoded date values are converted to intermediate values which are each the number of days since a system-defined fixed date.
2. The second of these values is then subtracted from the first, giving the number of days between the two dates.
3. The result is returned as if it were in the ANSI SQL:2011 form INTERVAL DAY, though the value itself is an integer.

Other arithmetic operations on date values may provide results, but those results are not meaningful.

Example

DATE/2 provides an integer result, but the value has no meaning.

There are no simple arithmetic operations that have meaning for time values. The reason is that a time value is simply a real number with time encoded as:

```
(HOUR*10000 + MINUTE*100 + SECOND)
```

where SECOND may include a fractional value.

Scalar Operations on Teradata DATE Values

The operations of addition and subtraction are allowed as follows, where integer values represent the number of days:

Argument 1	Operation	Argument 2	Result
DATE	+	INTEGER	DATE
DATE	-	INTEGER	DATE
INTEGER	+	DATE	DATE
DATE	-	DATE	INTEGER

Adding 90 days, for example, is not identical to adding 3 months, because of the varying number of days in months.

Also, adding multiples of 365 days is not identical to adding years because of leap years.

Note that scalar operations on Teradata DATE expressions are performed using ANSI SQL:2011 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:2011 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 850.

EXTRACT Function

Use the EXTRACT function to get the year, month, or day from a date. The result has INTEGER data type.

For details, see "EXTRACT" on page 864.

LAST_DAY

Purpose

Returns the date of the last day of month that contains *date_timestamp_value*.

Syntax

```
┬──────────────────┬─ LAST_DAY ── ( date_timestamp_value ) ──────────────────
└─ TD_SYSFNLIB. ──┘
```

1101A898

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
date_timestamp_value	a data or timestamp argument. If *date_timestamp_value* is NULL, NULL is returned

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

LAST_DAY is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE

Since TIMESTAMP values are stored in UTC time within the database and lack a time zone, the session time zone is used to interpret the time stamp value within the function. For TIMESTAMP WITH TIME ZONE values, the time zone component is used to interpret the time stamp value within the function.

Result Type

The return data type is DATE.

Example 1

The following query:

```
SELECT LAST_DAY(DATE '2009-12-20');
```

returns the result 09/12/31.

Example 2

The following query:

```
SELECT LAST_DAY(TIMESTAMP '2009-08-25 10:14:59');
```

returns the result 09/08/31.

Example 3

The following query:

```
SELECT LAST_DAY(TIMESTAMP '2009-06-15 10:14:59-8:00');
```

returns the result 09/06/30.

NEXT_DAY

Purpose

Returns the date of the first weekday (*day_value*) that is later than the date specified by *date_timestamp_value*.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── NEXT_DAY ( date_timestamp_value, day_value ) ──────────────────────
```

1101A899

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
date_timestamp_value	a date or timestamp argument. If *date_timestamp_value* is NULL, NULL is returned.
day_value	a character argument. *day_value* can be any day of the week or its 3 character abbreviation. Valid values are: • 'SUNDAY' or 'SUN' • 'MONDAY' or 'MON' • 'TUESDAY' or 'TUE' • 'WEDNESDAY' or WED' • 'THURSDAY' or 'THU' • 'FRIDAY' or 'FRI' • 'SATURDAY', or 'SAT' The day of week is not case sensitive. Any characters that follow a valid abbreviation are ignored. If *day_value* is NULL, NULL is returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

NEXT_DAY is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *date_timestamp_value* = DATE or TIMESTAMP(0)
- *day_value* = VARCHAR

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Since TIMESTAMP values are stored in UTC time within the database and lack a time zone, the session time zone is used to interpret the time stamp value within the function. For TIMESTAMP WITH TIME ZONE values, the time zone component is used to interpret the time stamp value within the function.

Result Type

The return data type is DATE.

Example 1

The following query:

```
SELECT NEXT_DAY(DATE '2009-12-20', 'TUESDAY');
```

returns the result 09/12/22 since 12/22 is the next Tuesday after Sunday, Dec 20, 2009.

Example 2

The following query:

```
SELECT NEXT_DAY(DATE '2009-12-20', 'FRIDAY');
```

returns the result 09/12/25 since 12/25 is the next Friday after Sunday, Dec 20, 2009.

Example 3

The following query:

```
SELECT NEXT_DAY(DATE '2009-12-25', 'FRIDAY');
```

returns the result 10/01/01 since Jan 1, 2010 is the next Friday after Friday, Dec 25, 2009.

MONTHS_BETWEEN

Purpose

Returns the number of months between *date_timestamp_value1* and *date_timestamp_value2*.

Syntax

```
┬──────────────────┬─ MONTHS_BETWEEN ── ( date_timestamp_value1, date_timestamp_value2 ) ──
└─ TD_SYSFNLIB. ───┘
```

1101A900

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
date_timestamp_value1	a date or timestamp argument. If *data_timestamp_value1* is NULL, NULL is returned.
date_timestamp_value2	a date or timestamp argument. If *data_timestamp_value2* is NULL, NULL is returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

MONTHS_BETWEEN is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE

The two argument to MONTHS_BETWEEN can be one of the above three data types, but the two arguments must be the same data type. For example, you cannot pass a DATE as the first parameter and a TIMESTAMP as the second parameter.

Since TIMESTAMP values are stored in UTC time within the database and lack a time zone, the session time zone is used to interpret the time stamp value within the function. For

TIMESTAMP WITH TIME ZONE values, the time zone component is used to interpret the time stamp value within the function.

If *date_timestamp_value2* is:

- Later than *date_timestamp_value1*, the result is negative.
- Earlier than *date_timestamp_value1*, the result is positive.

If *date_timestamp_value1* and *date_timestamp_value2* are either the same days of the month or both last days of months, the result is always an integer. Otherwise, MONTHS_BETWEEN calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *date_timestamp_value1* and *date_timestamp_value2*. The Gregorian calendar is supported.

Timestamps without a time zone are compared in the local time zone. This matters solely for the purpose of determining if two timestamps reside on the same day or are both the last day of a month. Timestamps with a time zone are converted to UTC before being compared in order to ensure the times are in the same zone.

Result Type

The return data type is NUMBER.

Example 1

The following query:

```
SELECT MONTHS_BETWEEN(DATE'1995-02-02', DATE'1995-01-01');
```

returns the result 1.0322580645161290322580645161290322580 6. The MONTHS_BETWEEN function calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *date_timestamp_value1* and *date_timestamp_value2*.

Example 2

The following query:

```
SELECT MONTHS_BETWEEN(DATE'2008-09-25', DATE'2008-09-25');
```

returns the result 0.0 since *date_timestamp_value1* and *date_timestamp_value2* are both the same date.

ADD_MONTHS

Purpose

Adds an integer number of months to a DATE or TIMESTAMP expression and normalizes the result.

Date Syntax

```
—— ADD_MONTHS — (date_expression, integer_expression) —
```

FF07D202

Timestamp Syntax

```
—— ADD_MONTHS — (timestamp_expression, integer_expression) —
```

FF07D208

where:

Syntax element ...	Specifies ...
date_expression	one of the following, to which *integer_expression* months are to be added: • A DATE value enclosed in apostrophes • A DATE literal • The CURRENT_DATE keyword • The DATE keyword • A UDT that has an implicit cast that casts between the UDT and a character or DATE type. CURRENT_DATE and DATE specify the current system DATE value.
timestamp_expression	one of the following, to which *integer_expression* months are to be added: • A TIMESTAMP literal • The CURRENT_TIMESTAMP keyword • A UDT that has an implicit cast that casts between the UDT and a character or TIMESTAMP type. CURRENT_TIMESTAMP specifies the current system TIMESTAMP value.
integer_expression	the number of integer months to be added to *date_expression* or *timestamp_expression*.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 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 not valid dates are normalized to ensure that all reported dates are valid.

Support for UDTs

IF this argument is a UDT ...	THEN Teradata Database performs implicit type conversion if the UDT has an implicit cast that casts between the UDT and any of the following predefined types ...
date_expression	• Character • Date • Timestamp
timestamp_expression	
integer_expression	Numeric

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including ADD_MONTHS, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see "Implicit Type Conversions" on page 579.

Scalar Arithmetic on Months Issues

Consistent handling of a target month having fewer days than the month in the source date is an important issue for scalar arithmetic on month intervals because the concept of a month has no fixed definition.

All scalar function operations on dates use the Gregorian calendar. Peculiarities of the Gregorian calendar ensure that arithmetic operations such as adding 90 days (to represent three months) or 730 days (to represent two years) to a DATE value generally do not provide the desired result. For more information, see "Gregorian Calendar Rules" on page 827.

The ADD_MONTHS function uses an algorithm that lets you add or subtract a number of months to a *date_expression* or *timestamp_expression* and to obtain consistently valid results.

When deciding whether to use the Teradata SQL ADD_MONTHS function or ANSI SQL:2011 DateTime interval arithmetic, you are occasionally faced with choosing between

returning a result that is valid, but probably neither desired nor expected, or not returning any result and receiving an error message.

A third option that does not rely on system-defined functions is to use the Teradata Database-defined Calendar view for date arithmetic. For information, see "CALENDAR View" in the *Data Dictionary* book.

Normalization Behavior of ADD_MONTHS

The standard approach to interval month arithmetic is to increment MONTH and YEAR values as appropriate and retain the source value for DAY. This is a problem for the case when the target DAY value is smaller than the source DAY value from the source date.

For example, what approach should be taken to handle the result of adding one MONTH to a source DATE value of '1999-01-31'? Using the standard approach, the answer would be '1999-02-31', but February 31 is not a valid date.

The behavior of ADD_MONTHS is equivalent to that of the ANSI SQL:2011 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:2011 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:2011 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 854 with the results of "Example 7" on page 855.

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:2011 DateTime interval arithmetic instead. For more information, see "ANSI Interval Expressions" on page 830.

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 854.

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 854.

You might expect it to return '1999-01-30', which is the source date in that example, but it does not.

```
SELECT ADD_MONTHS ('1999-02-28' , -1);
```

ADD_MONTHS returns the result 1999-01-28.

The function performs as designed and this result is not an error, though it might not be what you would expect from reading "Example 5."

Example 8

You might expect the following statement to return '1999-03-31', but it does not.

```
SELECT ADD_MONTHS ('1999-02-28' , 1);
```

ADD_MONTHS returns the result 1999-03-28.

Example 9

You might expect the following statement to return '1999-03-31', but it does not.

```
SELECT ADD_MONTHS ('1999-04-30' , -1);
```

ADD_MONTHS returns the result 1999-03-30.

Example 10

You might expect the following statement to return '1999-05-31', but it does not.

```
SELECT ADD_MONTHS ('1999-04-30' , 1);
```

ADD_MONTHS returns the result 1999-05-30.

OADD_MONTHS

Purpose

Adds a specified *date_timestamp_value* to a specified *num_months* and returns the resulting date.

Syntax

```
┬──────────────────┬── OADD_MONTHS ── ( date_timestamp_value, num_months ) ──────────────
└── TD_SYSFNLIB. ──┘
```

1101A901

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
date_timestamp_value	a date or timestamp argument. If *date_timestamp_value* is NULL, NULL is returned.
num_months	a numeric argument. If *num_months* is NULL, NULL is returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

OADD_MONTHS is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *date_timestamp_value* = DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE
- *num_months* = INTEGER

For the *num_months* argument, you can also pass values with data types that can be converted to INTEGER using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *date_timestamp_value* argument.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Since TIMESTAMP values are stored in UTC time within the database and lack a time zone, the session time zone is used to interpret the time stamp value within the function. For TIMESTAMP WITH TIME ZONE values, the time zone component is used to interpret the time stamp value within the function.

If the day component of *date_timestamp_value* is the last day of the month, or if the resulting month has fewer days than the day component of *date_timestamp_value*, OADD_MONTHS returns the last day of the resulting month. Otherwise, OADD_MONTHS returns a value that has the same day component as *date_timestamp_value*.

The difference between OADD_MONTHS and ADD_MONTHS is that if a month is added to an end-of-month date in OADD_MONTHS, the function always returns an end-of-month date. The following queries illustrate the difference between ADD_MONTHS and OADD_MONTHS:

- This query:

  ```
  SELECT ADD_MONTHS ('2008-02-29', 1);
  ```

  returns '08/03/29'
- This query:

  ```
  SELECT OADD_MONTHS ('2008-02-29', 1);
  ```

  returns '08/03/31'

Result Type

OADD_MONTHS is a scalar function whose return data type is DATE.

Example 1

The following query:

```
SELECT OADD_MONTHS (DATE '2008-02-15', 1);
```

returns the result 08/03/15.

Example 2

The following query:

```
SELECT OADD_MONTHS (DATE '2008-02-20', 2);
```

returns the result 08/04/20.

Example 3

The following query:

```
SELECT OADD_MONTHS (DATE '2008-02-29', 1);
```

returns the result 08/03/31.

Since 29 is the last day in February, March 31 is returned since 31 is the last day in March.

TRUNC(Date)

Purpose

Returns *date_value* with the time portion of the day truncated to the unit specified by *fmt*.

Syntax

```
┌──────────────────────────┐
│                          │
└─ TD_SYSFNLIB. ───────────┘─ TRUNC ── ( date_value ┌──────────┐ ) ──────────────────────────
                                                   └─ , fmt ───┘
```

1101A902

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
date_value	a numeric or timestamp argument.
fmt	a character argument. If *fmt* is NULL, NULL is returned. If *fmt* is omitted, *date_value* is truncated to the nearest day. Sunday is considered the first day of the week. For ISO years, Monday is considered the first day of the week. If *fmt* is not valid, an error is returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

TRUNC(DATE) is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *date_value* = DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE
- *fmt* = VARCHAR CHARACTER SET LATIN

For the *fmt* argument, you can also pass values with data types that can be converted to VARCHAR using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *date_value* argument.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Date Formats for TRUNC

Element	Description
CC SCC	One year greater than the first two digits of a 4-digit year.
SYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IYY IY I	ISO year
MONTH MON MM RM	Month (rounds up on the 16th day)
Q	Quarter (rounds up on the 16th day of the 2nd month of the quarter)
WW	Same day of the week as the 1st day of the year
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour

Element	Description
MI	Minute

Result Type

TRUNC is a scalar function whose return data type is DATE.

Example 1

The following query:

```
SELECT TRUNC(CAST('2003/09/17' AS DATE), 'D') (FORMAT 'yyyy-mm-dd');
```

returns the result 2003-09-14. The date was truncated to the first day of that week.

Example 2

The following query:

```
SELECT TRUNC(CAST('2003/09/17' AS DATE), 'RM') (FORMAT 'yyyy-mm-dd');
```

returns the result 2003-09-01. The date was truncated to the beginning of the month.

ROUND(Date)

Purpose

Returns *date_value* with the time portion of the day rounded to the unit specified by *fmt*.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── ROUND ── ( date_value ┌──────────┐ ) ──────────────────────
                                                │          │
                                                └─ , fmt ──┘
```

1101A903

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
date_value	a date or timestamp argument.
fmt	a character argument. If *fmt* is NULL, NULL is returned. If *fmt* is omitted, *date_value* is rounded to the nearest day. Sunday is considered the first day of the week. For ISO years, Monday is considered the first day of the week. If *fmt* is not valid, an error is returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

ROUND(Date) is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *date_value* = DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE
- *fmt* = VARCHAR CHARACTER SET LATIN

For the *fmt* argument, you can also pass values with data types that can be converted to VARCHAR using the implicit data type conversion rules that apply to UDFs. Implicit type conversion is not supported for the *date_value* argument.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

For date formats for ROUND(Date), see "TRUNC(Date)" on page 859.

Result Type

ROUND is a scalar function whose return data type is DATE.

Example 1

The following query:

```
SELECT ROUND(CAST('2003/09/17' AS DATE), 'D') (FORMAT 'yyyy-mm-dd');
```

returns the result 2003-09-14. The date was rounded to the first day of that week.

Example 2

The following query:

```
SELECT ROUND(CAST('2003/09/17' AS DATE), 'RM') (FORMAT 'yyyy-mm-dd');
```

returns the result 2003-10-01. Since the day is greater than or equal to 16, the date is rounded to the beginning of the next month.

EXTRACT

Purpose

Extracts a single specified full ANSI SQL:2011 field from any DateTime or Interval value, converting it to an exact numeric value.

Syntax

```
EXTRACT ( YEAR            FROM value )
          MONTH
          DAY
          HOUR
          MINUTE
          SECOND
          TIMEZONE_HOUR
          TIMEZONE_MINUTE
```

FF07D144

where:

Syntax element ...	Specifies ...
YEAR	that the integer value for YEAR is to be extracted from the date represented by *value*.
MONTH	that the integer value for MONTH is to be extracted from the date represented by *value*.
DAY	that the integer value for DAY is to be extracted from the date represented by *value*.
HOUR	that the integer value for HOUR is to be extracted from the date represented by *value*.
MINUTE	that the integer value for MINUTE is to be extracted from the date represented by *value*.
TIMEZONE_HOUR	that the integer value for TIMEZONE_HOUR is to be extracted from the date represented by *value*.
TIMEZONE_MINUTE	that the integer value for TIMEZONE_MINUTE is to be extracted from the date represented by *value*.
SECOND	that the integer value for SECOND is to be extracted from the date represented by *value*.
value	an expression that results in a DateTime, Interval, or UDT value.

ANSI Compliance

EXTRACT is partially ANSI SQL:2011 compliant.

ANSI SQL:2011 EXTRACT allows extraction of any field in any DateTime or Interval value. In addition to the ANSI SQL:2011 extract function, Teradata SQL also supports HOUR, MINUTE, or SECOND extracted from a floating point value.

Arguments

IF *value* is ...	THEN ...
a character string expression that represents a date	the string must match the 'YYYY-MM-DD' format.
a character string expression that represents a time	the string must match the 'HH:MI:SS.SSSSSS' format.
a floating point type	*value* must be a time value encoded with the algorithm HOUR * 10000 + MINUTE * 100 + SECOND. Only HOUR, MINUTE, and SECOND can be extracted from a floating point value. Externally created time values can be appropriately encoded and stored in a REAL column to any desired precision if the encoding creates a value representable by REAL without precision loss. Do not store time values as REAL in any new applications. Instead, use the more rigorously defined ANSI SQL:2011 DateTime data types.
a UDT	the UDT must have an implicit cast that casts between the UDT and any of the following predefined types: • Numeric • Character • DateTime To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*. Implicit type conversion of UDTs for system operators and functions, including EXTRACT, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*. For more information on implicit type conversion of UDTs, see "Implicit Type Conversions" on page 579.
not a character string expression or floating point type or UDT	the expression must evaluate to a DateTime or Interval type.

Results

EXTRACT returns an exact numeric value for ANSI SQL:2011 DateTime values.

EXTRACT returns values adjusted for the appropriate time zone if the data type of the argument is TIME or TIMESTAMP.

If no time zone is specified for the argument, then the time zone displacement based on the current session time zone is used; otherwise, the explicit time zone of the argument is used. You can use the AT clause to explicitly specify a time zone for the argument. For details, see "ANSI DateTime Expressions" on page 822.

If you extract SECOND, then if the value has a seconds fraction precision of zero, the result is integer; if the value has a seconds factional prevision of greater than zero, the result is DECIMAL with the scaling as specified for the SECOND field in its data description.

If you extract anything else, the result is INTEGER with 32 bits of precision.

If you extract ...	THEN ...
SECOND	If *value* has a seconds fractional of precision of: • zero, the result is INTEGER. • greater than zero, the result is DECIMAL with the scaling as specified for the SECOND field in its data description.
anything else	the result is INTEGER, with 32 bits of precision.

If *value* is NULL, the result is NULL.

Example 1

The following example returns the year, as an integer, from the current date.

```
SELECT EXTRACT (YEAR FROM CURRENT_DATE);
```

Example 2

Assuming PurchaseDate is a DATE field, this example returns the month of the *date value* formed by adding 90 days to PurchaseDate as an integer.

```
SELECT EXTRACT (MONTH FROM PurchaseDate+90) FROM SalesTable;
```

Example 3

The following returns 12 as an integer.

```
SELECT EXTRACT (DAY FROM '1996-12-12');
```

Example 4

This example returns an error because the character literal does not evaluate to a valid date.

```
SELECT EXTRACT (DAY FROM '1996-02-30');
```

Example 5

The following returns an error because the character string literal does not match the ANSI SQL:2011 date format.

```
SELECT EXTRACT (DAY FROM '96-02-15');
```

If the argument to EXTRACT is a value of type DATE, the value contained is warranted to be a valid date, for which EXTRACT cannot return an error.

Example 6

The following example relates to non-ANSI DateTime definitions. If the argument is a character literal formatted as a time value, it is converted to REAL and processed. In this example, 59 is returned.

```
SELECT EXTRACT (MINUTE FROM '23:59:17.3');
```

Example 7

This example returns the hour, as an integer, from the current time.

```
SELECT EXTRACT (HOUR FROM CURRENT_TIME);
```

Current time is retrieved as the system value TIME, to the indicated precision.

Example 8

The following example returns the seconds as DECIMAL(8,2). This is based on the fractional seconds precision of 2 for CURRENT_TIME.

```
SELECT EXTRACT (SECOND FROM CURRENT_TIME (2));
```

GetTimeZoneDisplacement

Purpose

Returns the rules and time zone displacement information for a specified time zone string.

Syntax

```
── GetTimeZoneDisplacement ── (time_zone_string) ──────────────────────────
                                                                  1101A720
```

where:

Syntax element ...	Specifies ...
time_zone_string	a valid time zone string specified using a VARBYTE data type. For a list of time zone strings supported by Teradata, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. If *time_zone_string* is not valid or unsupported, GetTimeZoneDisplacement returns a value of 1 in the first byte to indicate that the time zone string does not exist.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Result

GetTimeZoneDisplacement returns a string of bytes containing the rules and time zone displacement information for the specified time zone string. The result data type is BYTE. The information returned is:

Byte	Value
First byte	• 1, if the time zone string is not found. That is, the time zone string specified in the input argument is not valid or unsupported. • 0, if the time zone string is found.

Byte	Value
Second byte	• 1, if the time zone string has separate daylight saving time and standard time zone displacements from Coordinated Universal Time (UTC) time. In this case, the next 480 or so bytes store the set of rules describing a valid standard time zone displacement, daylight saving time zone displacement, and the start and end time for daylight saving time. A maximum of 6 rules are stored for each time zone string. • 0, if the time zone string does not have separate daylight saving time and standard time zone displacements from UTC time. In this case, the next 4 bytes store the time zone displacement hour and minute values.

Usage Notes

GetTimeZoneDisplacement is a system user-defined function (UDF) that Teradata Database invokes internally to resolve a time zone string specified in an SQL statement or Specification for Data Formatting (SDF) file. You do not invoke this function directly; however, you can modify this UDF to add new time zone strings or add or modify the rules of an existing time zone string.

Limitation on the Use of TimeZone Strings

When using standard Teradata Database system time zone strings, no time zone rules are enforced for the years 1986 and before, for example, daylight saving time (DST) shifts. Valid years for Teradata Database standard time zone strings are 1987 through 9999.

If Teradata Database standard time zone strings do not meet your requirements, you can add a new custom time zone string or modify an existing string by modifying or adding new rules to GetTimeZoneDisplacement.

Note: When time zone strings are used for the locales that observe DST, but time zone offset values are not included with time values, Teradata Database cannot determine whether time values within this ambiguous hour should be interpreted and treated as DST or as standard time. In these cases, the DBS Control setting for DisplacementOnOverlap determines whether these times are assumed by Teradata Database to have a DST or standard time zone offset.

This issue occurs only when time zone strings are used for DST locales, when time values do not include a time zone offset value, and when the DBS Control setting for TimeDateWZControl is set to 3 (date and time values without time zone offset information are stored using system local time).

For more information on DisplacementOnOverlap TimeDateWZControl, see the DBS Control chapter in *Utilities: Volume 1 (A-K)*.

Adding or Modifying Time Zone Strings

Teradata Database provides a set of time zone strings that represent commonly used time zones. For a list of supported time zone strings, see "AT LOCAL and AT TIME ZONE Time Zone Specifiers" on page 823. The GetTimeZoneDisplacement UDF stores and maintains

these time zone strings and the related rules for converting between UTC and the time in the local time zone.

If the supplied time zone strings do not meet your requirements, you may add or modify the time zone strings by modifying the GetTimeZoneDisplacement UDF, which is located in the SYSLIB database. The source code for the UDF is available as part of the DBS package and is located at /tdbms/etc/dem/src.

To define new time zone strings or add or modify the rules of an existing time zone string:

1. Make a backup copy of the existing GetTimeZoneDisplacement UDF.
2. To modify an existing time zone string:
   a. Find the time zone string entry in the TZ_DST structure of the GetTimeZoneDisplacement UDF.
   b. Modify the rules and information associated with the time zone string entry or add new rules to the entry.
3. To add a new time zone string:
   a. Create a new entry in the TZ_DST structure for the new time zone string and its related rules.
   b. Place the new time zone string entry in the correct alphabetical position within the TZ_DST structure.
4. Recompile the UDF using the REPLACE FUNCTION statement. For more information, see "CREATE FUNCTION (External Form)/ REPLACE FUNCTION (External Form)" in *SQL Data Definition Language*. For example:

```
Database SYSLIB;
DROP FUNCTION GetTimeZoneDisplacement;

REPLACE FUNCTION GetTimeZoneDisplacement
                 (tzstringinfo VARBYTE(130))
RETURNS BYTE(340)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL; //or specify the path of the new source code.
```

The TZ_DST Structure

The TZ_DST structure is an array of TZwithDST elements where each element describes a time zone string and its related rules. The definition of the TZwithDST structure is:

```
typedef struct TZwithDST
{
    CHARACTER_LATIN  tzstring[TZSTRINGSIZE];
    int              number_of_rules;
    DSTRules         TZRules[TZRulesEntries];
    SMALLINT         Standardtzdispl_hour;
    SMALLINT         Standardtzdispl_minute;
} TZwithDST;
```

where:

Field	Description
`tzstring`	The name of the time zone string. For example, "America Pacific." The maximum length of a time zone string is 130 bytes.
`number_of_rules`	The number of rules related to this time zone string. A maximum of 6 rules is allowed for each time zone string.
`TZRules`	An array where each DSTRules element describes a rule. These rules are used to calculate the time zone displacement for the time zone string.
`Standardtzdispl_hour`	The standard time zone displacement hour.
`Standardtzdispl_minute`	The standard time zone displacement minute.

Each DSTRules element of the TZRules array describes a rule for the time zone string. The definition of the DSTRules structure is:

```
typedef struct DSTRules
{
    startendDSTInfo  startDST;
    startendDSTInfo  endDST;
    yearDisplInfo    validyrs;
} DSTRules;
```

where:

Field	Description
`startDST`	Specifies the date and time when daylight saving time (DST) starts.
`endDST`	Specifies the date and time when daylight saving time ends.
`validyrs`	Specifies the years in which the DST start and end dates apply. The following information related to this year range is included: • start_year - the year when these DST rules start. • end_year - the year when these DST rules end. • Standardtzdispl_hour - the standard time zone displacement hour. • Standardtzdispl_minute - the standard time zone displacement minute. • DSTtzdispl_hour - the time zone displacement hour for daylight saving time. • DSTtzdispl_minute -the time zone displacement minute for daylight saving time.

You can specify the following for startDST and endDST. Enter zero if a field is not applicable.

Field	Description
rule_type	Indicates how the start and end date for DST is specified. The valid values are: • 0 - No DST start or end information is specified. The standard time zone displacement is used. • 1 - DST starts or ends on the specified fixed date. The date is specified by the month and day_of_month fields. • 2 - DST starts or ends on the 1st, 2nd, or 3rd weekday of the month as indicated by the month, day_of_week, and week_of_month fields. • 3 - DST starts or ends on the 2nd to the last, 3rd to the last, or the last weekday of the month as indicated by the month, day_of_week, and week_of_month fields. • 4 - DST starts or ends on the next weekday on or immediately after the date specified in the day_of_month field. The month and weekday are specified in the month and day_of_week fields. For example, for time zone string 'America Pacific', the start date rule is the first Sunday after March 8th, which gives us March 14th for the year 2010.
month	The month when DST starts or ends. Valid values are 0- 12. This field is used for rule_type 1, 2, 3, and 4. For example, for time zone string 'America Pacific', the start date rule is the first Sunday after March 8; therefore, this field has a value of 3 in the startDST structure to represent March.
day_of_month	If rule_type is 1, this field specifies the day of the month when DST starts or ends. For example, if DST ends at 12:00 am local time on August 21, this field contains the value 21 in the endDST structure. If rule_type is 4, DST starts or ends on the next weekday on or immediately after the date specified by this field. For example, for time zone string 'America Pacific', the start date rule is the first Sunday after March 8; therefore, this field has a value of 8 in the startDST structure. When rule_type is 0, 2 or 3, this field is not used and the value is 0.
day_of_week	The valid values are 0-7 representing the weekdays Sunday-Saturday. This field is used for rule_type 2, 3 and 4. For example, for time zone string 'America Pacific', the start date rule is the first Sunday after March 8; therefore, this field has a value of 0 in the startDST structure to represent Sunday.
week_of_month	The valid values are 1, 2, 3, 4, 5, -1, and -2 representing the 1st, 2nd, 3rd, 4th, 5th, last, and second to the last weekday of the month. This field is used for rule_type 2 and 3. For example, for time zone string 'Europe Azores', the start date rule is the last Sunday in March; therefore, this field has a value of -1 in the startDST structure to represent the last week of the month.
loctime	The local time when DST starts or ends. For example, "02:00:00" indicates that DST starts or ends at 2:00 am local time.

Example

Assume that you want to add a new time zone string 'Europe Azores', which has one rule with the following time zone displacement information:

- DST starts on the last Sunday in March at 12:00 am local time.
- DST ends on the last Sunday in October at 1:00 am local time.
- The standard time zone offset from UTC is -1.
- The daylight saving time offset from UTC is 0.
- The start year for the rule is 2009.
- The end year for the rule is 2010.

Based on this information, the new time zone string entry for 'Europe Azores' is:

```
{"Europe Azores", 1,                          <= 1 rule defined for 'Europe Azores'
  {{{3,  3, 0, 0, -1, "00:00:00"},            <= Start of rule 1, startDST information
    {3, 10, 0, 0, -1, "01:00:00"},            <= endDST information
    {2009, 2010, -1, 0, 0, 0}},               <= validyrs information
   {{0, 0, 0, 0, 0, "00:00:00"},              <= Start of rule 2
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}},
   {{0, 0, 0, 0, 0, "00:00:00"},              <= Start of rule 3
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}},
   {{0, 0, 0, 0, 0, "00:00:00"},              <= Start of rule 4
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}},
   {{0, 0, 0, 0, 0, "00:00:00"},              <= Start of rule 5
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}},
   {{0, 0, 0, 0, 0, "00:00:00"},              <= Start of rule 6
    {0, 0, 0, 0, 0, "00:00:00"},
    {0, 0, 0, 0, 0, 0}}
  },
  -1, 0                                       <= Standard time zone displacement
},
```

Note that the time zone string entry has space for 6 rules but only one rule is used for the start year 2009 and end year 2010.

You must place the new 'Europe Azores' time zone string in between the 'Australia Western' and 'Europe Central' time zone strings in the TZ_DST structure to maintain the alphabetical order of the structure.

Related Topics

For more information on...	See...
Setting session time zones	SET TIME ZONE, CREATE USER, MODIFY USER in *SQL Data Definition Language*.
System time zone settings	"System TimeZone Hour" and "System TimeZone Minute" in *Utilities*.

For more information on...	See...
Automatic adjustment of the system time to account for daylight saving time	"SDF file" and "Teradata Locale Definition Utility (tdlocaledef)" in *Utilities*.

CHAPTER 16 Export Width Procedures

This chapter describes the stored procedures used to:

- Create a new user-defined export width definition.
- Modify an existing user-defined export width definition.
- Delete a user-defined export width definition.

About Export Widths

The number of characters or bytes the Teradata Database exports can exceed or be less than the number of characters or bytes the client application expects to receive unless the export width is set properly. The export width defines how the system reserves space for each character field in result set rows returned to Teradata clients.

You can set the DBS Control field, Export Width Table ID, to define the default export width the system uses when returning a result set to a client. For more information, see "DBS Control (dbscontrol)" in *Utilities*.

You can override the system-level export width by using the EXPORTWIDTH option in the CREATE USER or MODIFY USER statements. The EXPORTWIDTH option accepts the following values:

Export Definition Name	Description
'Expected'	Provides reasonable default widths for the character data type and client form of use.
'Compatibility'	Enables Unicode data to be processed by applications that are written to process Latin or KANJI1 data.
'Maximum'	Provides maximum default width of the character data type and client form of use.
'*user-defined name*'	Provides user-defined default widths for the character data type and client form of use.
DEFAULT	Uses the setting of the DBS Control field: Export Width Table ID.

You can use the ReplaceExportDefinition stored procedure to:

- Create a new user-defined export definition name and the associated export width rules.
- Update the export width rules associated with an existing user-defined export definition name.

You can use the RemoveExportDefinition stored procedure to delete a user-defined export definition name and the associated export width rules.

For more information about...	See...
export widths and creating user-defined export width definitions	*International Character Set Support*.
specifying a user-level export width setting	"CREATE USER" or "MODIFY USER" in *SQL Data Definition Language*.
setting the system default for the export width	"DBS Control (dbscontrol)" in *Utilities*.

ReplaceExportDefinition

Purpose

Creates or modifies a user-defined export width definition.

Syntax

where:

Syntax element ...	Specifies ...
SYSLIB	the name of the database where the stored procedure is located.
export_definition_name	a user-defined export definition name. You cannot specify 'Expected', 'Compatibility', or 'Maximum'.
export_width_rule_set	a user-defined BYTE string of 40 hexadecimal digits used to define the export width rules associated with *export_definition_name*.
result_message	the output message of the stored procedure.

Authorization

You must have EXECUTE privileges on the stored procedure or on the SYSLIB database.

Argument Types

Expressions passed to this procedure must have the following data types:

- *export_definition_name* = VARCHAR(*n*) CHARACTER SET LATIN, where the maximum value for *n* is 30.
- *export_width_rule_set* = BYTE(20)
- *result_message* = VARCHAR(310) CHARACTER SET UNICODE

You can also pass arguments with data types that can be converted to the above types using Teradata Database implicit data type conversion rules.

Usage Notes

To create a new user-defined export width definition, call the ReplaceExportDefinition stored procedure with the following arguments:

- *export_definition_name* is a LATIN character string that specifies the name of the new export width definition.
- *export_width_rule_set* is a BYTE string of 40 hexadecimal digits, which encode the export width rules for the new export width definition. For details about the export width rules, see "Export Width Rules" on page 878.
- *result_message* is an output parameter that contains the output message of the stored procedure.

ReplaceExportDefinition creates the new export width definition with the specified rules and updates the DBC.ExportWidth table.

To modify the export width rules for an existing user-defined export width definition, call the ReplaceExportDefinition stored procedure with the following arguments:

- *export_definition_name* is a LATIN character string that specifies the name of an existing user-defined export width definition that you want to modify.
- *export_width_rule_set* is a BYTE string of 40 hexadecimal digits, which define the new export width rules for *export_definition_name*. For details about the export width rules, see "Export Width Rules" on page 878.
- *result_message* is an output parameter that contains the output message of the stored procedure.

ReplaceExportDefinition modifies the specified export width definition with the new rules and updates the DBC.ExportWidth table. The revised export width definition takes effect at the next transaction for each affected user. In process transactions continue with the existing export width definition.

Note: You cannot specify 'Expected', 'Compatibility', or 'Maximum' for *export_definition_name*.

Export Width Rules

export_width_rule_set is a BYTE string of 40 hexadecimal digits with the following format:

- The 1st ten digits define the export width for LATIN strings.
- The 2nd ten digits define the export width for UNICODE strings.
- The 3rd ten digits define the export width for KANJISJIS strings.
- The 4th ten digits define the export width for GRAPHIC strings.

For example, if *export_width_rule_set* is '1112211111222232222211121111112222322222'XB, then:

- The 1st ten digits (1112211111) define the export width for LATIN strings.
- The 2nd ten digits (2222322222) define the export width for UNICODE strings.
- The 3rd ten digits (1112111111) define the export width for KANJISJIS strings.

- The 4th ten digits (2222322222) define the export width for GRAPHIC strings.

Each position in a set of 10 digits specify the export width conversion multiplier for a session character set:

This position in a set of 10 digits...	Is the conversion multiplier for....
1st	• any session character set that ends in the string '_0I'. • the session character set 'KATAKANAEBCDIC'.
2nd	any session character set that ends in the string '_0U'.
3rd	any session character set that ends in the string '_0S'.
4th	the session character set 'UTF16'.
5th	the session character set 'UTF8'.
6th	any site-defined session character set with STATEMACHINE EUC1211.
7th	any site-defined session character set with STATEMACHINE EUC1223.
8th	any site-defined session character set with STATEMACHINE S80.
9th	any site-defined session character set with STATEMACHINE S80A1E0.
10th	any site-defined session character set with STATEMACHINE SOSI0E0F.

Each position can have a value of 1, 2, 3, or 4, except the 4th position in each 10 character set (UTF16), which must have a value of either 2 or 4.

For example, if *export_width_rule_set* is '1112211111222232222211121111112222322222'XB:

- When exporting from the LATIN server character set (1st set of 10 digits, 1112211111), for the UTF8 session character set (the 5th position in the 1st set of 10 digits), the export width is 2.
- When exporting from the UNICODE server character set (2nd set of 10 digits, 2222322222), for the UTF8 session character set (the 5th position in the 2nd set of 10 digits), the export width is 3.
- When exporting from the KANJISJIS server character set (3rd set of 10 digits, 1112111111), for the UTF16 session character set (the 4th position in the 3rd set of 10 digits), the export width is 2.

Example

In this example, if MyExportWidth does not exist, ReplaceExportDefinition creates a new export width definition named MyExportWidth with an associated export width rule set of

'1422232221332232322321123111122322323221'XB. If MyExportWidth already exists, then ReplaceExportDefinition replaces the current export width rule set of MyExportWidth with the new export width rule set of '1422232221332232322321123111122322323221'XB.

```
CALL SYSLIB.ReplaceExportDefinition
('MyExportWidth','1422232221332232322321123111122322323221'xb,msg);
```

RemoveExportDefinition

Purpose

Removes a user-defined export width definition.

Syntax

where:

Syntax element ...	Specifies ...
SYSLIB	the name of the database where the stored procedure is located.
export_definition_name	an existing user-defined export definition name. You cannot specify 'Expected', 'Compatibility', or 'Maximum'.
result_message	the output message of the stored procedure.

Authorization

You must have EXECUTE privileges on the stored procedure or on the SYSLIB database.

Argument Types

Expressions passed to this procedure must have the following data types:

- *export_definition_name* = VARCHAR(*n*) CHARACTER SET LATIN, where the maximum value for *n* is 30.
- *result_message* = VARCHAR(100) CHARACTER SET UNICODE

You can also pass arguments with data types that can be converted to the above types using Teradata Database implicit data type conversion rules.

Usage Notes

To delete an existing user-defined export width definition, call the RemoveExportDefinition stored procedure with the following arguments:

- *export_definition_name* is a LATIN character string that specifies the name of the existing export width definition you want to delete.
- *result_message* is an output parameter that contains the output message of the stored procedure.

RemoveExportDefinition deletes the specified export width definition from the DBC.ExportWidth table.

Note: You cannot specify 'Expected', 'Compatibility', or 'Maximum' for *export_definition_name*. In addition, you cannot delete an export width definition if it is assigned to a user.

Example

The following statement deletes the user-defined export width definition named MyExportWidth.

```
CALL SYSLIB.RemoveExportDefinition('MyExportWidth',msg);
```

CHAPTER 17 Hash-Related Functions

Hash-related functions return information about the:

- Primary or fallback AMP that corresponds to a given hash bucket number
- Hash bucket number that corresponds to a given row hash value
- Row hash value for the primary index of a row
- Highest AMP number
- Highest hash bucket number
- Maximum value that can be generated by applying the hash function to an unsigned integer

Use the hash-related functions to identify the statistical properties of the current primary index or secondary index, or to evaluate these properties for other columns to determine their suitability as a future primary index or secondary index. The statistics can help you to minimize hash synonyms and enhance the uniformity of data distribution.

HASHAMP

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

```
── HASHAMP ──( ┌──────────────┐ )──────────
           └─ expression ─┘
                                  HH01A027
```

where:

Syntax element ...	Specifies ...
expression	an optional expression that evaluates to a valid hash bucket number. For information on obtaining a hash bucket number that you can use for *expression*, see "HASHBUCKET" on page 890.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Argument Type and Value

The *expression* argument must evaluate to INTEGER data type where the valid range of values depends on the system setting for the hash bucket size.

IF the hash bucket size is ...	THEN the range of values for *expression* is ...
16 bits	0 to 65535.
20 bits	0 to 1048575.

For information on how to specify the system setting for the hash bucket size, see "DBS Control utility" in *Utilities*.

If *expression* cannot be implicitly converted to an INTEGER, an error is reported.

If *expression* results in a UDT, Teradata Database performs implicit type conversion on the UDT, provided that the UDT has an implicit cast that casts between the UDT and any of the following predefined types:

- Numeric
- Character
- DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including HASHAMP, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 13: "Data Type Conversions."

Result

IF *expression* ...	THEN ...
evaluates to a valid hash bucket number	HASHAMP determines the primary AMP corresponding to the hash bucket and returns the AMP identification number. The result is an INTEGER value that is greater than or equal to zero and less than the maximum number of AMPs in the configuration.
does not appear in the argument list	HASHAMP returns an INTEGER value that is one less than the maximum number of AMPs in the system.
evaluates to NULL	HASHAMP returns NULL.

For information on the hash map that defines the relationship between hash buckets and primary AMPs, see "Reconfiguration utility" in the *Utilities* book.

Examples

The following examples assume a table T with columns column_1, column_2, and an INTEGER column B populated with integer numbers from zero to the maximum number of hash buckets on the system.

```
CREATE TABLE T
   (column_1 INTEGER
   ,column_2 INTEGER
   ,B INTEGER)
UNIQUE PRIMARY INDEX (column_1, column_2);
```

Example 1

If you call HASHAMP without an argument, it returns one less than the maximum number of AMPs on the system.

```
SELECT HASHAMP();
```

Example 2

If you call HASHAMP with an argument of NULL, it returns NULL.

```
SELECT HASHAMP(NULL);
```

Example 3

The following query returns the distribution of the hash buckets among the primary AMPs.

```
SELECT B, HASHAMP (B)
FROM T
ORDER BY 1;
```

Example 4

The following query returns the number of rows on each primary AMP where column_1 and column_2 are to be the primary index of table T.

```
SELECT HASHAMP (HASHBUCKET (HASHROW (column_1,column_2))), COUNT (*)
FROM T
GROUP BY 1
ORDER BY 1;
```

HASHBAKAMP

Purpose

Returns the identification number of the fallback AMP corresponding to the specified hash bucket. If no hash bucket is specified, HASHBAKAMP returns one less than the maximum number of AMPs in the system.

Syntax

```
── HASHBAKAMP ── ( ┌──────────────┐ ) ──────────
                   └ expression ─┘
                                          HH01A028
```

where:

Syntax element ...	Specifies ...
expression	an optional expression that evaluates to a valid hash bucket number. For information on obtaining a hash bucket number that you can use for *expression*, see "HASHBUCKET" on page 890.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Argument Type and Value

The *expression* argument must evaluate to INTEGER data type where the valid range of values depends on the system setting for the hash bucket size.

IF the hash bucket size is ...	THEN the range of values for *expression* is ...
16 bits	0 to 65535.
20 bits	0 to 1048575.

For information on how to specify the system setting for the hash bucket size, see "DBS Control utility" in *Utilities*.

If *expression* cannot be implicitly converted to an INTEGER, an error is reported.

If *expression* results in a UDT, Teradata Database performs implicit type conversion on the UDT, provided that the UDT has an implicit cast that casts between the UDT and any of the following predefined types:

- Numeric
- Character
- DATE

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including HASHBAKAMP, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 13: "Data Type Conversions."

Result

IF *expression* ...	THEN ...
does not appear in the argument list	HASHBAKAMP returns an INTEGER value that is one less than the maximum number of AMPs in the system.
evaluates to NULL	HASHBAKAMP returns NULL.
evaluates to a valid hash bucket number	HASHBAKAMP determines the fallback AMP corresponding to the hash bucket and returns the identification number of the AMP. The result is an INTEGER value that is greater than or equal to zero and less than the maximum number of AMPs in the configuration.

For information on the hash map that defines the relationship between hash buckets and fallback AMPs, see "Reconfiguration utility" in the *Utilities* book.

Examples

The following examples assume a table T with an INTEGER column B populated with integer numbers from zero to the maximum number of hash buckets on the system.

Example 1

If you call HASHBAKAMP without an argument, it returns one less than the maximum number of AMPs on the system.

```
SELECT HASHBAKAMP();
```

Example 2

If you call a HASHBAKAMP function with an argument of NULL, the function returns NULL.

```
SELECT HASHBAKAMP(NULL);
```

Example 3

This query returns the distribution of the hash buckets among the fallback AMPs.

```
SELECT B, HASHBAKAMP (B)
FROM T
ORDER BY 1;
```

HASHBUCKET

Purpose

Returns the hash bucket number that corresponds to a specified row hash value. If no row hash value is specified, HASHBUCKET returns the highest hash bucket number.

Syntax

```
── HASHBUCKET ──( ┬──────────────┬ )──────────
              └ expression ┘
```

HH01A026

where:

Syntax element ...	Specifies ...
expression	an optional expression that evaluates to a valid BYTE(4) row hash value. If *expression* results in a UDT, Teradata Database performs implicit type conversion on the UDT, provided that the UDT has an implicit cast to a predefined byte type. To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*. Implicit type conversion of UDTs for system operators and functions, including HASHBUCKET, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*. For more information on implicit type conversion, see Chapter 13: "Data Type Conversions." For information on obtaining a row hash value that you can use for *expression*, see "HASHROW" on page 893.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 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. The range of values for hash bucket numbers depends on the system setting of the hash bucket size. • If the hash bucket size is 16 bits, the hash bucket numbers can have a value from 0 to 65535. • If the hash bucket size is 20 bits, the hash bucket numbers can have a value from 0 to 1048575.

Using HASHBUCKET to Convert a BYTE Type to an INTEGER Type

When a byte data type is the source type of a conversion using CAST syntax or Teradata Conversion syntax, the target data type must also be a byte type.

To convert a BYTE(1) or BYTE(2) data type to INTEGER, you can use the HASHBUCKET function.

Consider the following table definition:

```
CREATE TABLE ByteData(b1 BYTE(1), b2 BYTE(2));
```

To convert column b1 to INTEGER regardless of the system setting of the hash bucket size, use the following:

```
SELECT HASHBUCKET('00'XB || b1 (BYTE(4))) / ((HASHBUCKET()+1)/65536)
FROM ByteData;
```

To convert column b2 to INTEGER regardless of the system setting of the hash bucket size, use the following:

```
SELECT HASHBUCKET(b2 (BYTE(4))) / ((HASHBUCKET()+1)/65536)
FROM ByteData;
```

Examples

The following examples assume a table T with columns C1 and C2 and possibly other columns.

Example 1

If you call HASHBUCKET without an argument, it returns the maximum hash bucket.

```
SELECT HASHBUCKET();
```

Example 2

If you call a HASHBUCKET function with an argument of NULL, the function returns NULL.

```
SELECT HASHBUCKET(NULL);
```

Example 3

Building on the previous example, you can nest a call to HASHROW within a HASHBUCKET call.

Calling HASHBUCKET (HASHROW (NULL)) returns the 0 hash bucket.

```
SELECT HASHBUCKET(HASHROW(NULL));
```

Example 4

The following example returns the number of rows in each hash bucket where C1 and C2 are to be the primary index of T.

```
SELECT HASHBUCKET (HASHROW (C1,C2)), COUNT (*)
   FROM T
   GROUP BY 1
   ORDER BY 1;
```

Example 5

The results of the following example lists each hash bucket that has one or more rows and its corresponding primary AMP.

```
SELECT HASHAMP (HASHBUCKET (HASHROW (C1, C2))),
   HASHBUCKET (HASHROW (C1,C2))
   FROM T
   GROUP BY 1,2
   ORDER BY 1,2 ;
```

HASHROW

Purpose

Returns the hexadecimal row hash value for an expression or sequence of expressions. If no expression is specified, HASHROW returns the maximum hash code value.

Syntax

```
HASHROW ( [ expression [ , ... ] ] )
```

1101B026

where:

Syntax element ...	Specifies ...
expression	an optional expression or comma-separated list of expressions that can appear in the expression list of the select clause of a SELECT statement; typically a comma-separated list of column names that make up a (potential) index. HASHROW does not support expressions that result in UDT data types.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Result

The resulting row hash value is typed BYTE(4).

IF the argument list is ...	THEN HASHROW ...
empty	returns the maximum hash code value.
an expression that evaluates to NULL	returns '00000000'XB.
a list of expressions where all the expressions evaluate to NULL	
an expression that evaluates to 0, '', ' ', or a similar value	
a valid, non-NULL expression that can appear in the select list of a SELECT statement	evaluates *expression* or the list of *expressions* and applies the hash function on the result. HASHROW returns the resulting row hash value.
a list of expressions that can appear in the select list of a SELECT statement, where some expressions can evaluate to NULL	

Usage Notes

HASHROW is particularly useful for identifying the statistical properties of the current primary index, or to evaluate these properties for other columns to determine their suitability as a future primary index. You can also use these statistics to help minimize hash synonyms and enhance the uniformity of data distribution.

There are a maximum of 4,294,967,295 hash codes available in the system, ranging from '00000000'XB to 'FFFFFFFF'XB.

You can embed a HASHROW call within a HASHBUCKET call. For information on HASHBUCKET, see "HASHBUCKET" on page 890.

Example 1

If you call HASHROW without an argument, it returns 'FFFFFFFF'XB, which is the maximum hash code in the system.

```
SELECT HASHROW();
```

Example 2

The following example returns the average number of rows per row hash, where columns date_field and time_field constitute the primary index of the table eventlog.

```
SELECT COUNT(*) / COUNT(DISTINCT HASHROW (date_field,time_field))
FROM eventlog;
```

If columns date_field and time_field qualify for a unique index, this example returns the average number of rows with the same hash synonym.

Example 3

The following example evaluates the efficiency of changing the decimal format of a numeric field to eliminate synonyms.

Assume that column_1 and column_2 are declared as DECIMAL(2,2).

You can determine the effect of reformatting the columns to DECIMAL(8,6) and DECIMAL(8,4) on hash collisions by submitting these two queries.

```
SELECT COUNT (DISTINCT column_1(DECIMAL(8,6)) ||
column_2(DECIMAL(8,4))
FROM T;

SELECT COUNT (DISTINCT HASHROW (column_1(DECIMAL(8,6)),
column_2 (DECIMAL(8,4)))
FROM T;
```

If the result of the second query is significantly less than the result of the first query, there are a significant number of hash collisions. That is, the closer the second result is to the first value indicates elimination of more hash synonyms.

CHAPTER 18 LOB Functions

This chapter describes functions that are used with BLOB (Binary Large Object) and CLOB (Character Large Object) data types.

EMPTY_BLOB

Purpose

Returns an empty BLOB (Binary Large Object), that is, one that contains 0 bytes.

Syntax

```
┬──────────────┬─ EMPTY_BLOB ( ) ──────────────────────────────────
└─ TD_SYSFNLIB. ─┘
```

1101A927

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

EMPTY_BLOB is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types

EMPTY_BLOB takes zero input arguments.

Result Type

EMPTY_BLOB is a scalar function whose return value data type is BLOB AS LOCATOR.

Example

The following query:

```
UPDATE employee SET picture = EMPTY_BLOB();
```

updates the picture column with an empty BLOB.

EMPTY_CLOB

Purpose

Returns an empty CLOB (Character Large Object), that is, one that contains 0 bytes.

Syntax

```
┬──────────────┬─ EMPTY_CLOB ( ) ──────────────────────────────────────
└─ TD_SYSFNLIB. ─┘
```

1101A928

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

EMPTY_CLOB is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types

EMPTY_CLOB takes zero input arguments.

Result Type

EMPTY_CLOB is a scalar function whose return value data type is CLOB AS LOCATOR in the LATIN character set.

Example

The following query:

```
UPDATE student SET essay = EMPTY_CLOB();
```

updates the essay column with an empty CLOB.

CHAPTER 19 Logical Predicates

This chapter describes SQL logical predicates.

Logical predicates are also known as comparison operators and conditional expressions. The ANSI SQL standard calls logical predicates *search conditions*. For general information on comparison operators, see Chapter 10: "Comparison Operators and Functions."

Logical Predicates

A logical predicate tests an operand against one or more other operands to evaluate to a logical (Boolean TRUE, FALSE, or UNKNOWN) result.

The tested operand can be one of the following:

- A column name
- A literal
- An arithmetic expression
- A Period expression, including a derived period
- The DEFAULT function
- A built-in function such as CURRENT_DATE or USER that evaluates to a system variable

Where Logical Predicates Are Used

Logical predicates are typically used in a WHERE, ON, or HAVING clause to qualify or disqualify rows as a table expression is evaluated in a SELECT statement.

Logical predicates can be used in a WHEN clause search condition in a searched CASE expression.

The type of test performed is a function of the predicate.

Conditional Expressions as a Collection of Logical Primitives

You can think of a conditional expression as a collection of logical predicate primitives where the order of evaluation is controlled by the use of the logical operators AND, OR, and NOT and by the placement of parentheses.

Superficially similar conditional expressions can produce radically different results depending on how you group their component primitives, so use caution in planning the logic of any conditional expressions.

SQL supports the logical predicate primitives listed in the following table. Note that Match and Unique conditions are *not* supported.

Logical Predicate Primitive Condition	SQL Logical Predicate	Function
Comparison	For a complete list of SQL comparison operators, see "Supported Comparison Operators" on page 497.	Tests for equality, inequality, or magnitude difference between two data values.
Range	BETWEEN NOT BETWEEN	Tests whether a data value is included within (or excluded from) a specified range of column data values.
Like	LIKE	Tests for a pattern match between a specified character string and a column data value.
In	IN NOT IN	Tests whether a data value is (or is not) a member of a specified set of column values. IN is equivalent to = ANY. NOT IN is equivalent to <> ALL.
All	ALL	Tests whether a data value compares TRUE to *all* column values in a specified set.
Any	ANY SOME	Tests whether a data value compares TRUE to *any* column value in a specified set.
Exists	EXISTS NOT EXISTS	Tests whether a specified table contains at least one row.
Period predicates	For a complete list of period predicate operators, see Chapter 22: "Period Functions and Operators."	Operates on: • Two Period expressions • Two derived periods • One Period expression and one derived period • One Period expression and one DateTime expression Evaluates to TRUE, FALSE, or UNKNOWN.
	OVERLAPS	Tests whether two time periods, including derived periods, overlap
	IS UNTIL_CHANGED IS NOT UNTIL_CHANGED	Tests whether the ending bound of a Period expression, including a derived period is (or is not) UNTIL_CHANGED.

Restrictions on the Data Types Involved in Predicates

The restrictions in the following table apply to operations involving predicates and CLOB, BLOB, and UDT types.

Data Type	Restrictions
BLOB	Predicates do not support BLOB or CLOB data types.
CLOB	You can explicitly cast BLOBs to BYTE and VARBYTE types and CLOBs to CHARACTER and VARCHAR types, and use the results in a predicate.
UDT	The LIKE and OVERLAPS logical predicates do not support UDTs. For EXISTS and NOT EXISTS: Multiple UDTs involved as predicate operands must be identical types because Teradata Database does not perform implicit type conversion on UDTs involved as predicate operands. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs and then explicitly invoke the CAST function within the operation involving predicates. For more information on CREATE CAST, see *SQL Data Definition Language*. For BETWEEN/ NOT BETWEEN and IN/NOT: • Multiple UDTs involved as predicate operands must be identical types because Teradata Database does not perform implicit type conversion on UDTs involved as predicate operands. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs and then explicitly invoke the CAST function within the operation involving predicates. • UDTs involved as predicate operands must have ordering definitions. Teradata Database generates ordering functionality for distinct UDTs where the source types are not LOBs. To create an ordering definition for structured UDTs or distinct UDTs where the source types are LOBs, or to replace system-generated ordering functionality, use CREATE ORDERING. For more information on CREATE CAST and CREATE ORDERING, see *SQL Data Definition Language*.

Restrictions on the DEFAULT Function in a Predicate

The DEFAULT function returns the default value of a column. It has two forms: one that specifies a column name and one that omits the column name. Predicates support both forms of the DEFAULT function, but the following conditions must be true when the DEFAULT function omits the column name:

- The predicate uses a comparison operator
- The comparison involves a single column reference
- The DEFAULT function is not part of an expression

For example, the following statement uses DEFAULT to compare the values of the Dept_No column with the default value of the Dept_No column. Because the comparison operation involves a single column reference, Teradata Database can derive the column context of the DEFAULT function even though the column name is omitted.

```
SELECT * FROM Employee WHERE Dept_No < DEFAULT;
```

Note that if the DEFAULT function evaluates to null, the predicate is unknown and the WHERE condition is false.

Logical Operators and Search Conditions

A search condition, or conditional expression, consists of one or more conditional terms connected by one or more of the following logical predicates:

- Comparison operators
- BETWEEN/NOT BETWEEN
- LIKE
- IN/NOT IN
- ALL or ANY/SOME
- EXISTS/NOT EXISTS
- OVERLAPS
- IS NULL/IS NOT NULL

Logical Operators

An operator applied to the result of a predicate to determine the result of a search condition.

The logical operators are:

- AND
- NOT
- OR

For example:

```
—— expression_1 —— OR —— expression_2 —— OR —— expression_3 ———
```

FF07D220

Use NOT to negate an expression, for example:

```
—— expression_1 —— AND NOT —— expression_2 ———
```

FF07D221

Where To Use Search Conditions

A search condition can be used in various SQL clauses such as WHERE, ON, QUALIFY, RESET WHEN, or HAVING.

When used in a HAVING clause, a logical expression can be used with an aggregate operator.

For example, the following query uses a search condition in a HAVING clause to select from the Employee table those departments with the number 100, 300, 500, or 600, and with a salary average of at least $35,000 but not more than $55,000:

```
SELECT AVG(Salary)
FROM Employee
WHERE DeptNo IN (100,300,500,600)
GROUP BY DeptNo
HAVING AVG(Salary) BETWEEN 35000 AND 55000 ;
```

Rules for Order of Evaluation

The following rules apply to evaluation order for conditional expressions:

- If an expression contains more than one of the same operator, the evaluation precedence is left to right.
- If an expression contains a combination of logical operators, the order of evaluation is as follows:

  1. NOT
  2. AND
  3. OR

- Parentheses can be used to establish the desired evaluation precedence.
- The logical expressions in a conditional expression are not always evaluated left to right.

  Avoid using a conditional expression if its accuracy depends on the order in which its logical expressions are evaluated.

  For example, compare the following two expressions:

  ```
  F2/(NULLIF(F1,0)) > 500
  F1 <> 0 AND F2/F1 > 500
  ```

  The first expression guarantees exclusion of division by zero.

The second allows the possibility of error, because the order of its evaluation determines the exclusion of zeros.

Evaluation Results

Each logical expression in a conditional expression evaluates to one of three results:

- TRUE
- FALSE
- UNKNOWN

AND Truth Table

The following table illustrates the AND logic used in evaluating search conditions.

	x FALSE	x UNKNOWN	x TRUE
y FALSE	FALSE	FALSE	FALSE
y UNKNOWN	FALSE	UNKNOWN	UNKNOWN
y TRUE	FALSE	UNKNOWN	TRUE

OR Truth Table

The following table illustrates the OR logic used in evaluating search conditions.

	x FALSE	x UNKNOWN	x TRUE
y FALSE	FALSE	UNKNOWN	TRUE
y UNKNOWN	UNKNOWN	UNKNOWN	TRUE
y TRUE	TRUE	TRUE	TRUE

NOT Truth Table

The following table illustrates the NOT logic used in evaluating search conditions.

	Result
x FALSE	TRUE
x UNKNOWN	UNKNOWN
x TRUE	FALSE

Subquery Restrictions

Predicates in search conditions cannot specify SELECT AND CONSUME statements in subqueries.

Examples of Logical Operators in Search Conditions

The following examples illustrate the use of logical operators in search conditions.

Example 1

The following example uses a search condition to select from a user table named Profile the names of applicants who have either more than two years of experience or at least twelve years of schooling with a high school diploma:

```
SELECT Name
FROM Profile
WHERE YrsExp > 2
OR (EdLev >= 12 AND Grad = 'Y') ;
```

Example 2

The following statement requests a list of all the employees who report to manager number 10007 or manager number 10012. The manager information is contained in the Department table, while the employee information is contained in the Employee table. The request is processed by joining the tables on DeptNo, their common column.

DeptNo must be fully qualified in every reference to avoid ambiguity and an extra set of parentheses is needed to group the ORed IN conditions. Without them, the result is a Cartesian product.

```
SELECT EmpNo,Name,JobTitle,Employee.DeptNo,Loc
FROM Employee,Department
WHERE (Employee.DeptNo=Department.DeptNo)
AND ((Employee.DeptNo IN
 (SELECT Department.DeptNo
  FROM Department
  WHERE MgrNo=10007))
  OR (Employee.DeptNo IN
   (SELECT Department.DeptNo
    FROM Department
    WHERE MgrNo=10012))) ;
```

Assuming that the Department table contains the following rows:

DeptNo	Department	Loc	MgrNo
100	Administration	NYC	10005
600	Manufacturing	CHI	10007
500	Engineering	ATL	10012
300	Exec Office	NYC	10018

DeptNo	Department	Loc	MgrNo
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

ANY / ALL / SOME Quantifiers

Purpose

Enables quantification in a comparison operation or IN/NOT IN predicate.

Syntax

1101C090

where:

Syntax element ...	Specifies ...
expression	an expression that specifies a value.
comparison_operator	a comparison operator that compares the expression or list of expressions and the literals in the list (Literals syntax) or the subquery (Subquery syntax) to produce a TRUE, FALSE or UNKNOWN result. For more information on comparison operators, see Chapter 10: "Comparison Operators and Functions."
[NOT] IN	a predicate that tests the existence of the expression or list of expressions in the list of literals (Literals syntax) or the subquery (Subquery syntax) to produce a TRUE, FALSE, or UNKNOWN result. For more information on IN/NOT IN, see "IN/NOT IN" on page 919.
literal	a literal value.
subquery	a subquery that selects the same number of expressions as are specified in the expression or list of expressions. The subquery cannot specify a SELECT AND CONSUME statement.

ANSI Compliance

ANY, SOME, and ALL are ANSI SQL:2011 compliant quantifiers.

ANY / ALL / SOME Quantifiers and Literal Syntax

When a list of literals is used with quantifiers and comparison operations or IN/NOT IN predicates, the results are determined as follows.

IF the predicate is ...	AND specifies ...	THEN the result is true when ...
a comparison operation	ALL	the comparison of *expression* and every literal in the list produces true results.
	ANY	the comparison of *expression* and any literal in the list is true.
	SOME	
IN	ALL	*expression* is equal to every literal in the list.
	ANY	*expression* is equal to any literal in the list.
	SOME	
NOT IN	ALL	*expression* is not equal to any literal in the list.
	ANY	*expression* is not equal to every literal in the list.
	SOME	

For comparison operations, implicit conversion rules are the same as for the comparison operators.

If *expression* evaluates to NULL, the result is considered to be unknown.

ANY / ALL / SOME Quantifiers and Subquery Syntax

When subqueries are used with quantifiers and comparison operations or IN/NOT IN predicates, the results are determined as follows.

IF this quantifier is specified ...	AND the predicate is ...	THEN the result is ...	WHEN ...
ALL	a comparison operation	TRUE	the comparison of *expression* and every value in the set of values returned by *subquery* produces true results.
	IN	TRUE	*expression* is equal to every value in the set of values returned by *subquery*.
	NOT IN	TRUE	*expression* is not equal to any value in the set of values returned by *subquery*.

IF this quantifier is specified ...	AND the predicate is ...	THEN the result is ...	WHEN ...
ALL	a comparison operation	TRUE	*subquery* returns no values.
	IN		
	NOT IN		
ANY SOME	a comparison operation	TRUE	the comparison of *expression* and at least one value in the set of values returned by *subquery* is true.
	IN	TRUE	*expression* is equal to at least one value in the set of values returned by *subquery*.
	NOT IN	TRUE	*expression* is not equal to at least one value in the set of values returned by *subquery*.
	a comparison operation	FALSE	*subquery* returns no values.
	IN		
	NOT IN		

Equivalences Using ANY / ALL / SOME and Comparison Operators

The following table provides equivalences for the ANY/ALL/SOME quantifiers, where *op* is a comparison operator:

This ...	Is equivalent to ...
`x` *op* `ALL (:a, :b, :c)`	`(x` *op* `:a) AND (x` *op* `:b) AND (x` *op* `:c)`
`x` *op* `ANY (:a, :b, :c)`	`(x` *op* `:a) OR (x` *op* `:b) OR (x` *op* `:c)`
`x` *op* `SOME (:a, :b, :c)`	

Here are some examples:

This expression ...	Is equivalent to ...
`x < ALL (:a, :b, :c)`	`(x < :a) AND (x < :b) AND (x < :c)`
`x > ANY (:a, :b, :c)`	`(x > :a) OR (x > :b) OR (x > :c)`
`x > SOME (:a, :b, :c)`	

Equivalences Using ANY/ALL/SOME and IN/NOT IN

The following table provides equivalences for the ANY/ALL/SOME quantifiers, where *op* is IN or NOT IN:

This ...	Is equivalent to ...[a]
`NOT (x op ALL (:a, :b, :c))`	`x NOT op ANY (:a, :b, :c)`
	`x NOT op SOME (:a, :b, :c)`
`NOT (x op ANY (:a, :b, :c))`	`x NOT op ALL (:a, :b, :c)`
`NOT (x op SOME (:a, :b, :c))`	

a. If *op* is NOT IN, then NOT *op* is IN, not NOT NOT IN.

Here are some examples:

This expression ...	Is equivalent to ...
`NOT (x IN ANY (:a, :b, :c))`	`x NOT IN ALL (:a, :b, :c)`
`NOT (x IN ALL (:a, :b, :c))`	`x NOT IN ANY (:a, :b, :c)`
`NOT (x NOT IN ANY (:a, :b, :c))`	`x IN ALL (:a, :b, :c)`
`NOT (x NOT IN ALL (:a, :b, :c))`	`x IN ANY (:a, :b, :c)`

Example 1

The following statement uses a comparison operator with the ANY quantifier to select the employee number, name, and department number of anyone in departments 100, 300, and 500:

This Expression ...	Is Equivalent to this expression...

`SELECT EmpNo, Name, DeptNo`
`FROM Employee`
`WHERE DeptNo = ANY (100,300,500) ;`
	`SELECT EmpNo, Name, DeptNo`
	`FROM Employee`
	`WHERE (DeptNo = 100)`
	`OR (DeptNo = 300)`
	`OR (DeptNo = 500) ;`
	and
	`SELECT EmpNo, Name, DeptNo`
	`FROM Employee`
	`WHERE DeptNo IN (100,300,500) ;`

Example 2

Here is an example that uses a subquery in a comparison operation that specifies the ALL quantifier:

```
SELECT EmpNo, Name, JobTitle, Salary, YrsExp
FROM Employee
WHERE (Salary, YrsExp) >= ALL
 (SELECT Salary, YrsExp FROM Employee) ;
```

Example 3

This example shows the behavior of ANY/ALL/SOME.

Consider the following table definition and contents:

```
CREATE TABLE t (x INTEGER);
INSERT t (1);
INSERT t (2);
INSERT t (3);
INSERT t (4);
INSERT t (5);
```

IF you use this query ...	THEN the result is ...
`SELECT * FROM t WHERE x IN ANY (1,2)`	`1, 2`
`SELECT * FROM t WHERE x = SOME (1,2)`	`1, 2`
`SELECT * FROM t WHERE x NOT IN ALL (1,2)`	`3, 4, 5`
`SELECT * FROM t WHERE NOT (x IN ANY (1,2))`	`3, 4, 5`
`SELECT * FROM t WHERE NOT (x = SOME (1,2))`	`3, 4, 5`
`SELECT * FROM t WHERE x NOT IN SOME (1, 2)`	`1, 2, 3, 4, 5`
`SELECT * FROM t WHERE x NOT = ANY (1, 2)`	`1, 2, 3, 4, 5`
`SELECT * FROM t WHERE x IN ALL (1,2)`	no rows
`SELECT * FROM t WHERE NOT (x NOT IN SOME (1,2))`	no rows
`SELECT * FROM t WHERE x = ALL (1,2)`	no rows
`SELECT * FROM t WHERE NOT (x NOT = ANY (1,2))`	no rows

BETWEEN / NOT BETWEEN

Purpose

Tests whether an expression value is between two other expression values.

Syntax

```
── expr1 ┬──────┬── BETWEEN ── expr2 ── AND ── expr3 ──
      └ NOT ┘
```

HH01A038

ANSI Compliance

BETWEEN and NOT BETWEEN are ANSI SQL:2011 compliant.

Usage Notes

The BETWEEN test is satisfied if the following condition is true.

```
expression_2 <= expression_1 <= expression_3
```

If the BETWEEN test fails, no rows are returned.

The BETWEEN test is treated as two separate logical comparisons.

```
expression_1 >= expression_2 AND expression_1 <= expression_3.
```

This expression ...	Is equivalent to ...
x BETWEEN y AND z	((x >= y) AND (x <=z))

Note that because *expression_1* is actually evaluated twice, using a nondeterministic function, such as RANDOM, can produce unexpected results.

Example

The following example uses a search condition in a HAVING clause to select from the Employee table those departments with the number 100, 300, 500, or 600, and with a salary average of at least $35,000 but not more than $55,000:

```
SELECT AVG(Salary)
FROM Employee
WHERE DeptNo IN (100,300,500,600)
GROUP BY DeptNo
HAVING AVG(Salary) BETWEEN 35000 AND 55000 ;
```

EXISTS / NOT EXISTS

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

```
                 EXISTS — subquery
  ┬──────────┬
  └─ NOT ────┘
```

HH01A047

ANSI Compliance

EXISTS and NOT EXISTS are ANSI SQL:2011 compliant.

Usage Notes

The function of the EXISTS predicate is to test the result of *subquery*.

If execution of the subquery returns response rows then the where condition is considered satisfied.

Note that use of the NOT qualifier for the EXISTS predicate reverses the sense of the test. Execution of the subquery does not, in fact, return any response rows. Instead, it returns a boolean result to indicate whether responses would or would not have been returned had they been requested.

Subquery Restrictions

The subquery cannot specify a SELECT AND CONSUME statement.

Relationship Between EXISTS/NOT EXISTS and IN/NOT IN

EXISTS predicate tests the existence of specified rows of a subquery. In general, EXISTS can be used to replace comparisons with IN and NOT EXISTS can be used to replace comparisons with NOT IN. However, the reverse is not true. Some problems can be solved only by using EXISTS and/or NOT EXISTS predicate. For an example, see "For ALL" on page 915.

For information on IN and NOT IN, see "IN/NOT IN" on page 919.

Example

To select rows of t1 whose values in column x1 are equal to the value in column x2 of t2, one of the following queries can be used:

```
SELECT *
FROM t1
WHERE x1 IN
 (SELECT x2
  FROM t2);
SELECT *
FROM t1
WHERE EXISTS
 (SELECT *
  FROM t2
  WHERE t1.x1=t2.x2);
```

To select rows of t1 whose values in column x1 are not equal to any value in column x2 of t2, you can use any one of the following queries:

```
SELECT *
FROM t1
WHERE x1 NOT IN
 (SELECT x2
  FROM t2);

SELECT *
FROM t1
WHERE NOT EXISTS
 (SELECT *
  FROM t2
  WHERE t1.x1=t2.x2);

SELECT 'T1 is not empty'
WHERE EXISTS
 (SELECT *
  FROM t1);

SELECT 'T1 is empty'
WHERE NOT EXISTS
 (SELECT *
  FROM t1);
```

EXISTS Predicate Versus NOT IN and Nulls

Use the NOT EXISTS predicate instead of NOT IN if the following conditions are true:

- Some column of the NOT IN condition is defined as nullable.
- Any rows from the main query with a null in any column of the NOT IN condition should always be returned.
- Any nulls returned in the select list of the subquery should not prevent any rows from the main query from being returned.

For example, if all of the previous conditions are true for the following query, use NOT EXISTS instead of NOT IN:

```
SELECT dept, DeptName
FROM Department
WHERE Dept NOT IN
 (SELECT Dept
  FROM Course);
```

The NOT EXISTS version looks like this:

```
SELECT dept, DeptName
FROM Department
WHERE NOT EXISTS
 (SELECT Dept
  FROM Course
  WHERE Course.Dept=Department.Dept);
```

That is, either Course.Dept or Department.Dept is nullable and a row from Department with a null for Dept should be returned and a null in Course.Dept should not prevent rows from Department from being returned.

For ALL

Two nested NOT EXISTS can be used to express a SELECT statement that embodies the notion of "for all (logical ∀) the values in a column, there exists (logical ∃) …"

For example the query to select a 'true' value if the library has at least one book for all the publishers can be expressed as follows:

```
SELECT 'TRUE'
WHERE NOT EXISTS
 (SELECT *
  FROM publisher pb
  WHERE NOT EXISTS
   (SELECT *
    FROM book bk
    WHERE pb.PubNum=bk.PubNum);
```

NOT EXISTS Clauses and Stored Procedures

You cannot specify a NOT EXISTS clause in a stored procedure conditional expression if that expression also references an alias for a local variable, parameter, or cursor.

NOT EXISTS and Recursive Queries

NOT EXISTS cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify the NOT EXISTS predicate.

Example 1: EXISTS with Correlated Subqueries

Select all student names who have registered in at least one class offered by some department.

```
SELECT SName, SNo
FROM student s
WHERE EXISTS
 (SELECT *
```

```
FROM department d
WHERE EXISTS
 (SELECT *
  FROM course c, registration r, class cl
  WHERE c.Dept=d.Dept
  AND c.CNo=r.CNo
  AND s.SNo=r.SNo
  AND r.CNo=cl.CNo
  AND r.Sec=cl.Sec));
```

The content of the student table is as follows:

Sname	SNo
Helen Chu	1
Alice Clark	2
Kathy Kim	3
Tom Brown	4

The content of the department table is as follows:

Dept	DeptName
100	Computer Science
200	Physic
300	Math
400	Science

The content of course table is as follows:

CNo	Dept
10	100
11	100
12	200
13	200
14	300

The content of the class table is as follows:

CNo	Sec
10	1
11	1
12	1
13	1
14	1

The content of the registration table is as follows:

CNo	SNo	Sec
10	1	1
10	2	1
11	3	1
12	1	1
13	2	1
14	1	1

The following rows are returned:

```
SName              SNo
-----------        ---
Helen Chu1           *
Alice Clark          2
Kathy Kim            3
```

For a full explanation of correlated subqueries, see "Correlated Subqueries" in *SQL Data Manipulation Language*.

Example 2: NOT EXISTS with Correlated Subqueries

Select the names of all students who have registered in at least one class offered by each department that offers a course.

```
SELECT SName, SNo
FROM student s
WHERE NOT EXISTS
 (SELECT *
  FROM department d
  WHERE d.Dept IN
   (SELECT Dept
    FROM course) AND NOT EXISTS
     (SELECT *
```

```
FROM course c, registration r, class cl
WHERE c.Dept=d.Dept
AND c.CNo=r.CNo
AND s.SNo=r.SNo
AND r.CNo=cl.CNo
AND r.Sec=cl.Sec)));
```

With the contents of the tables as in "Example 1: EXISTS with Correlated Subqueries" on page 915, 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 literals.
- 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:2011 compliant.

Using TO in a list of literals is a Teradata extension to the ANSI standard.

Syntax 1: *expression* IN and NOT IN *expression* or literals

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 literals named by *literal*, *signed_literal* TO *signed_literal*, or *datetime_literal*.

Syntax element ...	Specifies ...
IN NOT IN	whether the test is inclusive or exclusive. You can substitute: • IN ANY • IN SOME • = ANY • = SOME for IN, unless a list of literals is specified and includes *signed_literal_1* TO *signed_literal_2* You can substitute: • <> ALL • NOT IN ALL for NOT IN, unless a list of literals is specified and includes *signed_literal_1* TO *signed_literal_2*
expression_2	the value in which the existence of *expression_1* is to be tested.
literal	• literal • macro parameter • built-in value such as TIME or DATE
signed_literal_1 TO *signed_literal_2*	a range of literals.
datetime_literal	an ANSI DateTime literal.

Result

If IN is used with a list of literals, the result is true if the value of *expression_1* is:

- equal to any *literal* in the list,
- between *signed_literal_1* and *signed_literal_2*, inclusively, when *signed_literal_1* is less than or equal to *signed_literal_2*, or
- between *signed_literal_2* and *signed_literal_1*, inclusively, when *signed_literal_2* is less than *signed_literal_1*

If the value of *expression_1* is null, then the result is considered to be unknown.

If the value of *expression_1* is not null, and none of the conditions are satisfied for the result to be true, then the result is false.

Using this form, the IN search condition is satisfied if the expression is equal to any of the values in the list of literals; the NOT IN condition is satisfied if none of the values in the list of literals are equal to the expression.

THE condition is true for this form ...	WHEN ...
expression_1 IN *expression_2*	*expression_1* = *expression_2*

THE condition is true for this form ...	WHEN ...
expression_1 NOT IN *expression_2*	*expression_1* <> *expression_2*
expression_1 IN (*const_1*, *const_2*)	(*expression_1* = *const_1*) OR (*expression_1* = *const_2*)
expression_1 NOT IN (*const_1*, *const_2*)	(*expression_1* <> *const_1*) AND (*expression_1* <> *const_2*)
expression_1 IN (*signed_const_1* TO *signed_const_2*) where *signed_const_1* <= *signed_const_2*	(*signed_const_1* <= *expression_1*) AND (*expression_1* <= *signed_const_2*)
expression_1 IN (*signed_const_1* TO *signed_const_2*) where *signed_const_2* < *signed_const_1*	(*signed_const_2* <= *expression_1*) AND (*expression_1* <= *signed_const_1*)
expression_1 NOT IN (*signed_const_1* TO *signed_const_2*) where *signed_const_1* <= *signed_const_2*	(*expression_1* < *signed_const_1*) OR (*expression_1* > *signed_const_2*)
expression_1 NOT IN (*signed_const_1* TO *signed_const_2*) where *signed_const_2* < *signed_const_1*	(*expression_1* < *signed_const_2*) OR (*expression_1* > *signed_const_1*)

Here are some examples:

This statement ...	Is equivalent to this statement ...

`SELECT DeptNo`
`FROM Department`
`WHERE DeptNo IN (500, 600);`
	`SELECT DeptNo`
	`FROM Department`
	`WHERE (DeptNo = 500)`
	`OR (DeptNo = 600);`

`UPDATE Employee`
`SET Salary=Salary + 200`
`WHERE DeptNo NOT IN (100, 700);`
	`UPDATE Employee`
	`SET Salary=Salary + 200`
	`WHERE (DeptNo ^= 100)`
	`AND (DeptNo ^= 700);`

Usage Notes

If IN is used with a single-term operator, that operator can be a literal or an expression. If a multiple-term operator is used, that operator must consist of literals; expressions are not allowed.

The *expression_1* data type and the *literal* values must be compatible. Implicit conversion rules are the same as for the comparison operators.

Relationship Between IN/NOT IN and EXISTS/NOT EXISTS

In general, you can use EXISTS to replace comparisons with IN, and NOT EXISTS to replace comparisons with NOT IN. However, the reverse is not true. The solutions to some problems

require using the EXISTS or NOT EXISTS predicate. For information on EXISTS and NOT EXISTS, see "EXISTS / NOT EXISTS" on page 913.

Equivalences Using IN / NOT IN, NOT, and ANY / ALL / SOME

The following table provides equivalences for the ANY/ALL/SOME quantifiers, where *op* is IN or NOT IN:

This usage ...	Is equivalent to ...[a]
`NOT (x` *op* `ALL (:a, :b, :c))`	`x NOT` *op* `ANY (:a, :b, :c)`
	`x NOT` *op* `SOME (:a, :b, :c)`
`NOT (x` *op* `ANY (:a, :b, :c))`	`x NOT` *op* `ALL (:a, :b, :c)`
`NOT (x` *op* `SOME (:a, :b, :c))`	
`NOT (x` *op* `(:a, :b, :c))`	`x NOT` *op* `(:a, :b, :c)`

a. In the equivalences, if *op* is NOT IN, then NOT *op* is IN, not NOT NOT IN.

Here are some examples:

This expression ...	Is equivalent to ...
`NOT (x IN ANY (:a, :b, :c))`	`x NOT IN ALL (:a, :b, :c)`
`NOT (x IN ALL (:a, :b, :c))`	`x NOT IN ANY (:a, :b, :c)`
`NOT (x NOT IN ANY (:a, :b, :c))`	`x IN ALL (:a, :b, :c)`
`NOT (x NOT IN ALL (:a, :b, :c))`	`x IN ANY (:a, :b, :c)`
`NOT (x IN (:a, :b, :c))`	`x NOT IN (:a, :b, :c)`
`NOT (x NOT IN (:a, :b, :c))`	`x IN (:a, :b, :c)`

Syntax 2: *expression* IN and NOT IN *subquery*

This syntax for IN and NOT IN is correct in either of the following two forms:

where:

Syntax element ...	Specifies ...
expression	the value of the expression whose existence is to be tested in *subquery*.
subquery	a SELECT statement that returns values that satisfy the stated search criterion. The *subquery* must: • Be enclosed in parentheses. • Not end with a semicolon. • Select the same number of expressions as are defined in the expression list. • Not specify a SELECT AND CONSUME statement.

Behavior of Nulls for IN

A statement result does not include column nulls when IN is used with a subquery.

Behavior of Nulls for NOT IN

The following table explains the behavior of nulls for NOT IN for queries of various forms:

FOR a query of the following form ...	IF ...	THEN ...

`SELECT ... FROM T1`
`WHERE x NOT IN`
`(SELECT y FROM T2);`
	one of the y values is null	no T1 rows are returned for the entire query.

	some rows are returned by the subquery, and if x contains some nulls	those T1 rows that contain a null in x are not returned.

`SELECT ... FROM T1`
`WHERE` *`expression_list_1`* `NOT IN`
`(SELECT` *`expression_list_2`*
`FROM T2);`
	a null is the first field in *expression_list_2*	no rows from T1 are returned.

	a null is in a field other than the first field of *expression_list_2*	some rows may be returned
	the subquery returns some rows, and if a null is in the first field in *expression_list_1*	the T1 rows containing a null in the first field of *expression_list_1* are not returned.

`SELECT ... FROM T1`
`WHERE` *`expression_list_1`* `NOT IN`
`(SELECT` *`expression_list_2`*
`FROM T2`
`WHERE` *`search_condition`*`);`
	the *search_condition* on T2 returns no rows	all T1 rows, including those containing a NULL in the first field of *expression_list_1*, are returned.

NOT IN Clauses and Stored Procedures

You cannot specify a NOT IN clause in a stored procedure conditional expression if that expression also references an alias for a local variable, parameter, or cursor.

NOT IN and Recursive Queries

NOT IN cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify the NOT IN predicate.

Queries With Large NOT IN Clauses Can Fail

Queries that contain thousands of arguments within an IN or NOT IN clause sometimes fail.

For example, suppose you ran the following query with 16000 IN clause arguments, and it failed.

```
SELECT MAX(emp_num)
FROM employee
WHERE emp_num IN(1,2,7,8,...,121347);
```

A workaround when this problem occurs is to rewrite the query using a temporary or volatile table to contain the arguments within the IN clause.

The following statements allow you to make the same selection, but without failure.

```
CREATE VOLATILE TABLE temp_IN_values (
 in_value INTEGER) ON COMMIT PRESERVE ROWS;

INSERT INTO temp_IN_values
SELECT emp_num
FROM table_with_emp_num_values;
```

The new query is as follows:

```
SELECT MAX(emp_num)
FROM employee AS e JOIN temp_IN_values AS en
ON (e.emp_num = en.in_value);
```

Example 1

The following statement searches for the names of all employees who work in Atlanta.

```
SELECT Name
FROM Employee
WHERE DeptNo IN
 (SELECT DeptNo
  FROM Department
  WHERE Loc = 'ATL');
```

Example 2

Using a similar example but assuming that the DeptNo is divided into two columns, the following statement could be used:

```
SELECT Name
FROM Employee
WHERE (DeptNoA, DeptNoB) IN
```

```
(SELECT DeptNoA, DeptNoB
 FROM Department
 WHERE Loc = 'LAX') ;
```

Example 3

This example shows the behavior of IN/NOT IN with a list of literals.

Consider the following table definition and contents:

```
CREATE TABLE t (x INTEGER);
INSERT t (1);
INSERT t (2);
INSERT t (3);
INSERT t (4);
INSERT t (5);
```

IF you use this query ...	THEN the result is ...
`SELECT * FROM t WHERE x IN (1,2)`	`1, 2`
`SELECT * FROM t WHERE x IN ANY (1,2)`	`1, 2`
`SELECT * FROM t WHERE NOT (x NOT IN (1,2))`	`1, 2`
`SELECT * FROM t WHERE x NOT IN (1,2)`	`3, 4, 5`
`SELECT * FROM t WHERE x NOT IN ALL (1,2)`	`3, 4, 5`
`SELECT * FROM t WHERE NOT (x IN (1, 2))`	`3, 4, 5`
`SELECT * FROM t WHERE NOT (x IN ANY (1,2))`	`3, 4, 5`
`SELECT * FROM t WHERE x IN (3 TO 5)`	`3, 4, 5`
`SELECT * FROM t WHERE x NOT IN SOME (1, 2)`	`1, 2, 3, 4, 5`
`SELECT * FROM t WHERE x IN (1, 2 TO 4, 5)`	`1, 2, 3, 4, 5`
`SELECT * FROM t WHERE x IN ALL (1,2)`	no rows
`SELECT * FROM t WHERE NOT (x NOT IN SOME (1,2))`	no rows
`SELECT * FROM t WHERE x NOT IN (1 TO 5)`	no rows

IS NULL / IS NOT NULL

Purpose

Searches for or excludes nulls in an expression.

Syntax

where:

Syntax element ...	Specifies ...
expression	an expression that specifies a value that is tested for nulls.

ANSI Compliance

IS NULL and IS NOT NULL are ANSI SQL:2011 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 the NULLs must appear separately.

For example, to select the names of all employees without the job title of “Manager” or “Vice Pres”, plus the names of all employees with a null in the JobTitle column, you must enter the statement as follows:

```
SELECT Name, JobTitle
FROM Employee
WHERE (JobTitle NOT IN ('Manager' OR 'Vice Pres'))
OR (JobTitle IS NULL) ;
```

Example 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) ;
```

LIKE

Purpose

Searches for a character string pattern within another character string or character string expression.

Syntax

FF07D196

where:

Syntax Element ...	Specifies ...
expression	a character string or character string expression argument to be searched for the substring *pattern_expression*.
pattern_expression	a character expression for which *expression* is to be searched.
ANY ALL SOME	a quantifier that allows one or more expressions to be searched for one or more patterns or for one or more values returned by a subquery. SOME is a synonym for ANY.
subquery	a SELECT statement argument. A subquery cannot specify a SELECT AND CONSUME statement.
ESCAPE *escape_character*	keyword/variable combination specifying a single escape character (single or multibyte).

ANSI Compliance

This is ANSI SQL:2011 compliant.

Optimized Performance Using a NUSI

If it is cost-effective, the Optimizer may choose to evaluate a LIKE expression by scanning a NUSI with or without accessing the base table. The cost of using a NUSI depends on the selectivity of the LIKE expression, the size of the NUSI subtable, and if the NUSI is a covering index or a partially covering index. For a partially covering index, the cost of sorting the RowID spool is also included. For details on NUSIs and query covering, see *Database Design*.

The Optimizer can perform a better cost comparison between using a NUSI and using an all-rows scan if the following are true:

- There are statistics collected for both the base table primary index and for the NUSI columns against which the *expression* string is evaluated.
- The *expression* string is either the mode or max value in at least one interval in the base table statistics histogram.

You cannot use a NUSI with a VARCHAR field for processing a LIKE expression when:

- the NUSI contains a VARCHAR field, and the VARCHAR field is used in a NOT LIKE operation.
- the NUSI contains a VARCHAR field, and the VARCHAR field is used in a string function. For example, the following is not allowed if d1 is a NUSI column of VARCHAR type.

  ```
  d1||'ab' LIKE 'b ab'
  ```

In addition, a NUSI with a VARCHAR field cannot be used as a partially covering index for an unconstrained aggregate query.

Null Expressions

If any expression in a comparison is null, the result of the comparison is unknown.

For a LIKE operation to provide a true result when searching fields that may contain nulls, the statement must include the IS [NOT] NULL operator.

Case Specification

If neither *pattern_expression* nor *expression* has been designated CASESPECIFIC, any lowercase letters in *pattern_expression* and *expression* are converted to uppercase before the comparison operation occurs. If ESCAPE is specified and the escape character is a lowercase character, it is also converted to uppercase before the comparison operation occurs.

If either *expression* or *pattern_expression* has been designated CASESPECIFIC, two letters match only if they are the same letters and the same case.

Wildcard Characters

The % and _ characters may be used in any combination in *pattern_expression*.

Character	Description
% (PERCENT SIGN)	Represents any string of zero or more arbitrary characters. Any string of characters is acceptable as a replacement for the percent.
_ (LOW LINE)	Represents exactly one arbitrary character. Any single character is acceptable in the position in which the underscore character appears.

The underscore and percent characters cannot be used in a pattern. To get around this, specify a single escape character in addition to *pattern_expression*. For details, see "ESCAPE Feature of LIKE" on page 931.

The following table describes how the metacharacters % and _ (and their fullwidth equivalents) behave when matching strings for various server character sets. Note that ANSI only defines the single byte spacing underscore and percent sign metacharacters.

Teradata SQL extends the permissible metacharacter set for the LIKE predicate to include the fullwidth underscore and the fullwidth percent sign.

FOR this server character set ...	USE this metacharacter ...	TO match this character or characters ...	
		ANSI Mode	Teradata Mode
KANJI1	spacing underscore	any *one* single- or multibyte character.	any *one* single byte character.
	fullwidth spacing underscore	any *one* single byte character or multibyte character.	any *one* single byte character or multibyte character.
	percent sign	any sequence of single or multibyte characters.	any sequence of single byte characters or multibyte characters.
	fullwidth percent sign	any sequence of single or multibyte characters.	any sequence of single byte characters or multibyte characters.
UNICODE LATIN KANJISJIS	fullwidth spacing underscore	none. These characters are not treated as metacharacters in order to maintain compliance with the ANSI SQL standard.	
	fullwidth percent		
GRAPHIC	fullwidth spacing underscore	any *one* single GRAPHIC character.	
	fullwidth percent sign	any sequence of GRAPHIC characters.	

ESCAPE Feature of LIKE

When the defined ESCAPE character is in the pattern string, it must be immediately followed by an underscore, percent sign, or another ESCAPE character.

In a left-to-right scan of the pattern string the following rules apply when ESCAPE is specified:

- Until an instance of the ESCAPE character occurs, characters in the pattern are interpreted at face value.
- When an ESCAPE character immediately follows another ESCAPE character, the two character sequence is treated as though it were a single instance of the ESCAPE character, considered as a normal character.
- When an underscore metacharacter immediately follows an ESCAPE character, the sequence is treated as a single underscore character (not a wildcard character).
- When a percent metacharacter immediately follows an ESCAPE character, the sequence is treated as a single percent character (not a wildcard character).
- When an ESCAPE character is not immediately followed by an underscore metacharacter, a percent metacharacter, or another instance of itself, the scan stops and an error is reported.

Example

The following example illustrates the use of ESCAPE:

To look for the pattern ‘95%’ in a string such as ‘Result is 95% effective’, if Result is the field to be checked, use:

```
WHERE Result LIKE '%95Z%%' ESCAPE 'Z'
```

This clause finds the value ‘95%’.

Pad Characters

The following notes apply to pad characters and how they are treated in strings:

- Pad characters are significant in both the character expression, and in the pattern string.
- When using pattern matching, be aware that both leading and trailing pad characters in the field or expression must match exactly with the pattern.

  For example, ‘A%BC’ matches ‘AxxBC’, but not ‘AxxBCΔ’, and ‘A%BCΔ’ matches ‘AxxBCΔ’, but not ‘AxxBC’ or ‘AxxBCΔΔ’ (Δ indicates a pad character).
- To retrieve the row in all cases, consider using the TRIM function, which removes both leading and trailing pad characters from the source string before doing the pattern match.

  For example, to remove trailing pad characters:

  ```
  TRIM (TRAILING FROM expression) LIKE pattern-string
  ```

  To remove leading and trailing pad characters:

  ```
  TRIM (BOTH FROM expression) LIKE pattern-string
  ```

- If *pattern_expression* is forced to a fixed length, trailing pad characters might be appended. In such cases, the field must contain the same number of trailing pad characters in order to match.

  For example, the following statement appends trailing pad characters to pattern strings shorter than 5 characters long.

  ```
  CREATE MACRO (pattern (CHAR(5)) AS
  field LIKE :pattern…
  ```

- To retrieve the row in all cases, apply the TRIM function to the pattern string (TRIM (TRAILING FROM :pattern)), or the macro parameter can be defined as VARCHAR.

  These two methods do not always return the same results.TRIM removes pad characters, while the VARCHAR method maintains the data pattern exactly as entered.

Example 1

The following example uses the LIKE predicate to select a list of employees whose job title contains the string “Pres”:

```
SELECT Name, DeptNo, JobTitle
FROM Employee
WHERE JobTitle LIKE '%Pres%' ;
```

The form %string% requires Teradata Database to examine much of each string x. If x is long and there are many rows in the table, the search for qualifying rows may take a long time.

The result returned is:

Name	DeptNo	JobTitle
Watson L	500	Vice President
Phan A	300	Vice President
Russel S	300	President

Example 2

This example selects a list of all employees whose last name begins with the letter P.

```
SELECT Name
FROM Employee
WHERE Name LIKE 'P%';
```

The result returned is:

```
Name
----------
Phan A
Peterson J
```

Example 3

This example uses the % and _ characters to select a list of employees with the letter A as the second letter in the last name. The length of the return string may be two or more characters.

```
SELECT Name
FROM Employee
WHERE Name LIKE '_a%';
```

returns the result:

```
Name
----------
Marston A
Watson L
Carter J
```

Replacing _a% with _a_ changes the search to a three-character string with the letter a as the second character. Because none of the names in the Employee table fit this description, the query returns no rows.

Both leading and trailing pad characters in a pattern are significant to the matching rules.

Example 4

LIKE 'ΔΔZ%' locates only those fields that start with two pad characters followed by Z.

ANY / ALL / SOME Quantifiers

SQL recognizes the quantifiers ANY (or SOME) and ALL. A quantifier allows one or more expressions to be compared with one or more values such as shown by the following generic example.

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.

```
── expr ┬──────┬── LIKE ── quantifier ──( subquery )──────────
     └ NOT ┘
```

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.

```
──(┬──────┬─┬──────┬)──┬──────┬── LIKE ── quantifier ──( subquery )──────
   └ , ┘   └ expr ┘    └ NOT ┘
```

HH01A046

For example:

```
(x,y) LIKE ALL (SELECT a,b FROM c)
```

is equivalent to:

```
(x LIKE c.a) AND (y LIKE c.b)
```

Behavior of the ESCAPE Character

When *escape_character* is used in (generic) *string_2*, it must be followed immediately by a metacharacter of the appropriate server character set or another *escape_character*.

The resultant two-character sequence matches a single character in *string_1* if and only if the character in *string_1* collates identically to the character following the *escape_character* in *string_2*.

In other words, *escape_character* is ignored for matching purposes and the character following *escape_character* is matched for a single occurrence of itself.

When *string_1* and *string_2* do not share a common server character set, then the valid metacharacters are SPACING UNDERSCORE and PERCENT SIGN because the arguments are translated to UNICODE automatically when mismatched. Their behavior then follows the rules described in "Implicit Character-to-Character Translation" on page 599.

Miscellaneous Examples

Function	Result
_KanjiSJIS '92年abc' LIKE _Unicode '%abc'	TRUE
_KanjiSJIS '92年abc' LIKE _Unicode '**%**abc'	FALSE[a]
'c%' LIKE 'c%%' ESCAPE '%'	TRUE
'c**%**' LIKE 'c%%' ESCAPE '%'	FALSE[b]

a. % (FULLWIDTH PERCENT SIGN) is not a metacharacter in either KanjiSJIS or Unicode.

b. % (FULLWIDTH PERCENT SIGN) does not match % (PERCENT SIGN).

KanjiEBCDIC Examples

The following examples indicate the behavior of LIKE with KanjiEBCDIC strings using the function (*expression* LIKE *pattern_expression*).

expression	*pattern_expression*	Server Character Set	Result
MN<**AB**>	%<**B**>	KANJI1	TRUE
MN<**AB**>P	<%**B**>%	KANJI1	TRUE
MN<**AB**>P	%P	KANJI1	TRUE

expression	*pattern_expression*	Server Character Set	Result
MN<**AB**>P	%<__**C**>%	KANJI1	FALSE

__ represents a FULLWIDTH UNDERSCORE.

KanjiEUC Examples

The following examples indicate the behavior of LIKE with KanjiEUC strings using the function (*expression* LIKE *pattern_expression*).

expression	*pattern_expression*	Server Character Set	Result
ss_3**A** ss_2**B** ss_3**C** ss_2**D**	% ss_2**B**%	UNICODE	TRUE
M ss_2**B N** ss_2**D**	**M** __%	GRAPHIC	TRUE
ss_3**A** ss_2**B** ss_3**C** ss_2**D**	__%	KANJISJIS	TRUE
ss_3**A** ss_2**B** ss_3**C** ss_2**D**	_ %	KANJISJIS	TRUE

__ represents a FULLWIDTH UNDERSCORE.

_ represents a SPACING UNDERSCORE.

KanjiShift-JIS Examples

The following examples indicate the behavior of LIKE with KanjiShift-JIS strings using the function (*expression* LIKE *pattern_expression*).

expression	*pattern_expression*	Server Character Set	ANSI Mode Result	Teradata Mode Result
ABCD	__B%	GRAPHIC	TRUE	TRUE
mn**ABCI**	%B%	UNICODE	TRUE	TRUE
mn**ABCI**	%**I**	UNICODE	TRUE	TRUE
mn**ABCI**	mn_%**I**	KANJI1	TRUE The underscore in *pattern_expression* matches a single byte- or multibyte character in ANSI mode.	FALSE The underscore in *pattern_expression* matches a single byte character in Teradata mode.
mn**ABCI**	mn__%**I**	KANJI1	TRUE	TRUE

__ represents a FULLWIDTH UNDERSCORE.

_ represents a SPACING UNDERSCORE.

OVERLAPS

For OVERLAPS, see "OVERLAPS" on page 1067.

CHAPTER 20 Null-Handling Functions

This chapter describes functions that handle null input values.

NVL

Purpose

Replaces a NULL with a numeric or a string value as the result value.

Syntax

```
┬──────────────────┬─ NVL ─── ( expr1, expr2 ) ──────────────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A920

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
expr1	a numeric or character argument. • If *expr1* is NULL, *expr2* is returned. • If *expr1* is not NULL, *expr1* is returned.
expr2	a numeric or character argument.

Invocation

NVL is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, NUMBER, CHAR, VARCHAR

All of the input arguments must be the same data type or else the types must be compatible.

Result Type

NVL is a scalar function whose return value data type depends on the data type associated with the arguments passed to it.

- If the input arguments are numeric types, the function determines which argument has the highest precedence, converts the other argument to that data type, and returns that

data type. For details about the order of precedence, see “Compatible Types” in *SQL External Routine Programming*.

If that data type is DECIMAL/NUMERIC and the precision and scale of the input arguments are different, the precision and scale of the return type is set to achieve the maximum precision possible. For example, if the input arguments are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return type would need three digits to the left of the decimal point and seven digits to the right of the decimal point to avoid any reduction in precision.

In cases where it is not possible to maintain the maximum precision, the data is rounded according to the DBS Control Record RoundHalfWayMagUp field. For example, if the input arguments are DECIMAL(32, 8) and DECIMAL(30, 28), the return type will be DECIMAL(38,14). This will allow for 24 digits to the left of the decimal point (required for the DECIMAL(32,8) argument), and 14 digits to the right of the decimal point.

If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).

- If the two arguments are character data types, the function converts the second argument to the data type of the first argument and returns the type as VARCHAR.
- If all input character types are LATIN, the result is LATIN. If any input is not LATIN, the function converts all input to Unicode and the return character set is Unicode.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Example

The following query:

```
SELECT department_name,
   NVL(last_name,'NO EMPLOYEE') "LAST NAME"
FROM employee E
   FULL OUTER JOIN Department D
ON department_number=d.department_number;
WHERE department_number IN(402,600);
```

returns:

```
Department_name      LAST NAME
--------- --------------------
Software Support     Crane
New Department       NO EMPLOYEE
```

In this example, the NVL function returned the result as last_name column value from the employee table and for the cases where the last_name value is NULL, the function returned the result as NO EMPLOYEE.

NVL2

Purpose

Returns one of two values based on whether or not *expr1* is NULL.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── NVL2 ── ( expr1, expr2, expr3 ) ──────────────────────────
```

1101A921

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
expr1	a numeric or character expression. • If *expr1* is not NULL, *expr2* is returned. • If *expr1* is NULL, *expr3* is returned.
expr2	a numeric or character expression.
expr3	a numeric or character expression.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

NVL2 is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

BYTEINT, SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, FLOAT/REAL/DOUBLE PRECISION, NUMBER, CHAR, VARCHAR

expr2 and *expr3* must be the same data type or else the types must be compatible.

Result Type

NVL2 is a scalar function whose return data type depends on the data types associated with the arguments passed to the function.

- If *expr2* and *expr3* are numeric types, the function determines which argument has the highest precedence, converts the other argument to that data type, and returns that data type. For details about the order of precedence, see "Compatible Types" in *SQL External Routine Programming*.

  If that data type is DECIMAL/NUMERIC and the precision and scale of the two arguments are different, the precision and scale of the return type will be set to achieve the maximum precision possible. For example, if the input arguments are DECIMAL(6,3), DECIMAL(7,4), and DECIMAL(8,7), the return type would need three digits to the left of the decimal point and seven digits to the right of the decimal point to avoid any reduction in precision. In this case, the return data type is set to DECIMAL(10,7)

  In cases where it is not possible to maintain the maximum precision, the data will be rounded according to the DBS Control Record RoundHalfWayMagUp field. For example, if the two arguments are DECIMAL(32, 8) and DECIMAL(30, 28), the return type will be DECIMAL(38,14). This will allow for 24 digits to the left of the decimal point (required for the DECIMAL(32,8) argument), and 14 digits to the right of the decimal point.

  If the data type is fixed point NUMBER and the precision is less than or equal to 38, the precision and scale of the return type are calculated with the same method used for DECIMAL/NUMERIC. However, if the precision is greater than 38, the return type is changed to NUMBER(*) to avoid loss of accuracy. If the data type is floating point NUMBER, the return type is NUMBER(*).
- If *expr2* and *expr3* are character types, the function converts the second argument to the data type of the first argument and returns the type as VARCHAR.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Example 1

The following query:

```
SELECT NVL2('England', 'France', 'Spain');
```

returns the second argument, 'France', because the first argument is not NULL.

Example 2

The following query:

```
SELECT NVL2(NULL, 'France', 'Spain');
```

returns the third argument, 'Spain', because the first argument is NULL.

CHAPTER 21 Ordered Analytical / Window Aggregate Functions

This chapter describes

- Ordered analytical functions
- Window Aggregate Functions

Ordered Analytical Functions

Ordered analytical functions provide support for many common operations in analytical processing and data mining that require an ordered set of results rows or depend on values in a previous row. Ordered analytical functions enable and expedite the processing of queries containing On Line Analytical Processing (OLAP) style decision support requests.

For example, computing a seven-day running sum requires:

- First, that rows be ordered by date.
- Then, that the value for the running sum be computed by:
  - Adding the current row value to the value of the sum from the previous row, and
  - Subtracting the value from the row eight days ago.

Benefits

Ordered analytical functions extend the Teradata Database query execution engine with the concept of an ordered set and with the ability to use the values from multiple rows in computing a new value.

The result of an ordered analytical function is handled the same as any other SQL expression. It can be a result column or part of a more complex arithmetic expression within its SELECT.

Each of the ordered analytical functions permit you to specify the sort ordering column or columns on which to sort the rows retrieved by the SELECT statement. The sort order and any other input parameters to the functions are specified the same as arguments to other SQL functions and can be any normal SQL expression.

Ordered Analytical Calculations at the SQL Level

Performing ordered analytical computations at the SQL level rather than through a higher-level OLAP calculation engine provides four distinct advantages.

- Reduced programming effort.
- Elimination of the need for external sort routines.

- Elimination of the need to export large data sets to external tools because ordered analytical functions enable you to target the specific data for analysis within the warehouse itself by specifying conditions in the query.
- Marked enhancement of analysis performance over the slow, single-threaded operations that external tools perform on large data sets.

Teradata Warehouse Miner

You need not directly code SQL queries to take advantage of ordered analytical functions. Both Teradata Database and many third-party query management and analytical tools have full access to the Teradata SQL ordered analytical functions. Teradata Warehouse Miner, for example, a tool that performs data mining preprocessing inside the database engine, relies on these features to perform functions in the database itself rather than requiring data extraction.

Teradata Warehouse Miner includes approximately 40 predefined data mining functions in SQL based on the Teradata SQL-specific functions. For example, the Teradata Warehouse Miner FREQ function uses the Teradata SQL-specific functions CSUM, RANK, and QUALIFY to determine frequencies.

Example

The following example shows how the SQL query to calculate a frequency of gender to marital status would appear using Teradata Warehouse Miner.

```
SELECT gender, marital_status, xcnt,xpct
   ,CSUM(xcnt, xcnt DESC, gender, marital_status) AS xcum_cnt
   ,CSUM(xpct, xcnt DESC, gender, marital_status) AS xcum_pct
   ,RANK(xcnt DESC, gender ASC, marital_status ASC) AS xrank
FROM
   (SELECT gender, marital_status, COUNT(*) AS xcnt
      ,100.000 * xcnt / xall (FORMAT 'ZZ9.99') AS xpct
   FROM customer_table A,
      (SELECT COUNT(*) AS xall
      FROM customer_table) B
GROUP BY gender, marital_status, xall
HAVING xpct >= 1) T1
QUALIFY xrank <= 8
ORDER BY xcnt DESC, gender, marital_status
```

The result for this query looks like the following table.

gender	marital_status	xcnt	xpct	xcum_cnt	xcum_pct	xrank
F	Married	3910093	36.71	3910093	36.71	1
M	Married	2419511	22.71	6329604	59.42	2
F	Divorced	1612130	15.13	7941734	74.55	3
M	Divorced	1412624	3.26	9354358	87.81	4
F	Single	491224	4.61	9845582	92.42	5
F	Widowed	319881	3.01	10165463	95.43	6

gender	marital_status	xcnt	xpct	xcum_cnt	xcum_pct	xrank
M	Single	319794	3.00	10485257	98.43	7
M	Widowed	197131	1.57	10652388	100.00	8

Characteristics of Ordered Analytical Functions

The Function Value

The function value for a column in a row considers that row (and a subset of all other rows in the group) and produces a new value.

The generic function describing this operation is as follows:

```
new_column_value = FUNCTION(column_value,rows_defined_by_window)
```

Use of QUALIFY Clause

Rows can be eliminated by applying conditions on the new column value. The QUALIFY clause is analogous to the HAVING clause of aggregate functions. The QUALIFY clause eliminates rows based on the function value, returning a new value for each of the participating rows. For example:

```
SELECT StoreID, SUM(profit) OVER (PARTITION BY StoreID)
FROM facts
QUALIFY SUM(profit) OVER (PARTITION BY StoreID) > 2;
```

An SQL query that contains both ordered analytical functions and aggregate functions can have both a QUALIFY clause and a HAVING clause, as in the following example:

```
SELECT StoreID, SUM(sale),
SUM(profit) OVER (PARTITION BY StoreID)
FROM facts
GROUP BY StoreID, sale, profit
HAVING SUM(sale) > 15
QUALIFY SUM(profit) OVER (PARTITION BY StoreID) > 2;
```

For details on the QUALIFY clause, see *SQL Data Manipulation Language*.

DISTINCT Clause Restriction

The DISTINCT clause is not permitted in window aggregate functions.

Permitted Query Objects

Ordered analytical functions are permitted in the following database query objects:

- Views
- Macros
- Derived tables
- INSERT ... SELECT

Where Ordered Analytical Functions are Not Permitted

Ordered analytical functions are not permitted in:

- Subqueries
- WHERE clauses
- SELECT AND CONSUME statements

Use of Standard SQL Features

You can use standard SQL features within the same query to make your statements more sophisticated.

For example, you can use ordered analytical functions in the following ways:

Use an analytical function in this operation ...	To ...
INSERT … SELECT	populate a new column.
derived table	create a new table to participate in a complex query.

Ordered analytical functions having different sort expressions are evaluated one after another, reusing the same spool file. Different functions having the same sort expression are evaluated simultaneously.

Unsupported Data Types

Ordered analytical functions do not operate on the following data types:

- CLOB or BLOB data types
- UDT data types

Note that CLOB, BLOB, or UDT data types are usable inside an expression if the result is a supported data type. For example:

```
SELECT
RANK() OVER
(PARTITION BY(CASE WHEN b IS NULL THEN 1 ELSE 0 END) ORDER BY id)
FROM btab;
```

However, the following example results in an error because the function cannot sort by BLOB:

```
SELECT
RANK() OVER
(PARTITION BY b ORDER BY id)
FROM btab;
```

Ordered Analytical Functions and Period Data Types

Expressions that evaluate to Period data types can be specified for any expression within the following ordered analytical functions: QUANTILE, RANK (Teradata-specific function), and RANK(ANSI SQL Window function).

Ordered Analytical Functions and Recursive Queries

Ordered analytical functions cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify an ordered analytical function.

Ordered Analytical Functions and Hash or Join Indexes

When a single table query specifies an ordered analytical function on columns that are also defined for a single table compressed hash or join index, the Optimizer does not select the hash or join index to process the query.

Ordered Analytical Functions and Row Level Security Tables

When a request that includes an ordered analytical function, such as MAVG, CSUM, or RANK, references a table protected by row level security, the operation is based on only the rows that are accessible to the requesting user. In order to apply all rows of the table to the function, the user must have one of the following:

- The required security credentials to access all rows of the table.
- The required OVERRIDE privileges on the security constraints in the table.

For more information about row level security, see *Security Administration*.

Computation Sort Order and Result Order

The sort order that you specify in the window specification defines the sort order of the rows over which the function is applied; it does not define the ordering of the results.

For example, to compute the average sales for the months following the current month, order the rows by month:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID ORDER BY SMonth
                 ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
FROM sales_tbl;

StoreID  SMonth  ProdID      Sales  Remaining Avg(Sales)
-------  ------  ------  ---------  --------------------
   1001       6  C        30000.00                     ?
   1001       5  C        30000.00              30000.00
   1001       4  C        25000.00              30000.00
   1001       3  C        40000.00              28333.33
   1001       2  C        25000.00              31250.00
   1001       1  C        35000.00              30000.00
```

The default sort order is ASC for the computation. However, the results are returned in the reverse order.

To order the results, use an ORDER BY phrase in the SELECT statement. For example:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID ORDER BY SMonth
                 ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
```

```
FROM sales_tbl
ORDER BY SMonth;

StoreID  SMonth  ProdID      Sales  Remaining Avg(Sales)
-------  ------  ------  ---------  --------------------
   1001       1  C        35000.00              30000.00
   1001       2  C        25000.00              31250.00
   1001       3  C        40000.00              28333.33
   1001       4  C        25000.00              30000.00
   1001       5  C        30000.00              30000.00
   1001       6  C        30000.00                     ?
```

Data in Partitioning Column of Window Specification and Resource Impact

The columns specified in the PARTITION BY clause of a window specification determine the partitions over which the ordered analytical function executes. For example, the following query specifies the StoreID column in the PARTITION BY clause to compute the group sales sum for each store:

```
SELECT StoreID, SMonth, ProdID, Sales,
SUM(Sales) OVER (PARTITION BY StoreID)
FROM sales_tbl;
```

At execution time, Teradata Database moves all of the rows that fall into a partition to the same AMP. If a very large number of rows fall into the same partition, the AMP can run out of spool space. For example, if the sales_tbl table in the preceding query has millions or billions of rows, and the StoreID column contains only a few distinct values, an enormous number of rows are going to fall into the same partition, potentially resulting in out-of-spool errors.

To avoid this problem, examine the data in the columns of the PARTITION BY clause. If necessary, rewrite the query to include additional columns in the PARTITION BY clause to create smaller partitions that Teradata Database can distribute more evenly among the AMPs. For example, the preceding query can be rewritten to compute the group sales sum for each store for each month:

```
SELECT StoreID, SMonth, ProdID, Sales,
SUM(Sales) OVER (PARTITION BY StoreID, SMonth)
FROM sales_tbl;
```

Using Ordered Analytical Functions

Example 1: Using RANK and AVG

Consider the result of the following SELECT statement using the following ordered analytical functions, RANK and AVG.

```
SELECT item, smonth, sales,
RANK() OVER (PARTITION BY item ORDER BY sales DESC),
AVG(sales) OVER (PARTITION BY item
                 ORDER BY smonth
                 ROWS 3 PRECEDING)
FROM sales_tbl
ORDER BY item, smonth;
```

The results table might look like the following:

Item	SMonth	Sales	Rank(Sales)	Moving Avg(Sales)
A	1996-01	110	13	110
A	1996-02	130	10	120
A	1996-03	170	6	137
A	1996-04	210	3	155
A	1996-05	270	1	195
A	1996-06	250	2	225
A	1996-07	190	4	230
A	1996-08	180	5	222
A	1996-09	160	7	195
A	1996-10	140	9	168
A	1996-11	150	8	158
A	1996-12	120	11	142
A	1997-01	120	11	132
B	1996-02	30	5	30
...	...	...	...	...

Example 2: Using QUALIFY With RANK

Adding a QUALIFY clause to a query eliminates rows from an unqualified table.

For example, if you wanted to see whether the high sales months were unusual, you could add a QUALIFY clause to the previous query.

```
SELECT item, smonth, sales,
RANK() OVER (PARTITION BY item ORDER BY sales DESC),
AVG(sales) OVER (PARTITION BY item ORDER BY smonth ROWS 3 PRECEDING)
FROM sales_tbl
ORDER BY item, smonth
QUALIFY RANK() OVER(PARTITION BY item ORDER BY sales DESC) <=5;
```

This additional qualifier produces a results table that might look like the following:

Item	SMonth	Sales	Rank(Sales)	Moving Avg(Sales)
A	1996-04	210	3	155
A	1996-05	270	1	195
A	1996-06	250	2	225

Item	SMonth	Sales	Rank(Sales)	Moving Avg(Sales)
A	1996-07	190	4	230
A	1996-08	180	5	222
B	1996-02	30	1	30
...	...	...	...	...

The result indicates that sales had probably been fairly low prior to the start of the current sales season.

Example 3: Using QUALIFY With RANK

Consider the following sales table named sales_tbl.

Store	ProdID	Sales
1003	C	20000.00
1003	D	50000.00
1003	A	30000.00
1002	C	35000.00
1002	D	25000.00
1002	A	40000.00
1001	C	60000.00
1001	D	35000.00
1001	A	100000.00
1001	B	10000.00

Now perform the following simple SELECT statement against this table, qualifying answer rows by rank.

```
SELECT store, prodID, sales,
RANK() OVER (PARTITION BY store ORDER BY sales DESC)
FROM sales_tbl
QUALIFY RANK() OVER (PARTITION BY store ORDER BY sales DESC) <=3;
```

The result appears in the following typical output table.

Store	ProdID	Sales	Rank(Sales)
1001	A	100000.00	1
1001	C	60000.00	2

Store	ProdID	Sales	Rank(Sales)
1001	D	35000.00	3
1002	A	40000.00	1
1002	C	35000.00	2
1002	D	25000.00	3
1003	D	50000.00	1
1003	A	30000.00	2
1003	C	20000.00	3

Note that every row in the table is returned with the computed value for RANK except those that do not meet the QUALIFY clause (sales rank is less than third within the store).

The Window Feature

The ANSI SQL:2011 window feature provides a way to dynamically define a subset of data, or *window*, in an ordered relational database table. A window is specified by the OVER() phrase, which can include the following clauses inside the parentheses:

- PARTITION BY
- ORDER BY
- RESET WHEN
- ROWS

To see the syntax for the OVER() phrase and the associated clauses, see "Window Aggregate Functions" on page 960.

The window specification can also be applied to a user-defined aggregate function. For details, see "Window Aggregate UDF" on page 1302.

PARTITION BY Phrase

PARTITION BY takes a column reference list and groups the rows based on the specified column reference list over which the ordered analytical function executes. Such a grouping is static. To define a group or partition based on a condition, use the RESET WHEN phrase. See "RESET WHEN Phrase" on page 955 for details.

If there is no PARTITION BY phrase or RESET WHEN phrase, then the entire result set, delivered by the FROM clause, constitutes a single partition, over which the ordered analytical function executes.

Consider the following table named sales_tbl.

StoreID	SMonth	ProdID	Sales
1001	1	C	35000.00
1001	2	C	25000.00
1001	3	C	40000.00
1001	4	C	25000.00
1001	5	C	30000.00
1001	6	C	30000.00
1002	1	C	40000.00
1002	2	C	35000.00
1002	3	C	110000.00
1002	4	C	60000.00
1002	5	C	35000.00
1002	6	C	100000.00

The following SELECT statement, which does not include PARTITION BY, computes the average sales for all the stores in the table:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER ()
FROM sales_tbl;

StoreID  SMonth  ProdID      Sales  Group Avg(Sales)
-------  ------  ------  ---------  ----------------
   1001       1  C        35000.00          47083.33
   1001       2  C        25000.00          47083.33
   1001       3  C        40000.00          47083.33
   1001       4  C        25000.00          47083.33
   1001       5  C        30000.00          47083.33
   1001       6  C        30000.00          47083.33
   1002       1  C        40000.00          47083.33
   1002       2  C        35000.00          47083.33
   1002       3  C       110000.00          47083.33
   1002       4  C        60000.00          47083.33
   1002       5  C        35000.00          47083.33
   1002       6  C       100000.00          47083.33
```

To compute the average sales for each store, partition the data in sales_tbl by StoreID:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID)
FROM sales_tbl;
```

```
StoreID  SMonth  ProdID      Sales  Group Avg(Sales)
-------  ------  ------  ---------  ----------------
   1001       3  C        40000.00          30833.33
   1001       5  C        30000.00          30833.33
   1001       6  C        30000.00          30833.33
   1001       4  C        25000.00          30833.33
   1001       2  C        25000.00          30833.33
   1001       1  C        35000.00          30833.33
   1002       3  C       110000.00          63333.33
   1002       5  C        35000.00          63333.33
   1002       6  C       100000.00          63333.33
   1002       4  C        60000.00          63333.33
   1002       2  C        35000.00          63333.33
   1002       1  C        40000.00          63333.33
```

ORDER BY Phrase

ORDER BY specifies how the rows are ordered in a partition, which determines the sort order of the rows over which the function is applied.

To add the monthly sales for a store in the sales_tbl table to the sales for previous months, compute the cumulative sales sum and order the rows in each partition by SMonth:

```
SELECT StoreID, SMonth, ProdID, Sales,
SUM(Sales) OVER (PARTITION BY StoreID ORDER BY SMonth
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM sales_tbl;
```

```
StoreID  SMonth  ProdID      Sales  Cumulative Sum(Sales)
-------  ------  ------  ---------  ---------------------
   1001       1  C        35000.00               35000.00
   1001       2  C        25000.00               60000.00
   1001       3  C        40000.00              100000.00
   1001       4  C        25000.00              125000.00
   1001       5  C        30000.00              155000.00
   1001       6  C        30000.00              185000.00
   1002       1  C        40000.00               40000.00
   1002       2  C        35000.00               75000.00
   1002       3  C       110000.00              185000.00
   1002       4  C        60000.00              245000.00
   1002       5  C        35000.00              280000.00
   1002       6  C       100000.00              380000.00
```

RESET WHEN Phrase

RESET WHEN is a Teradata extension to the ANSI SQL standard.

Depending on the evaluation of the specified condition, RESET WHEN determines the group or partition, over which the ordered analytical function operates. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. To define a partition based on a column reference list, use the PARTITION BY phrase. See "PARTITION BY Phrase" on page 953 for details.

If there is no RESET WHEN phrase or PARTITION BY phrase, then the entire result set, delivered by the FROM clause, constitutes a single partition, over which the ordered analytical function executes.

You can have different RESET WHEN clauses in the same SELECT list.

Note: A window specification that specifies a RESET WHEN clause must also specify an ORDER BY clause.

RESET WHEN Condition Rules

The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify conditional partitioning.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

The following rules apply for RESET WHEN conditions.

A RESET WHEN condition can contain the following:

- Ordered analytical functions that do not include the RESET WHEN clause
- Scalar subqueries
- Aggregate operators
- DEFAULT functions

  However, DEFAULT without an explicit column specification is valid only if it is specified as a standalone condition in the predicate. See "Rules For Using a DEFAULT Function As Part of a RESET WHEN Condition" on page 956 for details.

A RESET WHEN condition *cannot* contain the following:

- Ordered analytical functions that include the RESET WHEN clause
- The SELECT statement
- LOB columns
- UDT expressions, including UDFs that return a UDT value

  However, a RESET WHEN condition can include an expression that contains UDTs as long as that expression returns a result that has a predefined data type.

Rules For Using a DEFAULT Function As Part of a RESET WHEN Condition

The following rules apply to the use of the DEFAULT function as part of a RESET WHEN condition:

- You can specify a DEFAULT function with a column name argument within a predicate. The system evaluates the DEFAULT function to the default value of the column specified as its argument. After the system evaluates the DEFAULT function, it treats it like a literal in the predicate.

- You can specify a DEFAULT function without a column name argument within a predicate only if there is one column specification and one DEFAULT function as the terms on each side of the comparison operator within the expression.
- Following existing comparison rules, a condition with a DEFAULT function used with comparison operators other than IS [NOT] NULL is unknown if the DEFAULT function evaluates to null.

  A condition other than IS [NOT]NULL with a DEFAULT function compared with a null evaluates to unknown.

IF a DEFAULT function is used with...	THEN the comparison is...
IS NULL	TRUE if the default is null, else it is FALSE.
IS NOT NULL	FALSE if the default is null, else it is TRUE.

See "DEFAULT" on page 293 for more information about the DEFAULT function.

Example 1

This example finds cumulative sales for all periods of increasing sales for each region.

```
SUM(sales) OVER (
     PARTITION BY region
     ORDER BY day_of_calendar
     RESET WHEN sales < /* preceding row */ SUM(sales) OVER (
          PARTITION BY region
          ORDER BY day_of_calendar
          ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
     ROWS UNBOUNDED PRECEDING
)
```

Example 2

This example finds sequences of increasing balances. This implies that we reset whenever the current balance is less than or equal to the preceding balance.

```
SELECT account_key, month, balance,
ROW_NUMBER() over
     (PARTITION BY account_key
      ORDER BY month
      RESET WHEN balance /* current row balance */ <=
      SUM(balance) over (PARTITION BY account_key ORDER BY month
      ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING) /* prev row */
      ) - 1 /* to get the count started at 0 */ as balance_increase
FROM accounts;
```

The possible results of the preceding SELECT appear in the table below:

```
account_key        month      balance  balance_increase
-----------        -----      -------  ----------------
          1            1           60                 0
          1            2           99                 1
```

```
          1            3           94                  0
          1            4           90                  0
          1            5           80                  0
          1            6           88                  1
          1            7           90                  2
          1            8           92                  3
          1            9           10                  0
          1           10           60                  1
          1           11           80                  2
          1           12           10                  0
```

Example 3

The following example illustrates a window function with a nested aggregate. The query is processed as follows:

1. We use the SUM(balance) aggregate function to calculate the sum of all the balances for a given account in a given quarter.
2. We check to see if a balance in a given quarter (for a given account) is greater than the balance of the previous quarter.
3. If the balance increased, we track a cumulative count value. As long as the RESET WHEN condition evaluates to false, the balance is increasing over successive quarters, and we continue to increase the count.
4. We use the ROW_NUMBER() ordered analytical function to calculate the count value. When we reach a quarter whose balance is less than or equal to that of the previous quarter, the RESET WHEN condition evaluates to true, and we start a new partition and ROW_NUMBER() restarts the count from 1. We specify ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING to access the previous value.
5. Finally, we subtract 1 to ensure that the count values start with 0.

The balance_increase column shows the number of successive quarters where the balance was increasing. In this example, we only have one quarter (1->2) where the balance has increased.

```
   SELECT account_key, quarter, sum(balance),
   ROW_NUMBER() over
        (PARTITION BY account_key
         ORDER BY quarter
         RESET WHEN sum(balance) /* current row balance */ <=
         SUM(sum(balance)) over (PARTITION BY account_key ORDER BY
quarter
         ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)/* prev row */
         ) - 1 /* to get the count started at 0 */ as balance_increase
   FROM accounts
   GROUP BY account_key, quarter;
```

The possible results of the preceding SELECT appear in the table below:

```
account_key    quarter    balance    balance_increase
-----------    -------    -------    ----------------
          1          1        253                   0
          1          2        258                   1
          1          3        192                   0
          1          4        150                   0
```

Example 4

In the following example, the condition in the RESET WHEN clause contains SELECT as a nested subquery. This is not allowed and results in an error.

```
SELECT SUM(a1) OVER
     (ORDER BY 1
      RESET WHEN 1 in (SELECT 1))
FROM t1;
$
*** Failure 3706 Syntax error: SELECT clause not supported in
RESET...WHEN clause.
```

ROWS Phrase

ROWS defines the rows over which the aggregate function is computed for each row in the partition.

If ROWS is specified, the computation of the aggregate function for each row in the partition includes only the subset of rows in the ROWS phrase.

If there is no ROWS phrase, then the computation includes all the rows in the partition.

To compute the three-month moving average sales for each store in the sales_tbl table, partition by StoreID, order by SMonth, and perform the computation over the current row and the two preceding rows:

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID
                 ORDER BY SMonth
                 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM sales_tbl;

StoreID  SMonth  ProdID      Sales  Moving Avg(Sales)
-------  ------  ------  ---------  -----------------
   1001       1  C        35000.00           35000.00
   1001       2  C        25000.00           30000.00
   1001       3  C        40000.00           33333.33
   1001       4  C        25000.00           30000.00
   1001       5  C        30000.00           31666.67
   1001       6  C        30000.00           28333.33
   1002       1  C        40000.00           40000.00
   1002       2  C        35000.00           37500.00
   1002       3  C       110000.00           61666.67
   1002       4  C        60000.00           68333.33
   1002       5  C        35000.00           68333.33
   1002       6  C       100000.00           65000.00
```

Multiple Window Specifications

In an SQL statement using more than one window function, each window function can have a unique window specification.

For example,

```
SELECT StoreID, SMonth, ProdID, Sales,
AVG(Sales) OVER (PARTITION BY StoreID
```

```
                    ORDER BY SMonth
                    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW),
RANK() OVER (PARTITION BY StoreID ORDER BY Sales DESC)
FROM sales_tbl;
```

Window Aggregate Functions

An aggregate function on which a window specification is applied is called a window aggregate function. Without a window specification, aggregate functions return one value for all qualified rows examined. Window aggregate functions return a new value for each of the qualifying rows participating in the query.

Thus, the following SELECT statement, which includes the aggregate AVG, returns one value only: the average of sales.

```
SELECT AVG(sale)
FROM monthly_sales;

Average(sale)
-------------
         1368
```

The AVG window function retains each qualifying row.

The following SELECT statement might return the results that follow.

```
SELECT territory, smonth, sales,
AVG(sales) OVER (PARTITION BY territory
                 ORDER BY smonth ROWS 2 PRECEDING)
FROM sales_history;

territory  smonth   sales  Moving Avg(sales)
---------  -------  -----  -----------------
East        199810     10                 10
East        199811      4                  7
East        199812     10                  8
East        199901      7                  7
East        199902     10                  9
West        199810      8                  8
West        199811     12                 10
West        199812      7                  9
West        199901     11                 10
West        199902      6                  8
```

For ANSI SQL:2011 compliant aggregate functions to which a window specification can be applied, see Chapter 2: “Aggregate Functions.”

User-defined aggregate function to which a window specification can be applied, see “Window Aggregate UDF” on page 1302.

The Window Specification

Purpose

Cumulative, group, moving, or remaining computation of an aggregate function.

Syntax

window

where:

Syntax element ...	Specifies ...
AVG CORR COUNT COVAR_POP COVAR_SAMP MAX MIN REGR_AVGX REGR_AVGY REGR_COUNT REGR_INTERCEPT REGR_R2 REGR_SLOPE REGR_SXX REGR_SXY REGR_SYY STDDEV_POP STDDEV_SAMP SUM VAR_POP VAR_SAMP	the aggregate function and arguments on which the window specification is applied. For descriptions of aggregate functions and arguments, see Chapter 2: "Aggregate Functions."
OVER	how values are grouped, ordered, and considered when computing the cumulative, group, or moving function. Values are grouped according to the PARTITION BY and RESET WHEN clauses, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.
PARTITION BY	in its *column_reference*, or comma-separated list of column references, the group, or groups, over which the function operates. PARTITION BY is optional. If there are no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a single group, or partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the order in which the values in a group, or partition, are sorted.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
NULLS FIRST	that NULL results are to be listed first.
NULLS LAST	that NULL results are to be listed last.

Syntax element ...	Specifies ...
RESET WHEN	the group or partition, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set, delivered by the FROM clause, constitutes a single partition. If RESET WHEN is specified, then the ORDER BY clause must be specified also.
condition	a conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition. The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions. For more information, see "RESET WHEN Condition Rules" on page 956 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.
ROWS	the starting point for the aggregation group within the partition. The aggregation group end is the current row. The aggregation group of a row R is a set of rows, defined relative to R in the ordering of the rows within the partition. If there are no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. The default when there is no ROWS clause for FIRST_VALUE/ LAST_VALUE is different. See "FIRST_VALUE / LAST_VALUE" on page 993.
ROWS BETWEEN	the aggregation group start and end, which defines a set of rows relative to the current row in the ordering of the rows within the partition. The row specified by the group start must precede the row specified by the group end. If there are no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.
UNBOUNDED PRECEDING	the entire partition preceding the current row.
UNBOUNDED FOLLOWING	the entire partition following the current row.
CURRENT ROW	the start or end of the aggregation group as the current row.

Syntax element ...	Specifies ...
value PRECEDING	the number of rows preceding the current row. The value for *value* is always a positive integer literal. The maximum number of rows in an aggregation group is 4096 when *value* PRECEDING appears as the group start or group end.
value FOLLOWING	the number of rows following the current row. The value for *value* is always a positive integer literal. The maximum number of rows in an aggregation group is 4096 when *value* FOLLOWING appears as the group start or group end.

ANSI Compliance

Window aggregate functions are partially ANSI SQL:2011 compliant.

In the presence of an ORDER BY clause and the absence of a ROWS or ROWS BETWEEN clause, ANSI SQL:2011 window aggregate functions use ROWS UNBOUNDED PRECEDING as the default aggregation group, whereas Teradata SQL window aggregate functions use ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

Type of Computation

Arguments to Window Aggregate Functions

To compute this type of function ...	Use this aggregation group ...
Cumulative	• ROWS UNBOUNDED PRECEDING • ROWS BETWEEN UNBOUNDED PRECEDING AND *value* PRECEDING • ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW • ROWS BETWEEN UNBOUNDED PRECEDING AND *value* FOLLOWING
Group	ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
Moving	• ROWS *value* PRECEDING • ROWS CURRENT ROW • ROWS BETWEEN *value* PRECEDING AND *value* PRECEDING • ROWS BETWEEN *value* PRECEDING AND CURRENT ROW • ROWS BETWEEN *value* PRECEDING AND *value* FOLLOWING • ROWS BETWEEN CURRENT ROW AND CURRENT ROW • ROWS BETWEEN CURRENT ROW AND *value* FOLLOWING • ROWS BETWEEN *value* FOLLOWING AND *value* FOLLOWING
Remaining	• ROWS BETWEEN *value* PRECEDING AND UNBOUNDED FOLLOWING • ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING • ROWS BETWEEN *value* FOLLOWING AND UNBOUNDED FOLLOWING

Window aggregate functions can take literals, literal expressions, column names (sales, for example), or column expressions (sales + profit) as arguments.

Window aggregates can also take regular aggregates as input parameters to the PARTITION BY and ORDER BY clauses. The RESET WHEN clause can take an aggregate as part of the RESET WHEN condition clause.

COUNT can take "*" as an input argument, as in the following SQL query:

```
SELECT city, kind, sales, profit,
COUNT(*) OVER (PARTITION BY city, kind
               ROWS BETWEEN UNBOUNDED PRECEDING AND
               UNBOUNDED FOLLOWING)
FROM activity_month;
```

Result Type and Format

The result data type and format for window aggregate functions are as follows.

Function	Result Type	Format
AVG(*x*) where *x* is a character type	FLOAT	Default format for FLOAT
AVG(*x*) where *x* is a numeric, DATE, or INTERVAL type	FLOAT	Same format as operand *x*
CORR(*x*,*y*) COVAR_POP(*x*,*y*) COVAR_SAMP(*x*,*y*) REGR_AVGX(*y*,*x*) REGR_AVGY(*y*,x) REGR_INTERCEPT(*x*,*y*) REGR_R2(*x*,*y*) REGR_SLOPE(*x*,*y*) REGR_SXX(*x*,*y*) REGR_SXY(*x*,*y*) REGR_SYY(*x*,*y*) STDDEV_POP(*x*) STDDEV_SAMP(*x*) VAR_POP(*x*) VAR_SAMP(*x*) where *x* is a character type	FLOAT	Default format for FLOAT
CORR(*x*,*y*) COVAR_POP(*x*,*y*) COVAR_SAMP(*x*,*y*) REGR_AVGX(*y*,*x*) REGR_AVGY(*y*,*x*) REGR_INTERCEPT(*x*,*y*) REGR_R2(*x*,*y*) REGR_SLOPE(*x*,*y*) REGR_SXX(*x*,*y*) REGR_SXY(*x*,*y*) REGR_SYY(*x*,*y*) STDDEV_POP(*x*) STDDEV_SAMP(*x*) VAR_POP(*x*) VAR_SAMP(*x*) where *x* is one of the following types: • Numeric • DATE • Interval	Same data type as operand *x*.	Default format for the data type of operand *x*
REGR_AVGX(*y*,*x*) REGR_AVGY(*y*, *x*) where x is a UDT		Default format for the data type to which the UDT is implicitly cast.

Function	Result Type	Format
COUNT(*x*) COUNT(*) REGR_COUNT(*x*,*y*) where the transaction mode is ANSI	If MaxDecimal in DBSControl is… • 0 or 15, then the result type is DECIMAL(15,0) and the format is -(15)9. • 18, then the result type is DECIMAL(18,0) and the format is -(18)9. • 38, then the result type is DECIMAL(38,0) and the format is -(38)9. ANSI transaction mode uses DECIMAL because tables frequently have a cardinality exceeding the range of INTEGER.	
COUNT(*x*) COUNT(*) REGR_COUNT(*x*,*y*) where the transaction mode is Teradata	INTEGER Teradata transaction mode uses INTEGER to avoid regression problems. Note: You can cast the final result of a COUNT window aggregate function; however, the cast is not used as part of the window function computation as it is for the COUNT aggregate function and, therefore, cannot be used to avoid numeric overflow errors that might occur during the computation.	Default format for INTEGER
MAX(*x*), MIN(*x*)	Same data type as operand *x*.	Same format as operand *x*
SUM(*x*) where *x* is a character type	Same as operand *x*.	Default format for FLOAT
SUM(*x*) where *x* is a DECIMAL(*n*,*m*) type	DECIMAL(*p*,*m*), where *p* is determined according to the following rules: If MaxDecimal in DBSControl is 0 or 15 and • $n \leq 15$, then $p = 15$. • $15 < n \leq 18$, $p = 18$. • $n > 18$, then $p = 38$. If MaxDecimal in DBSControl is 18 and • $n \leq 18$, then $p = 18$. • $n > 18$, then $p = 38$. If MaxDecimal in DBSControl is 38 and n = any value, the $p = 38$.	Default format for DECIMAL
SUM(*x*) where *x* is any numeric type other than DECIMAL	Same as operand *x*.	Default format for the data type of the operand

For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Result Title

The default title that appears in the heading for displayed or printed results depends on the type of computation performed.

IF the type of computation is ...	THEN the result title is ...

cumulative	Cumulative *Function_name* (*argument_list*)
	For example, consider the following computation:
	`SELECT AVG(sales) OVER (PARTITION BY region ORDER BY smonth ROWS UNBOUNDED PRECEDING) FROM sales_history;`
	The title that appears in the result heading is:
	`Cumulative Avg(sales)`
	`----------`

group	Group *Function_name* (*argument_list*)
	For example, consider the following computation:
	`SELECT AVG(sales) OVER (PARTITION BY region ORDER BY smonth ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) FROM sales_history;`
	The title that appears in the result heading is:
	`Group Avg(sales)`
	`----------`

moving	Moving *Function_name* (*argument_list*)
	For example, consider the following computation:
	`SELECT AVG(sales) OVER (PARTITION BY region ORDER BY smonth ROWS 2 PRECEDING) FROM sales_history;`
	The title that appears in the result heading is:
	`Moving Avg(sales)`
	`----------`

remaining	Remaining *Function_name* (*argument_list*)
	For example, consider the following computation:
	`SELECT AVG(sales) OVER (PARTITION BY region ORDER BY smonth ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) FROM sales_history;`
	The title that appears in the result heading is:
	`Remaining Avg(sales)`
	`----------`

Problems With Missing Data

Ensure that data you analyze has no missing data points. Computing a moving function over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Nesting Aggregates in Window Functions

Although you can nest aggregates in window functions, including the select list, HAVING, QUALIFY, and ORDER BY clauses, the HAVING clause can only contain aggregate function references because HAVING cannot contain nested syntax like RANK() OVER (ORDER BY SUM(x)).

Aggregate functions cannot be specified with Teradata-specific functions.

Example

The following query nests the SUM aggregate function within the RANK ordered analytical function in the select list:

```
SELECT state, city, SUM(sale),
RANK() OVER (PARTITION BY state ORDER BY SUM(sale))
FROM T1
WHERE T1.cityID = T2.cityID
GROUP BY state, city
HAVING MAX(sale) > 10;
```

Alternative: Using Derived Tables

Although only window functions allow aggregates specified together in the same SELECT list, window functions and Teradata-specific functions can be combined with aggregates using derived tables or views. Using derived tables or views also clarifies the semantics of the computation.

Example

The following example shows the sales rank of a particular product in a store and its percent contribution to the store sales for the top three products in each store.

```
SELECT RT.storeid, RT.prodid, RT.sales,
RT.rank_sales, RT.sales * 100.0/ST.sum_store_sales
FROM (SELECT storeid, prodid, sales, RANK(sales) AS rank_sales
FROM sales_tbl
GROUP BY storeID
QUALIFY RANK(sales) <=3) AS RT,
(SELECT storeID, SUM(sales) AS sum_store_sales
FROM sales_tbl
GROUP BY storeID) AS ST
WHERE RT.storeID = ST.storeID
ORDER BY RT.storeID, RT.sales;
```

The results table might look something like the following:

storeID	prodID	sales	rank_sales	sales*100.0/sum_store_sales
1001	D	35000.00	3	17.949
1001	C	60000.00	2	30.769
1001	A	100000.00	1	51.282
1002	D	25000.00	3	25.000

storeID	prodID	sales	rank_sales	sales*100.0/sum_store_sales
1002	C	35000.00	2	35.000
1002	A	40000.00	1	40.000
1003	C	20000.00	3	20.000
1003	A	30000.00	2	30.000
1003	D	50000.00	1	50.000
...	...	...	...	

Teradata-Specific Alternatives to Ordered Analytical Functions

Teradata SQL supports two syntax alternatives for ordered analytical functions:

- Teradata-specific
- ANSI SQL:2011 compliant

Window aggregate, rank, distribution, and row number functions are ANSI SQL:2011 compliant. Teradata-specific functions are not.

Teradata-Specific Functions and ANSI SQL:2011 Window Functions

The following table identifies equivalent ANSI SQL:2011 window functions for Teradata-specific functions:

Note: The use of the Teradata-specific functions listed in the following table is strongly discouraged. These functions are retained only for backward compatibility with existing applications. Be sure to use the ANSI-compliant window functions for any new applications you develop.

Teradata-Specific Functions	Equivalent ANSI SQL:2011 Window Functions
CSUM	SUM
MAVG	AVG
MDIFF(x, w, y)	composable from SUM
MLINREG	composable from SUM and COUNT
QUANTILE	composable from RANK and COUNT
RANK	RANK
MSUM	SUM

Comparing Window Aggregate Functions and Teradata-Specific Functions

Avoid using Teradata-specific functions such as MAVG, CSUM, and MSUM for applications intended to be ANSI-compliant and portable.

ANSI Function	Teradata Function	Relationship
AVG	MAVG	The form of the AVG window function that specifies an aggregation group of ROWS *value* PRECEDING is the ANSI equivalent of the MAVG Teradata-specific function. Note that the ROWS *value* PRECEDING phrase specifies the number of rows preceding the current row that are used, together with the current row, to compute the moving average. The total number of rows in the aggregation group is *value* + 1. For the MAVG function, the total number of rows in the aggregation group is the value of *width*. For AVG window function, an aggregation group of ROWS 5 PRECEDING, for example, means that the 5 rows preceding the current row, plus the current row, are used to compute the moving average. Thus the moving average for the 6th row of a partition would have considered row 6, plus rows 5, 4, 3, 2, and 1 (that is, 6 rows in all). For the MAVG function, a *width* of 5 means that the current row, plus 4 preceding rows, are used to compute the moving average. The moving average for the 6th row would have considered row 6, plus rows 4, 5, 3, and 2 (that is, 5 rows in all).
SUM	CSUM MSUM	Be sure to use the ANSI-compliant SUM window function for any new applications you develop. Avoid using CSUM and MSUM for applications intended to be ANSI-compliant and portable. The following defines the relationship between the SUM window function and the CSUM and MSUM Teradata-specific functions, respectively: • The SUM window function that uses the ORDER BY clause and specifies ROWS UNBOUNDED PRECEDING is the ANSI equivalent of CSUM. • The SUM window function that uses the ORDER BY clause and specifies ROWS *value* PRECEDING is the ANSI equivalent of MSUM. Note that the ROWS *value* PRECEDING phrase specifies the number of rows preceding the current row that are used, together with the current row, to compute the moving average. The total number of rows in the aggregation group is *value* + 1. For the MSUM function, the total number of rows in the aggregation group is the value of *width*. Thus for the SUM window function that computes a moving sum, an aggregation group of ROWS 5 PRECEDING means that the 5 rows preceding the current row, plus the current row, are used to compute the moving sum. The moving sum for the 6th row of a partition, for example, would have considered row 6, plus rows 5, 4, 3, 2, and 1 (that is, 6 rows in all). For the MSUM function, a *width* of 5 means that the current row, plus 4 preceding rows, are used to compute the moving sum. The moving sum for the 6th row, for example, would have considered row 6, plus rows 5, 4, 3, and 2 (that is, 5 rows in all). Moreover, for data having fewer than *width* rows, MSUM computes the sum using all the preceding rows. MSUM returns the current sum rather than nulls when the number of rows in the sample is fewer than *width*.

Example 1: Moving Average

Determine, for a business with several sales territories, the sales in each territory averaged over the current month and the preceding 2 months.

The following query might return the results found in the table that follows it.

```
SELECT territory, smonth, sales,
AVG(sales) OVER (PARTITION BY territory
                 ORDER BY smonth
                 ROWS 2 PRECEDING)
FROM sales_history;

territory  smonth  sales  Moving Avg(sales)
---------  ------  -----  -----------------
East       199810  10     10
East       199811  4      7
East       199812  10     8
East       199901  7      7
East       199902  10     9
West       199810  8      8
West       199811  12     10
West       199812  7      9
West       199901  11     10
West       199902  6      8
```

The meanings of the phrases in the example query are as follows:

Phrase	Meaning
PARTITION BY	Indicates that the rows delivered by the FROM clause, the rows of sales_history, should be assigned to groups, or partitions, based on their territory. If no PARTITION clause is specified, then the entire result set constitutes a single group, or partition.
ORDER BY	Indicates that rows are sorted in ascending order of month within each group, or partition. Ascending is the default sort order.
ROWS 2 PRECEDING	Defines the number of rows used to compute the moving average. In this case, the computation uses the current row and the 2 preceding rows of the group, or partition, as available.

Thus, the moving average for the first row of the partition East (199810), which has no preceding rows, is 10. That is, the value of the first row, the current row (10)/ the number of rows (1) = 10.

The moving average for the second row of the partition East (199811), which has only 1 preceding row, is 7. That is, the value of the second row, the current row, and the preceding row (10 + 4) / the number of rows (2) = 7.

The moving average for the third row of the partition East (199812), which has 2 preceding rows, is 8. That is, the value of the third row, the current row, and the 2 preceding rows (10 + 4 + 10) / the number of rows (3) = 8. And so on.

Month is specified as a six-digit numeric in the YYYYMM format.

Example 2: Group Count

The following SQL query might yield the results that follow it, where the group count for sales is returned for each of the four partitions defined by city and kind. Notice that rows that have no sales are not counted.

```
SELECT city, kind, sales, profit,
COUNT(sales) OVER (PARTITION BY city, kind
                   ROWS BETWEEN UNBOUNDED PRECEDING AND
                   UNBOUNDED FOLLOWING)
FROM activity_month;

city     kind      sales  profit  Group Count(sales)
-------  --------  -----  ------  ------------------
LA       Canvas       45     320                   4
LA       Canvas      125     190                   4
LA       Canvas      125     400                   4
LA       Canvas       20     120                   4
LA       Leather      20      40                   1
LA       Leather       ?       ?                   1
Seattle  Canvas       15      30                   3
Seattle  Canvas       20      30                   3
Seattle  Canvas       20     100                   3
Seattle  Leather      35      50                   1
Seattle  Leather       ?       ?                   1
```

Example 3: Remaining Count

To count all the rows, including rows that have no sales, use COUNT(*). Here is an example that counts the number of rows remaining in the partition after the current row:

```
SELECT city, kind, sales, profit,
COUNT(*) OVER (PARTITION BY city, kind ORDER BY profit DESC
               ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
FROM activity_month;

city     kind      sales  profit  Remaining Count(*)
-------  --------  -----  ------  ------------------
LA       Canvas       20     120                   ?
LA       Canvas      125     190                   1
LA       Canvas       45     320                   2
LA       Canvas      125     400                   3
LA       Leather       ?       ?                   ?
LA       Leather      20      40                   1
Seattle  Canvas       15      30                   ?
Seattle  Canvas       20      30                   1
Seattle  Canvas       20     100                   2
Seattle  Leather       ?       ?                   ?
Seattle  Leather      35      50                   1
```

Note that the sort order that you specify in the window specification defines the sort order of the rows over which the function is applied; it does not define the ordering of the results.

In the example, the DESC sort order is specified for the computation, but the results are returned in the reverse order.

To order the results, use the ORDER BY phrase in the SELECT statement:

```
SELECT city, kind, sales, profit,
```

```
COUNT(*) OVER (PARTITION BY city, kind ORDER BY profit DESC
               ROWS BETWEEN 1 FOLLOWING AND
               UNBOUNDED FOLLOWING)
FROM activity_month
ORDER BY city, kind, profit DESC;
```

```
city     kind      sales  profit  Remaining Count(*)
-------  --------  -----  ------  ------------------
LA       Canvas      125     400                   3
LA       Canvas       45     320                   2
LA       Canvas      125     190                   1
LA       Canvas       20     120                   ?
LA       Leather      20      40                   1
LA       Leather       ?       ?                   ?
Seattle  Canvas       20     100                   2
Seattle  Canvas       20      30                   1
Seattle  Canvas       15      30                   ?
Seattle  Leather      35      50                   1
Seattle  Leather       ?       ?                   ?
```

Example 4: Cumulative Maximum

The following SQL query might yield the results that follow it, where the cumulative maximum value for sales is returned for each partition defined by city and kind.

```
SELECT city, kind, sales, week,
MAX(sales) OVER (PARTITION BY city, kind
                 ORDER BY week ROWS UNBOUNDED PRECEDING)
FROM activity_month;
```

```
city     kind      sales  week  Cumulative Max(sales)
-------  --------  -----  ----  ---------------------
LA       Canvas      263    16                    263
LA       Canvas      294    17                    294
LA       Canvas      321    18                    321
LA       Canvas      274    20                    321
LA       Leather     144    16                    144
LA       Leather     826    17                    826
LA       Leather     489    20                    826
LA       Leather     555    21                    826
Seattle  Canvas      100    16                    100
Seattle  Canvas      182    17                    182
Seattle  Canvas       94    18                    182
Seattle  Leather     933    16                    933
Seattle  Leather     840    17                    933
Seattle  Leather     899    18                    933
Seattle  Leather     915    19                    933
Seattle  Leather     462    20                    933
```

Example 5: Cumulative Minimum

The following SQL query might yield the results that follow it, where the cumulative minimum value for sales is returned for each partition defined by city and kind.

```
SELECT city, kind, sales, week,
MIN(sales) OVER (PARTITION BY city, kind
```

```
                 ORDER BY week
                 ROWS UNBOUNDED PRECEDING)
FROM activity_month;

city     kind      sales  week  Cumulative Min(sales)
-------  --------  -----  ----  ---------------------
LA       Canvas      263    16                    263
LA       Canvas      294    17                    263
LA       Canvas      321    18                    263
LA       Canvas      274    20                    263
LA       Leather     144    16                    144
LA       Leather     826    17                    144
LA       Leather     489    20                    144
LA       Leather     555    21                    144
Seattle  Canvas      100    16                    100
Seattle  Canvas      182    17                    100
Seattle  Canvas       94    18                     94
Seattle  Leather     933    16                    933
Seattle  Leather     840    17                    840
Seattle  Leather     899    18                    840
Seattle  Leather     915    19                    840
Seattle  Leather     462    20                    462
```

Example 6: Cumulative Sum

The following query returns the cumulative balance per account ordered by transaction date:

```
SELECT acct_number, trans_date, trans_amount,
SUM(trans_amount) OVER (PARTITION BY acct_number
                        ORDER BY trans_date
                        ROWS UNBOUNDED PRECEDING) as balance
FROM ledger
ORDER BY acct_number, trans_date;
```

Here are the possible results of the preceding SELECT:

acct_number	trans_date	trans_amount	balance
73829	1998-11-01	113.45	113.45
73829	1988-11-05	-52.01	61.44
73929	1998-11-13	36.25	97.69
82930	1998-11-01	10.56	10.56
82930	1998-11-21	32.55	43.11
82930	1998-11-29	-5.02	38.09

Example 7: Group Sum

The query below finds the total sum of meat sales for each city.

```
SELECT city, kind, sales,
SUM(sales) OVER (PARTITION BY city ROWS BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING) FROM monthly;
```

The possible results of the preceding SELECT appear in the following table:

city	kind	sales	Group Sum (sales)
Omaha	pure pork	45	220
Omaha	pure pork	125	220
Omaha	pure pork	25	220
Omaha	variety pack	25	220
Chicago	variety pack	55	175
Chicago	variety pack	45	175
Chicago	pure pork	50	175
Chicago	variety pack	25	175

Example 8: Group Sum

The following query returns the total sum of meat sales for all cities. Note there is no PARTITION BY clause in the SUM function, so all cities are included in the group sum.

```
SELECT city, kind, sales,
SUM(sales) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND
                 UNBOUNDED FOLLOWING)
FROM monthly;
```

The possible results of the preceding SELECT appear in the table below:

city	kind	sales	Group Sum (sales)
Omaha	pure pork	45	395
Omaha	pure pork	125	395
Omaha	pure pork	25	395
Omaha	variety pack	25	395
Chicago	variety pack	55	395
Chicago	variety pack	45	395
Chicago	pure pork	50	395
Chicago	variety pack	25	395

Example 9: Moving Sum

The following query returns the moving sum of meat sales by city. Notice that the query returns the moving sum of sales by city (the partition) for the current row (of the partition) and three preceding rows where possible.

The order in which each meat variety is returned is the default ascending order according to profit.

Where no sales figures are available, no moving sum of sales is possible. In this case, there is a null in the sum(sales) column.

```
SELECT city, kind, sales, profit,
SUM(sales) OVER (PARTITION BY city, kind
                 ORDER BY profit ROWS 3 PRECEDING)
FROM monthly;
```

city	kind	sales	profit	Moving sum (sales)
Omaha	pure pork	25	40	25
Omaha	pure pork	25	120	50
Omaha	pure pork	45	140	95
Omaha	pure pork	125	190	220
Omaha	pure pork	45	320	240
Omaha	pure pork	1255	400	340
Omaha	variety pack	?	?	?
Omaha	variety pack	25	40	25
Omaha	variety pack	25	120	50
Chicago	pure pork	?	?	?
Chicago	pure pork	15	10	15
Chicago	pure pork	54	12	69
Chicago	pure pork	14	20	83
Chicago	pure pork	54	24	137
Chicago	pure pork	14	34	136
Chicago	pure pork	95	80	177
Chicago	pure pork	95	140	258
Chicago	pure pork	15	220	219
Chicago	variety pack	23	39	23
Chicago	variety pack	25	40	48
Chicago	variety pack	125	70	173
Chicago	variety pack	125	100	298
Chicago	variety pack	23	100	298
Chicago	variety pack	25	120	298

Example 10: Remaining Sum

The following query returns the remaining sum of meat sales for all cities. Note there is no PARTITION BY clause in the SUM function, so all cities are included in the remaining sum.

```
SELECT city, kind, sales,
SUM(sales) OVER (ORDER BY city, kind
```

```
                    ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
   FROM monthly;
```

The possible results of the preceding SELECT appear in the table below:

```
city     kind           sales    Remaining Sum(sales)
-------  -------------  -------  --------------------
Omaha    variety pack   25       ?
Omaha    pure pork      125      25
Omaha    pure pork      25       150
Omaha    pure pork      45       175
Chicago  variety pack   55       220
Chicago  variety pack   25       275
Chicago  variety pack   45       300
Chicago  pure pork      50       345
```

Note that the sort order for the computation is alphabetical by city, and then by kind. The results, however, appear in the reverse order.

The sort order that you specify in the window specification defines the sort order of the rows over which the function is applied; it does not define the ordering of the results. To order the results, use an ORDER BY phrase in the SELECT statement.

For example:

```
   SELECT city, kind, sales,
   SUM(sales) OVER (ORDER BY city, kind
                    ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
   FROM monthly
   ORDER BY city, kind;
```

The possible results of the preceding SELECT appear in the table below:

```
city     kind           sales    Remaining Sum(sales)
-------  -------------  -------  --------------------
Chicago  pure pork      50       345
Chicago  variety pack   55       265
Chicago  variety pack   25       320
Chicago  variety pack   45       220
Omaha    pure pork      25       70
Omaha    pure pork      125      95
Omaha    pure pork      45       25
Omaha    variety pack   25       ?
```

GROUP BY Clause

GROUP BY and Window Functions

For window functions, the GROUP BY clause must include all the columns specified in the:

- Select list of the SELECT clause
- Window functions in the select list of a SELECT clause
- Window functions in the search condition of a QUALIFY clause
- The condition in the RESET WHEN clause

For example, the following SELECT statement specifies the column City in the select list and the column StoreID in the COUNT window function in the select list and QUALIFY clause. Both columns must also appear in the GROUP BY clause:

```
SELECT City, StoreID, COUNT(StoreID) OVER ()
FROM sales_tbl
GROUP BY City, StoreID
QUALIFY COUNT(StoreID) >=3;
```

For window functions, GROUP BY collapses all rows with the same value for the group-by columns into a single row.

For example, the following statement uses the GROUP BY clause to collapse all rows with the same value for City and StoreID into a single row:

```
SELECT City, StoreID, COUNT(StoreID) OVER ()
FROM sales_tbl
GROUP BY City, StoreID;
```

The results look like this:

```
City   StoreID  Group Count(StoreID)
-----  -------  --------------------
Pecos     1001                     3
Pecos     1002                     3
Ozona     1003                     3
```

Without the GROUP BY, the results look like this:

```
City   StoreID  Group Count(StoreID)
-----  -------  --------------------
Pecos     1001                     9
Pecos     1001                     9
Pecos     1001                     9
Pecos     1001                     9
Pecos     1002                     9
Pecos     1002                     9
Pecos     1002                     9
Ozona     1003                     9
Ozona     1003                     9
```

GROUP BY and Teradata-Specific Functions

For Teradata-specific functions, GROUP BY determines the partitions over which the function executes. The clause does *not* collapse all rows with the same value for the group-by columns into a single row. Thus, the GROUP BY clause in these cases need only specify the partitioning column for the function.

For example, the following statement computes the running sales for each store by using the GROUP BY clause to partition the data in sales_tbl by StoreID:

```
SELECT StoreID, Sales, CSUM(Sales, StoreID)
FROM sales_tbl
GROUP BY StoreID;
```

The results look like this:

```
StoreID     Sales  CSum(Sales,StoreID)
-------  --------  -------------------
   1001   1100.00              1100.00
```

```
1001    400.00              1500.00
1001   1000.00              2500.00
1001   2000.00              4500.00
1002    500.00               500.00
1002   1500.00              2000.00
1002   2500.00              4500.00
1003   1000.00              1000.00
1003   3000.00              4000.00
```

Combining Window Functions, Teradata-Specific Functions, and GROUP BY

The following table provides the semantics of the allowable combinations of window functions, Teradata-specific functions, aggregate functions, and the GROUP BY clause.

Combination				Semantics
Window Function	Teradata-Specific Function	Aggregate Function	GROUP BY Clause	
X				A value is computed for each row.
	X			A value is computed for each row. The entire table constitutes a single group, or partition, over which the Teradata-specific function executes.
		X		One aggregate value is computed for the entire table.
X			X	GROUP BY collapses all rows with the same value for the group-by columns into a single row, and a value is computed for each resulting row.
	X		X	GROUP BY determines the partitions over which the Teradata-specific function executes. The clause does *not* collapse all rows with the same value for the group-by columns into a single row.
		X	X	An aggregation is performed for each group.
X	X			Teradata-specific functions do not have partitions. The whole table is one partition.
X	X		X	GROUP BY determines partitions for Teradata-specific functions. GROUP BY does *not* collapse all rows with the same value for the group-by columns into a single row, and does not affect window function computation.

Combination				Semantics
Window Function	Teradata-Specific Function	Aggregate Function	GROUP BY Clause	
X		X	X	GROUP BY collapses all rows with the same value for the group-by columns into a single row. For window functions, a value is computed for each resulting row; for aggregate functions, an aggregation is performed for each group.

Possible Result Overflow with SELECT Sum

When using this function, the result can create an overflow when the data type and format are not in sync. For a column defined as:

```
Salary Decimal(15,2) Format '$ZZZ,ZZ9.99'
```

The following query:

```
SELECT SUM (Salary) FROM Employee;
```

causes an overflow because the decimal operand and the format are not in sync.

To avoid possible overflows, explicitly specify the format for decimal sum to specify a format large enough to accommodate the decimal sum resultant data type.

```
SELECT Sum(Salary) (format '$Z,ZZZ,ZZZ,ZZ9.99) FROM Employee;
```

CSUM

Purpose

Returns the cumulative (or running) sum of a value expression for each row in a partition, assuming the rows in the partition are sorted by the *sort_expression* list.

Type

Teradata-specific function.

Syntax

```
— CSUM —( — value_expression, ┌──────────────, ──────────────┐ ) —
                              │ sort_expression               │
                              │                ┌─ ASC ─┐      │
                              │                └─ DESC ─┘      │
```

1101A398

where:

Syntax element ...	Specifies ...
value_expression	a numeric literal or column expression for which a running sum is to be computed. By default, CSUM uses the default data type of *value_expression*. Larger numeric values are supported by casting it to a higher data type. The expression cannot contain any ordered analytical or aggregate functions.
sort_expression	a literal or column expression or comma-separated list of literal or column expressions to be used to sort the values. For example, CSUM(Sale, Region ASC, Store DESC), where Sale is the *value_expression,* and Region ASC, Store DESC is the *sort_expression* list. The expression cannot contain any ordered analytical or aggregate functions.
ASC	ascending sort order. The default sort direction is ASC.
DESC	descending sort order.

ANSI Compliance

CSUM is a Teradata extension to the ANSI SQL:2011 standard.

Using SUM Instead of CSUM

The use of CSUM is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is equivalent to the ANSI-compliant SUM window function that specifies ROWS UNBOUNDED PRECEDING as its aggregation group. CSUM is retained only for backward compatibility with existing applications.

For more information on the SUM window function, see "Window Aggregate Functions" on page 960.

Meaning of Cumulative Sums

CSUM accumulates a sum over an ordered set of rows, providing the current value of the SUM on each row.

Possible Result Overflow with SELECT Sum

When using this function, the result can create an overflow when the data type and format are not in sync. For a column defined as:

```
Salary Decimal(15,2) Format '$ZZZ,ZZ9.99'
```

The following query:

```
SELECT SUM (Salary) FROM Employee;
```

causes an overflow because the decimal operand and the format are not in sync.

To avoid possible overflows, explicitly specify the format for decimal sum to specify a format large enough to accommodate the decimal sum resultant data type.

```
SELECT Sum(Salary) (format '$Z,ZZZ,ZZZ,ZZ9.99) FROM Employee;
```

Result Type and Attributes

The data type, format, and title for CSUM are as follows:

Data Type: Same as operand x

- If operand x is character, the format is the default format for FLOAT.
- If operand x is numeric, the format is the same format as x.

For information on the default format of data types and an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example 1

Report the daily running sales total for product code 10 for each month of 1998.

```
SELECT cmonth, CSUM(sumPrice, cdate)
FROM
(SELECT a2.month_of_year,
a2.calendar_date,a1.itemID, SUM(a1.price)
FROM Sales a1, SYS_CALENDAR.Calendar a2
WHERE a1.calendar_date=a2.calendar_date
```

```
   AND a2.calendar_date=1998
   AND a1.itemID=10
   GROUP BY a2.month_of_year, a1.calendar_date,
   a1.itemID) AS T1(cmonth, cdate, sumPrice)
   GROUP BY cmonth;
```

Grouping by month allows the total to accumulate until the end of each month, when it is then set to zero for the next month. This permits the calculation of cumulative totals for each item in the same query.

Example 2

Provide a running total for sales of each item in store 5 in January and generate output that is ready to export into a graphing program.

```
   SELECT Item, SalesDate, CSUM(Revenue,Item,SalesDate) AS
CumulativeSales
   FROM
   (SELECT Item, SalesDate, SUM(Sales) AS Revenue
   FROM DailySales
   WHERE StoreId=5 AND SalesDate BETWEEN
   '1/1/1999' AND '1/31/1999'
   GROUP BY Item, SalesDate) AS ItemSales
   ORDER BY SalesDate;
```

The result might like something like the following table:

Item	SalesDate	CumulativeSales
InstaWoof dog food	01/01/1999	972.99
InstaWoof dog food	01/02/1999	2361.99
InstaWoof dog food	01/03/1999	5110.97
InstaWoof dog food	01/04/1999	7793.91

CUME_DIST

Purpose

Calculates the cumulative distribution of a value in a group of values.

Type

ANSI SQL:2011 window function.

Syntax

where:

Syntax element ...	Specifies ...
OVER	how the values, grouped according to the PARTITION BY and RESET WHEN clauses and named by *value_expression* in the ORDER BY clause, are ranked.
PARTITION BY	in its *column_reference* the column or columns, according to which ranking resets. PARTITION BY is optional. If there are no PARTITION BY or RESET WHEN clauses, then the entire result set, specified by the ORDER BY clause, constitutes a single group, or partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the column or columns being ranked.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
NULLS FIRST	that NULL results are to be listed first.

Syntax element ...	Specifies ...
NULLS LAST	that NULL results are to be listed last.
RESET WHEN	the group or partition over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set constitutes a single partition. If RESET WHEN is specified, then the ORDER BY clause must be specified also.
condition	a conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition. The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions. For more information, see "RESET WHEN Condition Rules" on page 956 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Using CUME_DIST

CUME_DIST is similar to PERCENT_RANK. Unlike PERCENT_RANK, which considers the RANK value in the presence of ties, CUME_DIST uses the highest tied rank, that is, the position of the last tied value when there are peers. CUME_DIST is the ratio of that position in the partition (RANK-HIGH/NUM ROWS).

Results

The range of values returned by CUME_DIST is >0 to <=1.

Example

The following SELECT statement:

```
SELECT lname, serviceyrs,
 CUME_DIST()OVER(ORDER BY serviceyrs)
 FROM schooltbl
 GROUP BY 1,2;
```

returns the cumulative distribution by service years for teachers listed in *schooltbl*.

lname	serviceyrs	CUME_DIST
Adams	10	0.333333

lname	serviceyrs	CUME_DIST
Peters	10	0.333333
Murray	10	0.333333
Rogers	15	0.444333
Franklin	16	0.555333
Smith	20	0.888889
Ford	20	0.888889
Derby	20	0.888889
Baker	20	1.000000

DENSE_RANK (ANSI)

Purpose

Returns an ordered ranking of rows based on the *value_expression* in the ORDER BY clause.

Type

ANSI SQL:2011 window function.

Syntax

where:

Syntax element ...	Specifies ...
OVER	how the values, grouped according to the PARTITION BY and RESET WHEN clauses and named by *value_expression* in the ORDER BY clause, are ranked.
PARTITION BY	in its *column_reference* the column, or columns, according to which ranking resets. PARTITION BY is optional. If there are no PARTITION BY or RESET WHEN clauses, then the entire result set, specified by the ORDER BY clause, constitutes a single group, or partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the column, or columns, being ranked.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
NULLS FIRST	that NULL results are to be listed first.

Syntax element …	Specifies …
NULLS LAST	that NULL results are to be listed last.
RESET WHEN	the group or partition over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set constitutes a single partition. If RESET WHEN is specified, then the ORDER BY clause must be specified also.
condition	a conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition. The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions. For more information, see "RESET WHEN Condition Rules" on page 956 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Using DENSE_RANK

The ranks are consecutive integers beginning with 1. Rows with equal values receive the same rank. Rank values are not skipped in the event of ties.

Result Type

The result data type is INTEGER.

Example

The following SELECT statement:

```
SELECT lname, serviceyrs,
 DENSE_RANK()OVER(ORDER BY serviceyrs)
 FROM schooltbl
 GROUP BY 1,2;
```

returns the ordered ranking by service years for teachers listed in *schooltbl*.

lname	serviceyrs	DENSE_RANK
Adams	10	1
Peters	10	1

lname	serviceyrs	DENSE_RANK
Murray	10	1
Rogers	15	2
Franklin	16	3
Smith	20	4
Ford	20	4
Derby	20	4
Baker	25	5

FIRST_VALUE / LAST_VALUE

Purpose

Returns the first value or last value in an ordered set of values.

Type

ANSI SQL:2011 window function.

Syntax

1145-026A

where:

Syntax element ...	Specifies ...
value_expression	a column expression. FIRST_VALUE and LAST_VALUE use the default data type of *value_expression*. Larger numeric values are supported by casting them to a higher data type. The expression cannot contain any ordered analytical or aggregate functions.
IGNORE NULLS	optional keyword that specifies not to return NULL values. • IGNORE NULLS (with FIRST_VALUE) = returns the first non-null value in the set, or NULL if all values are NULL. • If IGNORE NULLS (with LAST_VALUE) = returns the last non-null value in the set, or NULL if all values are NULL. If all values are null, NULL is returned.
RESPECT NULLS	optional keyword that specifies whether to return NULL values • RESPECT NULLS (with FIRST_VALUE) = returns the first value, whether or not it is null. • RESPECT NULLS (with FIRST_VALUE) = returns the last value, whether or not it is null. If all values are null, NULL is returned.

Syntax element ...	Specifies ...
window	a group, cumulative, or moving computation. For Window Aggregate Function syntax, see "Window Aggregate Functions" on page 960. In presence of ties in the sort key of the Window Aggregate Function syntax, FIRST_VALUE and LAST_VALUE are non-deterministic. They return *value_expression* from any one of the rows with tied order by values. Note: If the ROWS phrase is omitted and there is an ORDER BY phrase, the default ROWS is UNBOUNDED PRECEDING AND CURRENT ROW. If the ROWS phrase is omitted and there is no ORDER BY phrase, the default ROWS is UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. If you want a default of ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING for LAST_VALUE (all rows in the partition), you need to set DIAGNOSTIC NONANSIFORLASTVALUE ON FOR SESSION.

ANSI Compliance

FIRST_VALUE and LAST_VALUE are Teradata extensions to the ANSI SQL:2011 standard.

Usage Notes

FIRST_VALUE and LAST_VALUE are especially valuable because they are often used as the baselines in calculations. For instance, with a partition holding sales data ordered by day, you may want to know how much the sales for each day were compared to the first sales day (FIRST_VALUE) for the period, or you may want to know, for a set of rows in increasing sales order, what the percentage size of each sale in the region was compared to the largest sale (LAST_VALUE) in the region.

IGNORE NULLS is particularly useful in populating an inventory table properly.

Selecting neither IGNORE NULLS or RESPECT NULLS is equivalent to selecting RESPECT NULLS.

Example

The following example returns by start date the salary, moving average (ma), and first and last salary in the moving average group.

Note: The functions are going to return the first/last value in the window. In the example, the first and last rows fall within the window. If the window were between 3 preceding and 2 preceding rows, you would see NULL for first value in the 1st two rows.

```
SELECT start_date, salary,
       AVG(salary) OVER(ORDER BY start_date
       ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING) ma,
       FIRST_VALUE(salary) OVER(ORDER BY start_date
       ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING) first,
       LAST_VALUE(salary) OVER(ORDER BY start_date
       ROWS BETWEEN 3 PRECEDING AND 1 FOLLOWING) last
FROM employee
```

```
ORDER BY start_date;
```

start_date	salary	ma	first	last
21-MAR-76	6661.78	6603.280	6661.78	6544.78
12-DEC-78	6544.78	5183.780	6661.78	2344.78
24-OCT-82	2344.78	4471.530	6661.78	2344.78
15-JAN-84	2334.78	4441.780	6661.78	4322.78
30-JUL-87	4322.980	4688.980	6544.78	7897.78
31-DEC-90	7897.78	3626.936	2344.78	1234.56
25-JUL-96	1234.56	3404.536	2334.78	1232.78
17-SEP-96	1232.78	3671.975	4322.78	1232.78

MAVG

Purpose

Computes the moving average of a value expression for each row in a partition using the specified value expression for the current row and the preceding *width*-1 rows.

Type

Teradata-specific function.

Syntax

```
— MAVG — ( — value_expression, — width, ┌──────────────────────────────┐
                                        └─ sort_expression ──┬──────┬─┘ ) —
                                                             ├─ ASC ─┤
                                                             └─ DESC ─┘
```

1101A399

where:

Syntax element ...	Specifies ...
value_expression	a numeric literal or column expression for which a moving average is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
width	number of previous rows to be used in computing the moving average. The value is always a positive integer literal. The maximum is 4096.
sort_expression	a literal or column expression or comma-separated list of literal or column expressions to be used to sort the values. For example, MAVG(Sale, 6, Region ASC, Store DESC), where Sale is the *value_expression*, 6 is the *width*, and Region ASC, Store DESC is the *sort_expression* list. The expression cannot contain any ordered analytical or aggregate functions.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Using AVG Instead of MAVG

The use of MAVG is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is equivalent to the ANSI-compliant AVG window function that specifies ROWS *value* PRECEDING as its aggregation group. MAVG is retained only for backward compatibility with existing applications.

For more information on the AVG window function, see "Window Aggregate Functions" on page 960.

Result Type and Attributes

The data type, format, and title for MAVG are as follows:

Data Type: Same as operand x

- If operand x is character, the format is the default format for FLOAT.
- If operand x is numeric, date, or interval, the format is the same format as x.

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Problems With Missing Data

Ensure that data you analyze using MAVG has no missing data points. Computing a moving average over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Computing the Moving Average When Number of Rows < *width*

For the (possibly grouped) resulting relation, the moving average considering *width* rows is computed where the rows are sorted by the *sort_expression* list.

When there are fewer than *width* rows, the average is computed using the current row and all preceding rows.

Example 1

Compute the 7-day moving average of sales for product code 10 for each day in the month of October, 1996.

```
SELECT cdate, itemID, MAVG(sumPrice, 7, date)
FROM (SELECT a1.calendar_date, a1.itemID,
SUM(a1.price)
FROM Sales a1
WHERE a1.itemID=10 AND a1.calendar_date
BETWEEN 96-10-01 AND 96-10-31
GROUP BY a1.calendar_date, a1.itemID) AS T1(cdate,
itemID, sumPrice);
```

Example 2

The following example calculates the 50-day moving average of the closing price of the stock for Zemlinsky Bros. Corporation. The ticker name for the company is ZBC.

```
SELECT MarketDay, ClosingPrice,
   MAVG(ClosingPrice,50, MarketDay) AS ZBCAverage
FROM MarketDailyClosing
WHERE Ticker = 'ZBC'
ORDER BY MarketDay;
```

The results for the query might look something like the following table:

MarketDay	ClosingPrice	ZBCAverage
12/27/1999	89 1/16	85 1/2
12/28/1999	91 1/8	86 1/16
12/29/1999	92 3/4	86 1/2
12/30/1999	94 1/2	87

MDIFF

Purpose

Returns the moving difference between the specified value expression for the current row and the preceding *width* rows for each row in the partition.

Type

Teradata-specific function.

Syntax

```
— MDIFF — ( — value_expression, — width, ┌──────────────,──────────────┐ ) —
                                         │ sort_expression ┌─ASC─┐     │
                                         │                 └─DESC─┘     │
                                         └──────────────────────────────┘
```

1101A400

where:

Syntax element ...	Specifies ...
value_expression	a numeric column or literal expression for which a moving difference is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
width	the number of previous rows to be used in computing the moving difference. The value is always a positive integer literal. The maximum is 4096.
sort_expression	a literal or column expression or comma-separated list of literal or column expressions to be used to sort the values. For example, MDIFF(Sale, 6, Region ASC, Store DESC), where Sale is the *value_expression,* 6 is the *width*, and Region ASC, Store DESC is the *sort_expression* list. The expression cannot contain any ordered analytical or aggregate functions.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Meaning of Moving Difference

A common business metric is to compare activity for some variable in a current time period to the activity for the same variable in another time period a fixed distance in the past. For example, you might want to compare current sales volume against sales volume for preceding quarters. This is a moving difference calculation where *value_expression* would be the quarterly sales volume, width is 4, and *sort_expression* might be the quarter_of_calendar column from the SYS_CALENDAR.Calendar system view.

Using SUM Instead of MDIFF

The use of MDIFF is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is retained only for backward compatibility with existing applications. MDIFF(x, w, y) is equivalent to:

```
x - SUM(x) OVER (ORDER BY y
                 ROWS BETWEEN w PRECEDING AND w PRECEDING)
```

For more information on the SUM window function, see "Window Aggregate Functions" on page 960.

Result Type and Attributes

The data type, format, and title for MDIFF are as follows:

- If operand x is character, the data type is the same as x and the format is the default format for FLOAT.
- If operand x is numeric, the data type is the same as x and the format is the same format as x.
- If operand is date, the data type is INTEGER and the format is the default format for INTEGER.

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Problems With Missing Data

Ensure that rows you analyze using MDIFF have no missing data points. Computing a moving difference over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Computing the Moving Difference When No Preceding Row Exists

When the number of preceding rows to use in a moving difference computation is fewer than the specified width, the result is null.

Example 1

Display the difference between each quarter and the same quarter sales for last year for product code 10.

```
SELECT year_of_calendar, quarter_of_calendar,
MDIFF(sumPrice, 4, year_of_calendar, quarter_of_calendar)
FROM (SELECT a2.year_of_calendar,
a2.quarter_of_calendar, SUM(a2.Price) AS sumPrice
FROM Sales a1, SYS_CALENDAR.Calendar a2
WHERE a1.itemID=10 and a1.calendar_date=a2.calendar_date
GROUP BY a2.year_of_calendar, a2.quarter_of_calendar) AS T1
ORDER BY year_of_calendar, quarter_of_year;
```

Example 2

The following example computes the changing market volume week over week for the stock of company Horatio Parker Imports. The ticker name for the company is HPI.

```
SELECT MarketWeek, WeekVolume,
   MDIFF(WeekVolume,1,MarketWeek) AS HPIVolumeDiff
FROM
(SELECT MarketWeek, SUM(Volume) AS WeekVolume
FROM MarketDailyClosing
WHERE Ticker = 'HPI'
GROUP BY MarketWeek)
ORDER BY MarketWeek;
```

The result might look like the following table. Note that the first row is null for column HPIVolume Diff, indicating no previous row from which to compute a difference.

MarketWeek	WeekVolume	HPIVolumeDiff
11/29/1999	9817671	?
12/06/1999	9945671	128000
12/13/1999	10099459	153788
12/20/1999	10490732	391273
12/27/1999	11045331	554599

MEDIAN

Purpose

Takes a numeric or datetime value and returns the middle value or an interpolated value that would be the middle value after the values are sorted. Nulls are ignored in the calculation.

Type

MEDIAN is an aggregate function.

Syntax

```
MEDIAN ———————— (value_expression) ————————————————————————————
```

1145-030A

where

Syntax element ...	Specifies ...
value_expression	a single expression that must be a Numeric data type.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Result

The function returns the same data type as the data type of the argument.

Example

MEDIAN, an inverse distribution function that assumes a continuous distribution model, is a specific case of PERCENTILE_CONT where the percentile value is 0.5.

```
MEDIAN (value_expression)
```

is same as:

```
PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY value_expression)
```

See “PERCENTILE_CONT / PERCENTILE_DISC” on page 1012.

MLINREG

Purpose

Returns a predicted value for an expression based on a least squares moving linear regression of the previous *width*-1 (based on *sort_expression*) column values.

Type

Teradata-specific function.

Syntax

```
— MLINREG — ( — value_expression, — width, — sort_expression ┬──────┬ ) —
                                                            ├ ASC  ┤
                                                            └ DESC ┘
```

1101A401

where:

Syntax element ...	Specifies ...
value_expression	a numeric literal or column expression for which a predicted value is to be computed. The expression cannot contain any ordered analytical or aggregate functions. The data type of the expression must be numeric or a data type that Teradata Database can successfully convert implicitly to numeric.
width	the number of rows to use to compute the function. *width*-1 previous rows are used to compute the linear regression and the row value itself is used for calculating the predicted value. The value is always a positive integer literal greater than 2. The maximum is 4096.
sort_expression	a column expression that defines the independent variable for calculating the linear regression. For example, MLINREG(Sales, 6, Fiscal_Year_Month ASC), where Sales is the *value_expression,* 6 is the *width*, and Fiscal_Year_Month ASC is the *sort_expression*. The data type of the column reference must be numeric or a data type that Teradata Database can successfully convert implicitly to numeric.

Syntax element ...	Specifies ...
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 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:2011 standard, and is retained only for backward compatibility with existing applications.

Result Type and Attributes

The data type, format, and title for MLINREG are as follows:

Data Type: Same as operand x

- If operand x is character, the format is the default format for FLOAT.
- If operand x is numeric, date, or interval, the format is the same format as x.

For information on the default format of data types and an explanation of the formatting characters in the format, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Default Independent Variable

MLINREG assumes that the independent variable is described by *sort_expression*.

Computing MLINREG When Preceding Rows < *width* - 1

When there are fewer than *width*-1 preceding rows, MLINREG computes the regression using all the preceding rows.

MLINREG Report Structure

All rows in the results table except the first two, which are always null, display the predicted value.

Example

Consider the *itemID*, *smonth*, and *sales* columns from *sales_table*:

```
SELECT itemID, smonth, sales
FROM fiscal_year_sales_table
ORDER BY itemID, smonth;

itemID  smonth    sales
------  --------  -----
A              1    100
A              2    110
A              3    120
A              4    130
A              5    140
A              6    150
A              7    170
A              8    190
A              9    210
A             10    230
A             11    250
A             12  ?
B              1     20
B              2     30
...
```

Assume that the NULL in the *sales* column is because in this example the month of December (month 12) is a future date and the value is unknown.

The following statement uses MLINREG to display the expected sales using past trends for each month for each product using the sales data for the previous six months.

```
SELECT itemID, smonth, sales, MLINREG(sales,7,smonth)
FROM fiscal_year_sales_table;
GROUP BY itemID;

itemID  smonth    sales  MLinReg(sales,7,smonth)
------  --------  -----  -----------------------
A              1    100  ?
A              2    110  ?
A              3    120          120
A              4    130          130
A              5    140          140
A              6    150          150
A              7    170          160
A              8    190          177
A              9    210          198
A             10    230          222
A             11    250          247
A             12  ?              270
B              1     20  ?
B              2     30  ?
...
```

MSUM

Purpose

Computes the moving sum specified by a value expression for the current row and the preceding *n*-1 rows. This function is very similar to the MAVG function.

Type

Teradata-specific function.

Syntax

```
— MSUM — ( — value_expression, — width, ┌──────────────── , ────────────────┐ sort_expression ┬──────────┬ ) —
                                        │                                   │                 ├─ ASC ────┤
                                        │                                   │                 └─ DESC ───┘
```

1101A402

where:

Syntax element ...	Specifies ...
value_expression	a numeric literal or column expression for which a moving sum is to be computed. The expression cannot contain any ordered analytical or aggregate functions.
width	the number of previous rows to be used in computing the moving sum. The value is always a positive integer literal. The maximum is 4096.
sort_expression	a literal or column expression or comma-separated list of literal or column expressions to be used to sort the values. For example, MSUM(Sale, 6, Region ASC, Store DESC), where Sale is the *value_expression,* 6 is the *width*, and Region ASC, Store DESC is the *sort_expression* list.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Using SUM Instead of MSUM

The use of MSUM is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard, and is equivalent to the ANSI-compliant SUM window function. MSUM is retained only for backward compatibility with existing applications.

For more information on the SUM window function, see "Window Aggregate Functions" on page 960.

Result Type and Attributes

The data type, format, and title for MSUM are as follows:

Data Type: Same as operand x

- If operand x is character, the format is the default format for FLOAT.
- If operand x is numeric, the format is the same format as x.

For information on the default format of data types, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Problems With Missing Data

Ensure that data you analyze using MSUM has no missing data points. Computing a moving average over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Computing MSUM When Number of Rows < *width*

For data having fewer than *width* rows, MSUM computes the sum using all the preceding rows. MSUM returns the current sum rather than nulls when the number of rows in the sample is fewer than *width*.

Possible Result Overflow with SELECT Sum

When using this function, the result can create an overflow when the data type and format are not in sync. For a column defined as:

```
Salary Decimal(15,2) Format '$ZZZ,ZZ9.99'
```

The following query:

```
SELECT SUM (Salary) FROM Employee;
```

causes an overflow because the decimal operand and the format are not in sync.

To avoid possible overflows, explicitly specify the format for decimal sum to specify a format large enough to accommodate the decimal sum resultant data type.

```
SELECT Sum(Salary) (format '$Z,ZZZ,ZZZ,ZZ9.99) FROM Employee;
```

PERCENT_RANK

Purpose

Returns the relative rank of rows for a *value_expression*.

Type

ANSI SQL:2011 window function.

Syntax

where:

Syntax element ...	Specifies ...
OVER	how the values, grouped according to the PARTITION BY and RESET WHEN clauses and named by *value_expression* in the ORDER BY clause, are ranked.
PARTITION BY	in its *column_reference* the column, or columns, according to which ranking resets. PARTITION BY is optional. If there are no PARTITION BY or RESET WHEN clauses, then the entire result set, specified by the ORDER BY clause, constitutes a single group or partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the column, or columns, being ranked.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

Syntax element ...	Specifies ...
NULLS FIRST	that NULL results are to be listed first.
NULLS LAST	that NULL results are to be listed last.
RESET WHEN	the group or partition over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set constitutes a single partition. If RESET WHEN is specified, then the ORDER BY clause must be specified also.
condition	a conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition. The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions. For more information, see "RESET WHEN Condition Rules" on page 956and the "QUALIFY Clause" in *SQL Data Manipulation Language*.

ANSI Compliance

This is ANSI SQL:2011 compliant.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

Computation

The formula for PERCENT_RANK is:

$$\text{PERCENT_RANK} = \frac{(\text{RK} - 1)}{(\text{NR} - 1)}$$

where:

This variable ...	Represents the ...
RK	rank of the row
NR	number of rows in the window partition

The assigned rank of a row is defined as 1 (one) plus the number of rows that precede the row and are not peers of it.

PERCENT_RANK is expressed as an approximate numeric ratio between 0.0 and 1.0.

PERCENT_RANK has this value ...	FOR the result row assigned this rank ...
0.0	1.
1.0	highest in the result.

Result Type and Attributes

For PERCENT_RANK() OVER (PARTITION BY *x* ORDER BY *y direction*), the data type, format, and title are as follows:

Data Type	Format	Title
REAL	the default format for DECIMAL(7,6).	Percent_Rank(y direction)

For an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example 1

Determine the relative rank, called the percent_rank, of Christmas sales.

The following query:

```
SELECT sales_amt,
PERCENT_RANK() OVER (ORDER BY sales_amt)
FROM xsales;
```

might return the following results. Note that the relative rank is returned in ascending order, the default when no sort order is specified and that the currency is not reported explicitly.

sales_amt	Percent_Rank
100.00	0.000000
120.00	0.125000
130.00	0.250000
140.00	0.375000
143.00	0.500000
147.00	0.625000
150.00	0.750000
155.00	0.875000
160.00	1.000000

Example 2

Determine the rank and the relative rank of Christmas sales.

```
SELECT sales_amt,
RANK() OVER (ORDER BY sales_amt),
PERCENT_RANK () OVER (ORDER BY sales_amt)
FROM xsales;
```

sales_amt	Rank	Percent_Rank
100.00	1	0.000000
120.00	2	0.125000
130.00	3	0.250000
140.00	4	0.375000
143.00	5	0.500000
147.00	6	0.625000
150.00	7	0.750000
155.00	8	0.875000
160.00	9	1.000000

Example 3

The following SQL statement illustrates the difference between PERCENT_RANK and cumulative distribution.

```
SELECT sales_amt,
 PERCENT_RANK() OVER (ORDER BY sales_amt),
 CUME_DIST() OVER (ORDER BY sales_amt)
 FROM xsales;
```

sales_amt	PERCENT_Rank	CUME_DIST
100.	.000000	0.125000
120.	.142857	0.250000
130	.285714	.375000
140.	.428571	.500000
147.	.571429	.625000
150.	.714286	.750000
155.	.857143	.875000
160.	1.000000	1.000000

PERCENTILE_CONT / PERCENTILE_DISC

Purpose

Returns an interpolated value that falls within its *value_expression* with respect to its sort specification.

Type

PERCENTILE_CONT and PERCENTILE_DISC are aggregate function.

Syntax

where:

Syntax element ...	Specifies ...
value_expression_1	a numeric value between 0 and 1 inclusive.
WITHIN GROUP	the group over which the function operates.
ORDER BY	the order in which the values in a group or partition are sorted.
value_expression_2	a single expression that must be a numeric value.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
NULLS FIRST	that NULL results are to be listed first.
NULLS LAST	that NULL results are to be listed last.

ANSI Compliance

PERCENTILE_CONT and PERCENTILE_DISC are Teradata extensions to the ANSI SQL:2011 standard.

Result

The function returns the same data type as the data type of the argument.

Nulls are ignored in the calculation.

Usage Notes

Both functions are inverse distribution functions that assume a continuous distribution.

- PERCENTILE_CONT returns a computed result after doing linear interpolation.
- PERCENTILE_DISC simply returns a value from the set of values.

Example

Using this table:

Area	Address	Price
Downtown	72 Easy Street	509000
Downtown	29 Right Way	402000
Downtown	45 Diamond Lane	203000
Downtown	76 Blind Alley	201000
Downtown	15 Tern Pike	199000
Downtown	444 Kanga Road	102000
Uptown	15 Peak Street	456000
Uptown	27 Primrose Path	349000
Uptown	44 Shady Lane	341000
Uptown	34 Design Road	244000
Uptown	2331 Highway 64	244000
Uptown	77 Sunset Strip	102000

the following SQL statement returns a computed result after doing linear interpolation, as shown in the table immediately below.

```
SELECT area,
       AVG(price),
       PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY price),
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY price)
FROM market
```

```
GROUP BY area;
```

Area	Average Price	PERC_DISC	PERC_CONT
Downtown	269333	201000	202000
Uptown	289333	244000	292500

QUANTILE

Purpose

Computes the quantile scores for the values in a group.

Type

Teradata-specific function.

Syntax

where:

Syntax element ...	Specifies ...
quantile_literal	a positive integer literal used to define the number of quantile partitions to be used.
sort_expression	a literal or column expression or comma-separated list of literal or column expressions to be used to sort the values. For example, QUANTILE(10, Region ASC, Store DESC), where 10 is the *quantile_literal* and Region ASC, Store DESC is the *sort_expression* list.
ASC	ascending sort order.
DESC	descending sort order. The default sort direction is DESC.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Definition

A quantile is a generic interval of user-defined width. For example, percentiles divide data among 100 evenly spaced intervals, deciles among 10 evenly spaced intervals, quartiles among 4, and so on. A quantile score indicates the fraction of rows having a *sort_expression* value lower than the current value. For example, a percentile score of 98 means that 98 percent of the rows in the list have a *sort_expression* value lower than the current value.

Using ANSI Window Functions Instead of QUANTILE

The use of QUANTILE is strongly discouraged. It is a Teradata extension to the ANSI SQL:2011 standard and is retained only for backward compatibility with existing applications.

To compute QUANTILE(q, s) using ANSI window functions, use the following:

```
(RANK() OVER (ORDER BY s) - 1) * q / COUNT(*) OVER()
```

QUANTILE Report

For each row in the group, QUANTILE returns an integer value that represents the quantile of the *sort_expression* value for that row relative to the *sort_expression* value for all the rows in the group.

Quantile Value Range

Quantile values range from 0 through (Q-1), where Q is the number of quantile partitions specified by *quantile_literal*.

Result Type and Attributes

The data type, format, and title for QUANTILE(Q, list) are as follows:

Data Type	Format	Title
INTEGER	the default format for the INTEGER data type	Quantile(Q, list)

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Example 1

Display each item and its total sales in the ninth (top) decile according to the total sales.

```
SELECT itemID, sumPrice
FROM (SELECT a1.itemID, SUM(price)
FROM Sales a1
GROUP BY a1.itemID) AS T1(itemID, sumPrice)
QUALIFY QUANTILE(10,sumPrice)=9;
```

Example 2

The following example groups all items into deciles by profitability.

```
SELECT Item, Profit, QUANTILE(10, Profit) AS Decile
FROM
   (SELECT Item, Sum(Sales) – (Count(Sales) * ItemCost) AS Profit
   FROM DailySales, Items
   WHERE DailySales.Item = Items.Item
   GROUP BY Item) AS Item;
```

The result might look like the following table:

Item	Profit	Decile
High Tops	97112	9
Low Tops	74699	7
Running	69712	6
Casual	28912	3
Xtrain	100129	9

Example 3

Because QUANTILE uses equal-width histograms to partition the specified data, it does not partition the data equally using equal-height histograms. In other words, do not expect equal row counts per specified quantile. Expect empty quantile histograms when, for example, duplicate values for *sort_expression* are found in the data.

For example, consider the following simple SELECT statement.

```
SELECT itemNo, quantity, QUANTILE(10,quantity) FROM inventory;
```

The report might look like this.

itemNo	quantity	Quantile(10, quantity)
13	1	0
9	1	0
7	1	0
2	1	0
5	1	0
3	1	0
1	1	0
6	1	0
4	1	0
10	1	0
8	1	0
11	1	0
12	9	9

Because the quantile sort is on quantity, and there are only two quantity scores in the inventory table, there are no scores in the report for deciles 1 through 8.

RANK (ANSI)

Purpose

Returns an ordered ranking of rows based on the *value_expression* in the ORDER BY clause.

Type

ANSI SQL:2011 window function.

Syntax

where:

Syntax element ...	Specifies ...
OVER	how the values, grouped according to the PARTITION BY and RESET WHEN clauses and named by *value_expression* in the ORDER BY clause, are ranked.
PARTITION BY	in its *column_reference* the column, or columns, according to which ranking resets. PARTITION BY is optional. If there are no PARTITION BY or RESET WHEN clauses, then the entire result set, specified by the ORDER BY clause, constitutes a single group, or partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the column, or columns, being ranked.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.

Syntax element ...	Specifies ...
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
NULLS FIRST	that NULL results are to be listed first.
NULLS LAST	that NULL results are to be listed last.
RESET WHEN	the group or partition over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set constitutes a single partition. If RESET WHEN is specified, then the ORDER BY clause must be specified also.
condition	a conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition. The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions. For more information, see "RESET WHEN Condition Rules" on page 956 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.
TIES LOW	that all ties get the lowest rank. Returns an Integer data type.
TIES HIGH	that all ties get the highest rank. Returns an Integer data type.
TIES AVG	that all ties get the average rank. Returns a Decimal data type.
TIES DENSE	that all ties are ranked as DENSE_RANK ranks them. Returns an Integer data type.

ANSI Compliance

This is ANSI SQL:2011 compliant.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

Meaning of Rank

RANK returns an ordered ranking of rows based on the *value_expression* in the ORDER BY clause. All rows having the same *value_expression* value are assigned the same rank.

If *n* rows have the same *value_expression* values, then they are assigned the same rank, call it rank *r*. The next distinct value receives rank *r*+*n*. And so on.

Less formally, RANK sorts a result set and identifies the numeric rank of each row in the result. RANK returns an integer that represents the rank of each row in the result.

Result Type and Attributes

For RANK() OVER (PARTITION BY *x* ORDER BY *y direction*), the data type, format, and title are as follows:

Data Type	Format	Title
INTEGER	the default format for the INTEGER data type	Rank(y direction)

For an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Example 1

This example ranks salespersons by sales region based on their sales.

```
SELECT sales_person, sales_region, sales_amount,
   RANK() OVER (PARTITION BY sales_region ORDER BY sales_amount DESC)
FROM sales_table;
```

sales_person	sales_region	sales_amount	Rank(sales_amount)
Garabaldi	East	100	1
Baker	East	99	2
Fine	East	89	3
Adams	East	75	4
Edwards	West	100	1
Connors	West	99	2
Davis	West	99	2

The rank column in the preceding table lists salespersons in declining sales order according to the column specified in the PARTITION BY clause (sales_region) and that the rank of their sales (sales_amount) is reset when the sales_region changes.

Example 2

The following SQL statement illustrates the difference between RANK(ANSI) and DENSE_RANK(ANSI), returning the RANK and DENSE_RANK for sales_person by sales_region and sales_amount.:

```
SELECT sales_person, sales_region, sales_amount,
 RANK() OVER
    (PARTITION BY sales_region ORDER BY sales_amount DESC) as "Rank",
```

```
DENSE_RANK() OVER
   (PARTITION BY sales_region ORDER BY sales_amount DESC) as "DenseRank"
FROM sales_table;
```

sales_person	sales_region	sales_amount	Rank	DenseRank
Garabaldi	East	100	1	1
Baker	East	100	1	1
Fine	East	89	3	2
Adams	East	75	4	3
Edwards	West	100	1	1
Connors	West	99	2	2
Davis	West	99	2	2
Russell	West	50	4	3

RANK (Teradata)

Purpose

Returns the rank (1 … *n*) of all the rows in the group by the value of *sort_expression* list, with the same *sort_expression* values receiving the same rank.

Type

Teradata-specific function.

Syntax

where:

Syntax element …	Specifies …
sort_expression	a literal or column expression or comma-separated list of literal or column expressions to be used to sort the values. USE ELEWHERE: For example, RANK(Region ASC, Store DESC), where Region ASC, Store DESC is the *sort_expression* list. The expression cannot contain any ordered analytical or aggregate functions.
ASC	ascending sort order.
DESC	descending sort order. The default sort direction is DESC.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 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:2011 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 (ANSI)" on page 1018.

Meaning of Rank

A rank *r* implies the existence of exactly *r*-1 rows with *sort_expression* value preceding it. All rows having the same *sort_expression* value are assigned the same rank.

For example, if *n* rows have the same *sort_expression* values, then they are assigned the same rank, call it rank *r*. The next distinct value receives rank *r*+*n*.

Less formally, RANK sorts a result set and identifies the numeric rank of each row in the result. The only argument for RANK is the sort column or columns, and the function returns an integer that represents the rank of each row in the result.

Computing Top and Bottom Values

You can use RANK to compute top and bottom values as shown in the following examples.

Top(*n*, column) is computed as QUALIFY RANK(column DESC) <=*n*.

Bottom(*n*, column) is computed as QUALIFY RANK(column ASC) <=*n*.

Result Type and Attributes

The data type, format, and title for RANK(x) are as follows:

Data Type	Format	Title
INTEGER	the default format for the INTEGER data type	Rank(x)

For information on the default format of data types, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

Example 1

Display each item, its total sales, and its sales rank for the top 100 selling items.

```
SELECT itemID, sumPrice, RANK(sumPrice)
FROM
   (SELECT a1.itemID, SUM(a1.Price)
   FROM Sales a1
   GROUP BY a1.itemID AS T1(itemID, sumPrice)
   QUALIFY RANK(sumPrice) <=100;
```

Example 2

Sort employees alphabetically and identify their level of seniority in the company.

```
SELECT EmployeeName, (HireDate - CURRENT_DATE) AS ServiceDays,
RANK(ServiceDays) AS Seniority
FROM Employee
ORDER BY EmployeeName;
```

The result might look like the following table:

EmployeeName	Service Days	Seniority
Ferneyhough	9931	2
Lucier	9409	4
Revueltas	9408	5
Ung	9931	2
Wagner	10248	1

Example 3

Sort items by category and report them in order of descending revenue rank.

```
SELECT Category, Item, Revenue, RANK(Revenue) AS ItemRank
FROM ItemCategory,
   (SELECT Item, SUM(sales) AS Revenue
   FROM DailySales
   GROUP BY Item) AS ItemSales
WHERE ItemCategory.Item = ItemSales.Item
ORDER BY Category, ItemRank DESC;
```

The result might look like the following table.

Category	Item	Revenue	ItemRank
Hot Cereal	Regular Oatmeal	39112.00	4
Hot Cereal	Instant Oatmeal	44918.00	3
Hot Cereal	Regular COW	59813.00	2
Hot Cereal	Instant COW	75411.00	1

ROW_NUMBER

Purpose

Returns the sequential row number, where the first row is number one, of the row within its window partition according to the window ordering of the window.

Type

ANSI SQL:2011 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 are no PARTITION BY or RESET WHEN clauses, then the entire result set, specified by the ORDER BY clause, constitutes a partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the order in which to sort the values in the partition.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
NULLS FIRST	that NULL results are to be listed first.

Syntax element ...	Specifies ...
NULLS LAST	that NULL results are to be listed last.
RESET WHEN	the group or partition over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set constitutes a single partition. If RESET WHEN is specified, then the ORDER BY clause must be specified also.
condition	a conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition. The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions. For more information, see "RESET WHEN Condition Rules" on page 956 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.

ANSI Compliance

This is ANSI SQL:2011 compliant.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

Window Aggregate Equivalent

```
ROW_NUMBER() OVER (PARTITION BY column ORDER BY value)
```

is equivalent to

```
COUNT(*) OVER (PARTITION BY column ORDER BY value
ROWS UNBOUNDED PRECEDING).
```

For more information on COUNT, see "Window Aggregate Functions" on page 960.

Example

To order salespersons based on sales within a sales region, the following SQL query might yield the following results.

```
SELECT ROW_NUMBER() OVER (PARTITION BY sales_region
                          ORDER BY sales_amount DESC),
sales_person, sales_region, sales_amount
FROM sales_table;

Row_Number()  sales_person  sales_region  sales_amount
------------  ------------  ------------  ------------
           1  Baker         East                   100
           2  Edwards       East                    99
           3  Davis         East                    89
           4  Adams         East                    75
```

```
1  Garabaldi     West                    100
2  Connors       West                     99
3  Fine          West                     99
```

CHAPTER 22 Period Functions and Operators

This chapter describes the Period functions and operators.

Period Value Constructor

Purpose

Initializes an instance of a Period data type.

Syntax

```
┬─ PERIOD (datetime_expression) ──────────────────────────────────────────┬
├─ PERIOD (datetime_expression, datetime_expression) ─────────────────────┤
├─ PERIOD (datetime_expression, UNTIL_CHANGED) ───────────────────────────┤
└─ PERIOD (datetime_expression, UNTIL_CLOSED) ────────────────────────────┘
```

1182A015

where:

Syntax element...	Specifies...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
UNTIL_CHANGED	a DATE or TIMESTAMP value that is considered to be forever or until it is changed. For PERIOD(DATE) types, UNTIL_CHANGED has a value of DATE '9999-12-31'. For PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) types, UNTIL_CHANGED has a value of TIMESTAMP '9999-12-31 23:59:59.999999 00:00'(with the precision truncated to the precision of the beginning bound and the time zone omitted if the beginning bound does not have a time zone). UNTIL_CHANGED supports derived periods. See "IS UNTIL_CHANGED/IS NOT UNTIL_CHANGED" on page 1047.
UNTIL_CLOSED	an ending bound for the Period value of a temporal table transaction-time column that indicates that the row is an open row. UNTIL_CLOSED has a data type of TIMESTAMP(6) WITH TIME ZONE and a value of TIMESTAMP '9999-12-31 23:59:59.999999+00:00'. For more information about temporal tables, see *Temporal Table Support*. UNTIL_CLOSED supports derived periods. See "IS UNTIL_CLOSED/IS NOT UNTIL_CLOSED" on page 1049.

Result Value

The following rules apply to the result value:

- If the beginning or ending bound is NULL, or both the bounds are NULL, the result is NULL.
- If the beginning and ending bounds are NULL or if the beginning bound is NULL and the ending bound is UNTIL_CHANGED, then the type of the period defaults to PERIOD(TIMESTAMP(0)).
- If only the beginning bound is specified, the result ending bound is the beginning bound plus one granule of the result element type. If the result ending bound exceeds or becomes equal to the maximum allowed DATE or TIMESTAMP value for result data type of PERIOD(DATE) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]), respectively, an error is reported.
- If an ending bound is specified as a value expression and the beginning bound and ending bound have different precisions, the result precision is the higher of the two precisions. Otherwise, the result precision is the precision of the beginning bound.
- UNTIL_CHANGED sets the result ending element to a maximum DATE or TIMESTAMP value depending on the data type of the beginning bound. If the data type of the beginning bound is TIMESTAMP(n) WITH TIME ZONE, the result ending element is set to the maximum TIMESTAMP(n) WITH TIME ZONE value at UTC (that is, the time zone displacement for the ending bound is INTERVAL '00:00' HOUR TO MINUTE).
- If the beginning bound or the ending bound or the beginning and ending bounds include a time zone value, and the ending bound is not UNTIL_CHANGED, the result data type is WITH TIME ZONE. If only one of the bounds includes a time zone value, the time zone field of the other is set to the current session time zone displacement. If both bounds include time zone values, the result bounds include the corresponding time zone value.
- The result Period data type has an element type that is the same as the DateTime data type of the beginning bound except with the precision and time zone as defined previously.
- The handling of leap seconds for Period data types with TIME and TIMESTAMP element types is as follows. If the value for the beginning or ending bound contains leap seconds, the seconds portion gets adjusted to 59.999999 with the precision truncated to the result precision. During this process, if the beginning and ending bounds are the same, an error is reported.

Usage Rules

The following rules apply to the Period value constructor:

- The beginning bound must have a DateTime data type and, if an ending bound is specified, the data types of the beginning and ending bounds must be comparable.
- The ending bound where the data type of the beginning bound is DATE or TIMESTAMP can be set to UNTIL_CHANGED.
- If the ending bound is set to UNTIL_CLOSED, the following must be true:
  - The data type of the beginning bound value must be comparable with TIMESTAMP(6) WITH TIME ZONE.

  - The constructor is only valid in an assignment operation where the target column to which the result is assigned is a transaction-time column.
  - Because the only way to set the value of a transaction-time column is by using nontemporal DML, the constructor is only valid in a nontemporal DML statement.
- Teradata Database reports an error if any of the following are true:
  - UNTIL_CHANGED is specified for the beginning bound.
  - The result beginning bound is greater than or equal to the result ending bound.
  - The data types of the beginning and ending bounds are not comparable.
  - UNTIL_CHANGED is specified for the ending bound and the data type of the beginning bound is TIME(n) [WITH TIME ZONE].
  - UNTIL_CLOSED is specified for the beginning bound.

Example

In the following example, assume t1 is a table with an INTEGER column c1 and a PERIOD(DATE) column c2 and t2 is a table with an INTEGER column a and two DATE columns b and c.

This example shows the Period value constructor used in two INSERT statements.

```
INSERT INTO t1
  VALUES (1, PERIOD(DATE '2005-02-03', DATE '2006-02-04'));
INSERT INTO t1 SELECT a, PERIOD(b, c) FROM t2;
```

Arithmetic Operators

Purpose

Adds or subtracts an Interval value to or from a Period value, or adds a Period value to an Interval value.

Syntax

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type.
interval_expression	an expression that evaluates to an INTERVAL data type. For information on INTERVAL data types, see *SQL Data Types and Literals*.

Usage Notes

Assuming that p is a Period expression of element type DATE or TIMESTAMP and v is an Interval expression:

- p + v and v + p are both equivalent to:

  ```
  PERIOD(BEGIN(p) + v, CASE WHEN END(p) IS UNTIL_CHANGED THEN END(p)
  ELSE (END(p) + v) END)
  ```
- p - v is equivalent to:

  ```
  PERIOD(BEGIN (p) - v, CASE WHEN END(p) IS UNTIL_CHANGED THEN END(p)
  ELSE (END(p) - v) END)
  ```

Assuming that p is a Period expression of element type TIME and v is an interval expression:

- p + v and v + p are both equivalent to:

  ```
  PERIOD(BEGIN(p) + v, END(p) + v)
  ```
- p - v is equivalent to:

  ```
  PERIOD(BEGIN (p) - v, END(p) - v)
  ```

Usage Rules

The following rules apply to arithmetic operators and Period data types:

- The interval expression must be a valid interval expression and must follow the rules of an Interval expression (see "ANSI Interval Expressions" on page 830). Otherwise, an error is reported. For example, the interval expression (DATE '2006-02-03' - DATE '2005-02-03') DAY, results in a value of 365 days which cannot fit into the default precision 2 of the interval qualifier DAY; therefore, an error is reported.
- The period arithmetic operations of adding or subtracting an Interval to or from a period or adding a period to an Interval follow the rules of DateTime expressions. Otherwise, errors are reported. See "ANSI Interval Expressions" on page 830 for details on DateTime expression rules.
- An interval expression can be subtracted from a Period expression but not vice versa. If a Period expression is subtracted from an interval expression, an error is reported.
- For a Period expression with an element type of TIME, if the Period arithmetic operation results in a beginning bound less than the ending bound, an error is reported.
- For a period of element type DATE or TIMESTAMP, if the ending bound is UNTIL_CHANGED, the ending bound in the result ending bound is UNTIL_CHANGED. If the ending bound is not UNTIL_CHANGED and the ending bound in the result evaluates to an UNTIL_CHANGED value, an error is reported.
- For a period arithmetic operation, one of the operands must be an INTERVAL data type. Otherwise, an error is reported.

Comparison of Period Types

Two Period values are comparable if their element types are of same DateTime data type. The DateTime data types are DATE, TIME and TIMESTAMP.

- The PERIOD(DATE) date type is comparable with the PERIOD(DATE) data type.
- The PERIOD(TIME(n)[WITH TIME ZONE]) data type is comparable with a PERIOD(TIME(m)[WITH TIME ZONE]) data type,
- The PERIOD(TIMESTAMP(n)[WITH TIME ZONE]) data type is comparable with a PERIOD(TIMESTAMP(m)[WITH TIME ZONE]) data type.

Teradata extends this to allow a CHARACTER and VARCHAR value to be implicitly cast as a Period data type for some operators and, therefore, have a Period data type. Since the Period data type is the data type of the other Period expression, these Period expressions will be comparable.

DateTime and Period data are saved internally with the maximum precision of 6, although the specified precision may be less than this and is padded with zeroes. Thus, the comparison operations with differing precisions work without any additional logic.

In addition, the internal value is saved in UTC for a Time or Timestamp value, or for a Period value with an element type of TIME or TIMESTAMP. All comparable Period expressions can be compared directly due to this internal representation irrespective of whether they contain a time zone value, or whether they have the same precision.

Note: The time zone values are ignored when comparing values.

All comparison operations involving UNTIL_CLOSED in a temporal table transaction-time column use the internal value of UNTIL_CLOSED (TIMESTAMP '9999-12-31 23:59:59:999999+00:00') to evaluate the result. For more information about temporal tables, see *Temporal Table Support*.

Comparison Operators

The following table describes the comparison operators.

Operator	Purpose
EQ or =	Assume p1 and p2 are Period expressions and have comparable Period data types. If BEGIN(p1) = BEGIN(p2) AND END(p1) = END(p2), the result of the comparison is TRUE; otherwise, the result is FALSE. If either Period expression is NULL, the result is UNKNOWN. If the Period expressions have different element types, one of them must be explicitly CAST as the other. If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR expression is implicitly converted, before comparison, to the data type of the Period expression based on the format of the Period expression. EQ supports comparisons between derived periods.
LT or <	Assume p1 and p2 are Period expressions and have comparable Period data types. If BEGIN(p1) < BEGIN(p2) OR (BEGIN(p1) = BEGIN(p2) AND END(p1) < END(p2)), the result of the comparison is TRUE; otherwise, the result is FALSE. If either Period expression is NULL, the result is UNKNOWN. If the Period expressions have different element types, one of them must be explicitly CAST as the other. If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR operand is implicitly converted, before comparison, to the data type of the Period expression based on the format of the Period expression. If the ending bound value of a temporal table transaction-time column is UNTIL_CLOSED, the ending bound value is only less than a TIMESTAMP column value or TIMESTAMP literal if the column value or literal is the maximum TIMESTAMP value with leap seconds. This can be possible only if the ending bound of the transaction-time column is used in a comparison with the timestamp value. For more information about temporal tables, see *Temporal Table Support*. LT supports comparisons between derived periods.

Operator	Purpose
GT or >	Assume p1 and p2 are Period expressions and have comparable Period data types. If BEGIN(p1) > BEGIN(p2) OR (BEGIN(p1) = BEGIN(p2) AND END(p1) > END(p2)), the result of the comparison is TRUE; otherwise, it is FALSE. If either Period expression is NULL, the result is UNKNOWN. If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR Period expression is implicitly converted, before comparison, to the data type of the Period expression based on the format of the Period expression. GT supports comparisons between derived periods.
NE or <> or NOT= or ^= or LE or <= or GE or >=	These comparison operators are supported for comparable Period expressions. If one Period expression has a Period data type and the other Period expression has CHARACTER or VARCHAR data type, the CHARACTER or VARCHAR Period expression is implicitly converted, before comparison, to the data type of the Period expression based on the format of the Period expression. NE, LE, and GE support comparisons between derived periods.

BEGIN

Purpose

Returns the beginning bound of a Period expression or a derived period.

Syntax

```
BEGIN ─┬─ ( period_expression ) ─┬──────────────
       └─ ( derived_period ) ────┘
                                   1145-036A
```

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type.
derived_period	any expression that evaluates to a derived period.

Return Value

The result data type of the BEGIN function is same as the element type of the Period expression or the data type of the begin column if the argument is a derived period column. If the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the element type of the Period expression or the format of the begin column if the argument is a derived period column.

Error Conditions

If the argument does not have a Period data type, an error is reported.

Example 1

BEGIN returns the beginning bound of a Period data type.

```
SELECT * FROM employee WHERE BEGIN(period1) = DATE '2004-06-19';
```

Assume the query is executed on the following *employee* table where period1 is a PERIOD(DATE) column:

```
ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-01-05')
```

```
Adams   Marketing     ('2004-06-19', '2005-02-09')
Mary    Development   ('2004-06-19', '2005-01-05')
Simon   Sales         ('2004-06-22', '2005-01-07')
```

The result is:

```
ename   dept          period1
-----   -----------   ----------------------------
Adams   Marketing     ('2004-06-19', '2005-02-09')
Mary    Development   ('2004-06-19', '2005-01-05')
```

Example 2

BEGIN returns the beginning bound (*jdbegin*) of a derived period called *jobduration* created using the following SQL statement:

```
CREATE TABLE employee(id INTEGER,
                      name VARCHAR(50),
                      jdbegin DATE NOT NULL FORMAT 'YYYY-MM-DD',
                      jdend   DATE NOT NULL FORMAT 'YYYY-MM-DD',
                      PERIOD FOR jobduration(jdbegin,jdend)
  )PRIMARY INDEX(id);
```

The following SQL statements:

```
INSERT INTO employee(1025,'John',DATE'2011-01-02',DATE'2012-05-02');
SELECT BEGIN (jobduration) FROM employee;
```

returns:

```
     jdbegin
  __________
  2011-01-02
```

CONTAINS

Purpose

Evaluates two Period expressions, or derived periods, or DateTime expressions to TRUE, FALSE, or UNKNOWN.

Syntax

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a period data type. Note: The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
derived_period	any expression that evaluates to a derived period.

Error Conditions

If either expression evaluates to a data type that is other than a Period or DateTime, an error is reported.

If the expressions do not have comparable data types, an error is reported.

Result Value

- If both expressions have a Period data type or a derived period, the function returns TRUE if the beginning bound of the first expression is less than or equal to the beginning bound

of the second expression and the ending bound of the first expression is greater than or equal to the ending bound of the second expression; otherwise, the function returns FALSE.

- If the first expression is a Period expression or a derived period and the second expression is a DateTime expression, the function returns TRUE if the beginning bound of the Period expression or derived period is less than or equal to the DateTime expression and the ending bound of the Period expression or derived period is greater than the DateTime expression; otherwise, the function returns FALSE.
- If the first expression is a DateTime expression and the second expression is a Period expression, the function returns TRUE if the DateTime expression is less than or equal to beginning bound of the Period expression and the DateTime expression plus one granule is greater than or equal to the ending bound of the Period expression; otherwise, the function returns FALSE.
- If either expression is NULL, the operator returns UNKNOWN.

Example 1

Assume the following query is executed on the *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period2 CONTAINS period1;
```

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')

Example 2

Assume that in *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
   eid INTEGER NOT NULL,
   name VARCHAR(100) NOT NULL,
   deptno INTEGER NOT NULL,
```

```
jobst1 DATE NOT NULL,
jobend1   DATE NOT NULL,
PERIOD FOR jobdur1(jobst1, jobend1),
jobst2 DATE NOT NULL,
jobend2   DATE NOT NULL,
PERIOD FOR jobdur2(jobst2, jobend2)
) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, CONTAINS is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 CONTAINS jobdur2;
```

The result is:

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

END

Purpose

Returns the ending bound of the period argument.

Syntax

```
END ──┬─ ( period_expression ) ─┬──────────────
      └─ ( derived_period ) ────┘
                                          1145-037A
```

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type.
derived_period	any expression that evaluates to a derived period.

Return Value

The result data type of the END function is same as the element type of the Period expression or the data type of the end column if the argument is a derived period column. If the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the element type of the Period expression or the format of the end column if the argument is a derived period column.

Error Conditions

If an argument of any data type other than a Period data type is passed to the function, an error is reported.

Example 1

Assume the following query:

```
SELECT * FROM employee WHERE END(period1) = DATE '2005-01-07';
```

executes on the *employee* table with PERIOD(DATE) column period1:

```
ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-01-05')
```

```
Adams   Marketing     ('2004-06-19', '2005-02-09')
Mary    Development   ('2004-06-19', '2005-01-05')
Simon   Sales         ('2004-06-22', '2005-01-07')
```

The result is:

```
ename   dept          period1
-----   -----------   ----------------------------
Simon   Sales         ('2004-06-22', '2005-01-07')
```

Example 2

END returns the ending bound (*jdend*) of a derived period called *jobduration* created using the following SQL statement:

```
CREATE TABLE employee(id INTEGER,
                      name VARCHAR(50),
                      jdbegin DATE NOT NULL FORMAT 'YYYY-MM-DD',
                      jdend   DATE NOT NULL FORMAT 'YYYY-MM-DD',
                      PERIOD FOR jobduration(jdbegin,jdend)
  )PRIMARY INDEX(id);
```

The following SQL statements:

```
INSERT INTO employee(1025,'John',DATE'2011-01-02',DATE'2012-05-02');
SELECT END (jobduration) FROM employee;
```

return is:

```
     jdend
__________
2011-05-02
```

EQUALS

Purpose

Evaluates two period expressions or derived periods to TRUE, FALSE, or UNKNOWN.

Syntax

where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type. Note: The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.
derived_period	any expression that evaluates to a derived period.

Result Value

- If both expressions have a Period data type or a derived period, the function returns TRUE if the beginning and ending bound of the first expression is equal to the beginning and ending bound of the second expression; otherwise, the function returns FALSE.
- If either operand is NULL, the operator returns UNKNOWN.

Example 1

Assume that in the *employee1* table, created by the following CREATE TABLE statement, period1 and period2 are PERIOD (DATE) columns.

```
CREATE TABLE employee1 (
       eid INTEGER NOT NULL,
       name VARCHAR(100) NOT NULL,
       deptno INTEGER NOT NULL,
       period1(date),
       period2(date)
```

```
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	Period 1	Period 2
1	Adams	101	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
2	Mary	201	('2005-04-02', '2006-01-03')	('2006-01-03', '2007-02-03')
3	Jones	301	('2004-01-02', '2005-03-05')	('2003-03-05', '2004-01-02')

In the following SQL statement, EQUALS is used with period columns of the employee1 table:

```
SELECT eid, name, depno, period1, period2
FROM employee1
WHERE period1 EQUALS period2;
```

The result is:

EID	Name	DeptNo	Period 1	Period 2
1	Adams	101	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')

Example 2

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
   eid INTEGER NOT NULL,
   name VARCHAR(100) NOT NULL,
   deptno INTEGER NOT NULL,
   jobst1 DATE NOT NULL,
   jobend1   DATE NOT NULL,
   PERIOD FOR jobdur1(jobst1, jobend1),
   jobst2 DATE NOT NULL,
   jobend2   DATE NOT NULL,
   PERIOD FOR jobdur2(jobst2, jobend2)
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, EQUALS is used with derived period columns of the employee table:

```
SELECT eid, name, deptno, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 EQUALS jobdur2;
```

The result is:

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	'DATE'2007-01-01'

IS UNTIL_CHANGED / IS NOT UNTIL_CHANGED

Purpose

Tests whether the ending bound value of a Period expression or a derived period IS UNTIL_CHANGED or IS NOT UNTIL_CHANGED.

Syntax

```
── END ──┬─ ( period_expression ) ─┬── IS ──┬──────┬── UNTIL_CHANGED ──────────
        └─ ( derived_period ) ────┘        └ NOT ─┘
                                                                  1145-044A
```

where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type, including: • PERIOD(TIMESTAMP WITH TIME ZONE) • PERIOD(TIMESTAMP) • PERIOD(DATE) type
derived_period	any expression that evaluates to a derived period.

Usage Notes

You can only compare UNTIL_CHANGED to the ending bound of a Period value with an element type of DATE or TIMESTAMP WITH TIME ZONE. Therefore, the result type of the END function must be DATE or TIMESTAMP WITH TIME ZONE. For information about the END function, see "END" on page 1042.

In comparisons, the precision of the UNTIL_CHANGED value is truncated to the precision of the ending bound value being compared. That is, the number of digits after the decimal point for UNTIL_CHANGED depends upon the precision of the ending bound to which it is compared. The time zone is omitted if the ending bound value has no time zone.

If the ending bound value is NULL, IS [NOT] UNTIL_CHANGED returns UNKNOWN.

You cannot use IS [NOT] UNTIL_CHANGED on the ending bound of a transaction-time column.

Example 1

Consider the following employee table, where the column *eduration* is defined as a PERIOD(DATE) data type:

```
ename               eid  eduration
----------  -----------  ------------------------
Adams            210677  ('05/03/01', '06/05/21')
Gunther          199347  ('04/06/06', '99/12/31')
Montoya          199340  ('04/06/02', '99/12/31')
Chan             210427  ('04/09/24', '99/12/31')
Fuller           197899  ('03/05/27', '03/11/30')
```

The following SQL query uses IS UNTIL_CHANGED to compare the ending bound value of the *eduration* column to UNTIL_CHANGED:

```
SELECT ename, eid
FROM employee
WHERE END(eduration) IS UNTIL_CHANGED;
```

The result is:

```
ename               eid
----------  -----------
Gunther          199347
Montoya          199340
Chan             210427
```

Example 2

If we assume that the *employee* table, created by the following SQL statement:

```
CREATE MULTISET TABLE employee(empno    INTEGER,
                               ename    VARCHAR(50),
                               deptno   INTEGER,
                               jobstart DATE NOT NULL,
                               jobend   DATE NOT NULL,
                               PERIOD FOR jobduration(jobstart, jobend))
                               PRIMARY INDEX(empno);
```

contains the following row:

```
INSERT INTO employee(1025, 'John', '999', DATE'2005-02-03',
UNTIL_CHANGED);
```

The following SELECT statement:

```
SELECT empno,ename(CHAR(30)) FROM employee WHERE END(jobduration) IS
UNTIL_CHANGED;
```

returns:

```
 empno ename
------ ------
  1025   John
```

IS UNTIL_CLOSED / IS NOT UNTIL_CLOSED

Purpose

Tests the ending bound value of a temporal table transaction-time column to see whether the row is open (the ending bound value IS UNTIL_CLOSED) or closed (the ending bound value IS NOT UNTIL_CLOSED).

For more information about temporal tables, see *Temporal Table Support*.

Syntax

where:

Syntax element ...	Specifies ...
period_expression	a reference to a transaction-time column.
derived_period	a reference to a transaction-time column defined using a derived period.

Usage Notes

When a row is created in a temporal table that has a transaction-time dimension (column), Teradata Database sets the ending bound of the column to UNTIL_CLOSED and the row is considered open. When the row is closed, Teradata Database sets the ending bound value to the closing timestamp.

IS UNTIL_CLOSED evaluates to true if the ending bound of the specified transaction-time column is the maximum timestamp value, 9999-12-31 23:59:59.999999+00:00.

Example 1

If we assume that the *employee* table, created by the following SQL statement:

```
CREATE MULTISET TABLE employee(id INTEGER,
                               name VARCHAR(50),
                               dept INTEGER,
                               tt PERIOD(TIMESTAMP(6)WITH TIME ZONE) NOT
                                 NULL AS TRANSACTIONTIME)
                               PRIMARY INDEX(id);
                                 NULL AS TRANSACTIONTIME)
                               PRIMARY INDEX(id);
```

contains the following row, a real period:

```
INSERT INTO employee(102, 'John', 222);
```

The following SELECT statement:

```
SELECT * FROM employee WHERE END(tt) IS UNTIL_CLOSED;
```

returns:

```
 id     name   dept
--------------------
  102   John    222
```

Example 2

If we assume that the *employee* table, created by the following SQL statement:

```
CREATE MULTISET TABLE employee(empno    INTEGER,
                               ename    VARCHAR(50),
                               deptno   INTEGER,
                               jobstart TIMESTAMP WITH TIME ZONE
                                   NOT NULL,
                               jobend   TIMESTAMP WITH TIME ZONE
                                   NOT NULL,
                               PERIOD FOR jobduration(jobstart,
                                   jobend) AS
                               TRANSACTIONTIME)
                               PRIMARY INDEX(empno);
```

contains the following row:

```
NONTEMPORAL
INSERT INTO employee(1025, 'John', '999',TIMESTAMP'2005-02-03
     12:12:12.123456+00:00', UNTIL_CLOSED);
```

The following SELECT statement:

```
SELECT empno,ename(CHAR(30)) FROM employee WHERE END(jobduration) IS
UNTIL_CLOSED;
```

returns:

```
 empno  ename
------ ------
  1025   John
```

IMMEDIATELY PRECEDES

Purpose

Evaluates two Period expressions or derived periods to TRUE, FALSE, or UNKNOWN.

Syntax

where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type. Note: The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.
derived_period	any expression that evaluates to a derived period.

Result Value

The function returns TRUE if the beginning bound of the first expression is less than or equal to the beginning bound of the second expression and the ending bound of the first expression is greater than or equal to the ending bound of the second expression; otherwise, the function returns FALSE.

If either expression is NULL, the operator returns UNKNOWN.

Example 1

Assume that in *employee1* table, created by the following CREATE TABLE statement, period1 and period2 are PERIOD (DATE) columns.

```
CREATE TABLE employee1 (
       eid INTEGER NOT NULL,
       name VARCHAR(100) NOT NULL,
       deptno INTEGER NOT NULL,
       period1(date),
       period2(date)
```

```
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	Period 1	Period 2
1	Adams	101	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
2	Mary	201	('2005-04-02', '2006-01-03')	('2006-01-03', '2007-02-03')
3	Jones	301	('2004-01-02', '2005-03-05')	('2003-03-05', '2004-01-02')

In the following SQL statement, IMMEDIATELY PRECEDES is used with period columns of the employee1 table:

```
SELECT eid, name, depno, period1, period2
FROM employee1
WHERE period1 IMMEDIATELY PRECEDES period2;
```

The result is:

EID	Name	DeptNo	Period 1	Period 2
2	Mary	201	('2005-04-02', '2006-01-03')	('2006-01-03', '2007-02-03')

Example 2

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
   eid INTEGER NOT NULL,
   name VARCHAR(100) NOT NULL,
   deptno INTEGER NOT NULL,
   jobst1 DATE NOT NULL,
   jobend1   DATE NOT NULL,
   PERIOD FOR jobdur1(jobst1, jobend1),
   jobst2 DATE NOT NULL,
   jobend2   DATE NOT NULL,
   PERIOD FOR jobdur2(jobst2, jobend2)
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, IMMEDIATELY PRECEDES is used with derived period columns of the employee table:

```
SELECT eid, name, deptno, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 IMMEDIATELY PRECEDES jobdur2;
```

The result is:

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'

IMMEDIATELY SUCCEEDS

Purpose

Evaluates two Period expressions or derived periods to TRUE, FALSE, or UNKNOWN.

Syntax

where:

Syntax element ...	Specifies ...
period_expression	any expression that evaluates to a Period data type. Note: The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.
derived_period	any expression that evaluates to a derived period.

Result Value

The function returns TRUE if the beginning bound of the first expression is less than or equal to the beginning bound of the second expression and the ending bound of the first expression is greater than or equal to the ending bound of the second expression; otherwise, the function returns FALSE.

If either expression is NULL, the operator returns UNKNOWN.

Example 1

Assume that in *employee1* table, created by the following CREATE TABLE statement, period1 and period2 are PERIOD (DATE) columns

```
CREATE TABLE employee1 (
       eid INTEGER NOT NULL,
       name VARCHAR(100) NOT NULL,
       deptno INTEGER NOT NULL,
       period1(date),
       period2(date)
```

```
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	Period 1	Period 2
1	Adams	101	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
2	Mary	201	('2005-04-02', '2006-01-03')	('2006-01-03', '2007-02-03')
3	Jones	301	('2004-01-02', '2005-03-05')	('2003-03-05', '2004-01-02')

In the following SQL statement, IMMEDIATELY SUCCEEDS is used with period columns of the employee1 table:

```
SELECT eid, name, deptno, period1, period2
FROM employee1
WHERE period1 IMMEDIATELY SUCCEEDS period2;
```

The result is:

EID	Name	DeptNo	Period 1	Period 2
3	Jones	301	('2004-01-02', '2005-03-05')	('2003-03-05', '2004-01-02')

Example 2

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
   eid INTEGER NOT NULL,
   name VARCHAR(100) NOT NULL,
   deptno INTEGER NOT NULL,
   jobst1 DATE NOT NULL,
   jobend1   DATE NOT NULL,
   PERIOD FOR jobdur1(jobst1, jobend1),
   jobst2 DATE NOT NULL,
   jobend2   DATE NOT NULL,
   PERIOD FOR jobdur2(jobst2, jobend2)
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, IMMEDIATELY SUCCEEDS is used with derived period columns of the employee table:

```
SELECT eid, name, depno, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 IMMEDIATELY SUCCEEDS jobdur2;
```

The result is:

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'

INTERVAL

Purpose

Finds the difference between the ending and beginning bounds of a Period expression or a derived period and returns the difference as the duration of the argument in a specified interval qualifier.

Syntax

```
INTERVAL ─┬─ ( period_expression ) ─┬──────────── interval_qualifier
          └─ ( derived_period ) ────┘
                                                  1145-039A
```

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type.
derived_period	any expression that evaluates to a derived period.
interval_qualifier	any interval qualifier appropriate for the argument's element type. The interval qualifiers are as follows: Year-Month intervals: • YEAR • YEAR TO MONTH • MONTH Day-Time intervals: • DAY • DAY TO HOUR, MINUTE or SECOND • HOUR • HOUR TO MINUTE or SECOND • MINUTE • MINUTE to SECOND • SECOND

Return Value

The result of the INTERVAL (p) IQ function is the value of (END(p) - BEGIN(p)) IQ, where argument p is a Period expression and IQ is an interval qualifier.

If the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the interval data type corresponding to the specified interval qualifier.

The title is INTERVAL(*period _expression*) *interval_qualifier*.

Error Conditions

An error may be reported:

- If the argument of the INTERVAL function does not have a Period data type.
- If the argument has a PERIOD(DATE) data type and the interval qualifier is not YEAR, YEAR TO MONTH, MONTH, or DAY.
- If the argument has a PERIOD(TIME(n) [WITH TIME ZONE]) data type and the interval qualifier is not HOUR, HOUR TO MINUTE, HOUR TO SECOND, MINUTE, MINUTE TO SECOND or SECOND.
- If the result of an INTERVAL expression violates the rules specified for the precision of an interval qualifier, an error is reported. For example, assume p1 is a PERIOD(TIMESTAMP(0)) expression that has a value of PERIOD '(2006-01-01 12:12:12, 2007-01-01 12:12:12)'. If INTERVAL(p1) DAY is specified, the default precision for the DAY interval qualifier is 2, and, since the result is 365 days which is a three-digit value that cannot fit into a DAY(2) interval qualifier, an error is reported.
- If the argument of the INTERVAL function is a period of a DATE or TIMESTAMP(n) [WITH TIME ZONE] and the ending bound value is UNTIL_CHANGED.

Example 1

Assume the following query is executed on the *employee* table with PERIOD(DATE) column period1:

```
SELECT INTERVAL (period1) MONTH FROM employee;

ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-03-05')
```

The result is:

```
INTERVAL(period) MONTH
-----------------------
                      2
```

Example 2

INTERVAL returns the interval value of a derived period called *jobduration* created using the following SQL statement:

```
CREATE TABLE employee(id INTEGER,
                      name VARCHAR(50),
                      jdbegin DATE NOT NULL FORMAT 'YYYY-MM-DD',
                      jdend   DATE NOT NULL FORMAT 'YYYY-MM-DD',
                      PERIOD FOR jobduration(jdbegin,jdend)
```

```
  )PRIMARY INDEX(id);
```

When the following values are inserted into the *employee* table:

```
INSERT INTO employee(1025,'John',DATE'2011-01-02',DATE'2012-05-02');
```

the following SQL statement:

```
SELECT INTERVAL(jobduration)MONTH(4) FROM employee;
```

returns:

```
    INTERVAL(jobduration) MONTH
  ----------------------------
                            16
```

LAST

Purpose

Returns the last value of the Period argument (that is, the ending bound minus one granule of the element type of the argument).

Syntax

```
LAST ─┬─ ( period_expression ) ─┬──────────────
      └─ ( derived_period ) ────┘
                                 1145-038A
```

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type.
derived_period	any expression that evaluated to a derived period.

Return Value

The result type of the LAST function is same as the element type of the Period expression or the data type of the begin or end column if the argument is a derived period. If the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the element type of the Period expression or the format of the begin or end column if the argument is a derived period.

Error Conditions

If an argument has a data type other than a Period data type, an error is reported.

Example 1

Assume the following query is executed on the *employee* table PERIOD(DATE) column period1:

```
SELECT * FROM employee WHERE LAST(period1) = DATE '2004-01-04';

ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-01-05')
```

```
Adams   Marketing     ('2004-06-19', '2005-02-09')
Mary    Development   ('2004-06-19', '2005-01-05')
Simon   Sales         ('2004-06-22', '2005-01-07')
```

The result is:

```
ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-01-05')
```

Example 2

LAST returns the last value of a derived period called *jobduration* created by the following SQL statement:

```
CREATE TABLE employee(id INTEGER,
                      name VARCHAR(50),
                      jdbegin DATE NOT NULL FORMAT 'YYYY-MM-DD',
                      jdend   DATE NOT NULL FORMAT 'YYYY-MM-DD',
                      PERIOD FOR jobduration(jdbegin,jdend)
  )PRIMARY INDEX(id);
```

When the following values are inserted into the *employee* table:

```
INSERT INTO employee(1025,'John',DATE'2011-01-02',DATE'2012-05-02');
```

the following SQL statement:

```
SELECT LAST (jobduration) FROM employee;
```

returns:

```
         LAST(jobduration)
  ------------------------
                2012-05-01
```

MEETS

Purpose

Evaluates two period expressions, or derived periods, or DateTime expressions and evaluates to TRUE, FALSE, or UNKNOWN.

Syntax

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type. Note: The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
derived_period	any expression that evaluates to a derived period.

Result Value

- If both expressions have a Period data type or a derived period, the function returns TRUE if the ending bound of the first expression is equal to the beginning bound of the second expression or the ending bound of the second expression is equal to the beginning bound of the first expression; otherwise, the function returns FALSE.
- If one expression is a Period data type and the other expression is a DateTime expression, the function returns TRUE if the ending bound of the Period expression is equal to the DateTime expression or if the DateTime expression plus one granule is equal to the beginning bound of the Period expression; otherwise, the function returns FALSE.
- If either expression is NULL, the function returns UNKNOWN.

Error Conditions

If either expression evaluates to a data type other than a Period or DateTime, an error is reported.

If the expressions are not comparable, an error is reported.

Example 1

Assume the following query is executed on the *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period2 MEETS period1;
```

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	period1	period2
Jones	('2004-01-02','2004-03-05')	('2004-03-05', '2004-10-07')

Example 2

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
   eid INTEGER NOT NULL,
   name VARCHAR(100) NOT NULL,
   deptno INTEGER NOT NULL,
   jobst1 DATE NOT NULL,
   jobend1   DATE NOT NULL,
   PERIOD FOR jobdur1(jobst1, jobend1),
   jobst2 DATE NOT NULL,
   jobend2   DATE NOT NULL,
   PERIOD FOR jobdur2(jobst2, jobend2)
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, MEETS is used with derived period columns of the employee table:

```
SELECT eid, name, deptno, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 MEETS jobdur2;
```

The result is:

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'

NEXT

Purpose

Returns the succeeding value of the argument so that there is one granule of the argument type between the argument and the returned value.

Syntax

```
── NEXT (datetime_expression) ──────────────────────────
                                                 1101A579
```

where:

Syntax element...	Specifies...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.

Return Value

The return data type is the same as that of the argument. If the value of the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the data type of the proximity argument.

Error Conditions

If the argument does not have a DateTime data type, an error is reported.

If the result is outside the permissible range of a value for the argument's data type, an error is reported. For example, if NEXT(DATE '9999-12-31') is specified, an error is reported.

Example

Assume the following query is executed on the *employee* table where period1 is a PERIOD(DATE) column:

```
SELECT *
FROM employee
WHERE NEXT(END(period1)) = DATE '2004-03-06';

ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-03-05')
```

```
Simon   Sales         ?
```

The result is:

```
ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-03-05')
```

OVERLAPS

Purpose

Tests whether two time periods overlap each another.

Syntax

where:

Syntax element ...	Specifies ...
datetime_expression	a start and end DateTime.
interval_expression	an end DateTime.
row_subquery	an element of a row subquery in a SELECT statement. The subquery cannot specify a SELECT AND CONSUME statement.
period_expression	any expression that evaluates to a Period data type.
derived_period	any expression that evaluates to a derived period.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Time Periods

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*.

Note: Implicit casting to a Period data type is not supported.

Results

Consider the general case of an OVERLAPS comparison, stated as follows.

```
(S1, E1) OVERLAPS (S2, E2)
```

The result of OVERLAPS is as follows.

```
(S1 > S2 AND NOT (S1 >= E2 AND E1 >= E2))
OR
(S2 > S1 AND NOT (S2 >= E1 AND E2 >= E1))
OR
(S1 = S2 AND (E1 = E2 OR E1 <> E2))
```

For Period data types or derived periods, where p1 is the first Period expression or derived period and p2 is the second Period expression or derived period, the values of S1, E1, S2, and E2 are as follows:

```
S1 = BEGIN(p1)
E1 = END(p1)
S2 = BEGIN(p2)
E2 = END(p2)
```

Rules

- When you specify two DateTime types, they must be comparable.
- When you specify two Period types, including derived periods, they must be comparable.
- If you specify a Period type for either one or both time periods, the Period expression must not include an explicit NULL.
- If the first columns of each left and right time periods are DateTime types, they must have the same data type: both DATE, both TIME, or both TIMESTAMP.
- If only one time period is a Period type, the first column of the other time period must have the same data type as the element type of the Period.
- If neither time period is a Period type, then the second column of each left and right time period must either be the same DateTime type as its corresponding first column or it must be an Interval type that involves only DateTime fields where the precision is such that its value can be added to that of the corresponding DateTime type.

Example 1

The following example compares two time periods that share a single common point, CURRENT_TIME.

The result returned is FALSE because when two time periods 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

The following example uses the *datetime_expression, datetime_expression* form of OVERLAPS. The two DATE periods overlap each other; thus 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 in the following example means the second *datetime_expression* has a start time of 2001-06-13 15:00:00 and a null end time.

```
SELECT 'OVERLAPS'
WHERE (TIMESTAMP '2001-06-12 10:00:00', TIMESTAMP '2001-06-15
08:00:00')
OVERLAPS (TIMESTAMP '2001-06-13 15:00:00', NULL);
```

Because the start time for the second expression falls within the TIMESTAMP interval defined by the first expression, the result is TRUE.

Example 6

Assume the following query is executed on the *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period2 OVERLAPS period1;
```

Ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')

Ename	period1	period2
Simon	?	('2005-02-03', '2005-07-27')

The result is:

Ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')

Example 7

Consider the following table and query:

```
CREATE TABLE project
   (id INTEGER,
    analysis_phase PERIOD(DATE))
UNIQUE PRIMARY INDEX (id);

INSERT project (1, PERIOD(DATE'2010-06-21',DATE'2010-06-25'));

SELECT 'OVERLAPS'
FROM project
WHERE analysis_phase OVERLAPS
      PERIOD(DATE'2010-06-24',NULL);
```

The SELECT statement returns an error because one of the operands of OVERLAP is a Period type with a Period expression specifying an explicit NULL.

Example 8

Assume that in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
   eid INTEGER NOT NULL,
   name VARCHAR(100) NOT NULL,
   deptno INTEGER NOT NULL,
   jobst1 DATE NOT NULL,
   jobend1   DATE NOT NULL,
   PERIOD FOR jobdur1(jobst1, jobend1),
   jobst2 DATE NOT NULL,
   jobend2   DATE NOT NULL,
   PERIOD FOR jobdur2(jobst2, jobend2)
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, OVERLAPS is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 OVERLAPS jobdur2;
```

The result is:

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

P_INTERSECT

Purpose

Returns the portion of the Period expression that is common to the Period expressions if they overlap.

Syntax

```
── period_expression ── P_INTERSECT ── period_expression ──
```

1101A584

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type. Note: The Period expressions specified must be comparable. Implicit casting to a Period data type is not supported.

Result Value

- If the Period expressions do not overlap, the result is NULL. If either Period expression is NULL, the result is NULL. Otherwise, the result has a Period data type that is comparable to the Period expressions.
- If the Period expressions have PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) or PERIOD(TIME(n) [WITH TIME ZONE]) data types but different precisions, the result is a Period value of the higher precision data type. If neither Period expression has a time zone, the resulting period does not have a time zone; otherwise, the resulting period has a time zone and the value of the time zone in the result is determined using the following rules:
  - If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the Period expression as defined by the Period value constructor that follows.
  - If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming *p1* and *p2* are Period expressions and the result element type as determined above is *rt*, the result of p1 P_INTERSECT p2 is as follows if p1 OVERLAPS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(BEGIN(p1) AS rt) >= CAST(BEGIN(p2) AS rt)
     THEN CAST(BEGIN(p1) AS rt)
```

```
    ELSE CAST(BEGIN(p2) AS rt) END,
  CASE WHEN CAST(END(p1) AS rt) <= CAST(END(p2) AS rt)
    THEN CAST(END(p1) AS rt)
    ELSE CAST(END(p2) AS rt) END)
```

Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the P_INTERSECT operation is performed if they overlap.

Format and Title

The format is the default format for the resulting Period data type.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

In the following example, the P_INTERSECT operator is used in the selection list.

```
SELECT period2 P_INTERSECT period1
FROM product_tests
WHERE pid = 11804;
```

Assume the query is executed on the following table *product_tests* where period1 is a PERIOD(TIME(1)) column and period2 is a PERIOD(TIME(0)) column:

```
pid     period1                          period2
-----   ----------------------------     -------------------------
11804   ('10:10:10.1', '11:10:10.1')     ('10:10:10', '10:10:11')
10996   ('11:10:10.1', '11:40:40.1')     ('10:10:10', '10:10:11')
```

The result is as follows:

```
(period2 P_INTERSECT period1)
-----------------------------
('10:10:10.1', '10:10:11.0')
```

P_NORMALIZE

Purpose

Returns a Period value that is the combination of the two Period expressions if the Period expressions overlap or meet. If the Period expressions neither meet nor overlap, the function returns NULL. If either Period expression is NULL, the function returns NULL.

Syntax

```
── period_expression ── P_NORMALIZE ── period_expression ──
```

1101A594

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type. Note: The Period expressions specified must be comparable. Implicit casting to a Period data type is not supported.

Result Value

- Assuming p1 and p2 are comparable Period expressions and ((BEGIN(p1) >= BEGIN(p2) AND BEGIN(p1) <= END(p2)) OR (BEGIN(p2) >= BEGIN(p1) AND BEGIN(p2) <= END(p1))) is TRUE, p1 P_NORMALIZE p2 returns PERIOD(minimum(BEGIN(p1), BEGIN(p2)), maximum(END(p1), END(p2))).
- If either Period expression is NULL or ((BEGIN(p1) >= BEGIN(p2) AND BEGIN(p1) <= END(p2)) OR (BEGIN(p2) >= BEGIN(p1) AND BEGIN(p2) <= END(p1))) is FALSE, the result is NULL. P_NORMALIZE returns a Period value if the Period expressions satisfy the MEETS or OVERLAPS condition.
- If the Period expressions have PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) data type but have different precisions, the result has the higher of the two precisions. If one of the Period expressions contains a time zone, the result contains a time zone for each element. The result time zones are determined using the following rules:
  - If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the Period expressions as defined by the Period value constructor that follows.
  - If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming p1 and p2 are Period expressions and the result element type as determined above is rt, the result of p1 P_NORMALIZE p2 is as follows if p1 OVERLAPS p2 OR p1 MEETS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(BEGIN(p1) AS rt) <= CAST(BEGIN(p2) AS rt)
     THEN CAST(BEGIN(p1) AS rt)
     ELSE CAST(BEGIN(p2) AS rt) END,
  CASE WHEN CAST(END(p1) AS rt) >= CAST(END(p2) AS rt)
     THEN CAST(END(p1) AS rt)
     ELSE CAST(END(p2) AS rt) END)
```

Internally, Period values are saved in UTC and the OVERLAPS or MEETS operator is evaluated using these UTC represented formats and the P_NORMALIZE operation is performed if they overlap or meet.

Format and Title

The format is the default format for the resulting Period data type.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

In the following example, the P_NORMALIZE operator is used to collapse two Period columns.

```
SELECT period2 P_NORMALIZE period1
FROM product_tests
WHERE pid = 11215;
```

Assume the query is executed on the following table *product_tests* where period1 is PERIOD(TIME(1)) column and period2 is PERIOD(TIME(0)) column:

```
pid     period1                          period2
-----   ----------------------------     -------------------------
11804   ('10:10:10.1', '11:10:10.1')     ('10:10:10', '10:10:11')
10996   ('11:10:10.1', '11:40:40.1')     ('10:10:10', '10:10:11')
11215   ('10:40:10.1', '11:20:20.1')     ('11:10:10', '11:50:10')
```

The result is as follows:

```
(period2 P_NORMALIZE period1)
-----------------------------
('10:40:10.1', '11:50:10.0')
```

PRECEDES

Purpose

Evaluates two Period expressions or derived periods, or DateTime expressions to TRUE, FALSE, or UNKNOWN.

Syntax

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type. Note: The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
derived_period	any expression that evaluates to a derived period.

Result Value

- If both expressions have a Period data type or a derived period, the function returns TRUE if the ending bound of the first expression is less than or equal to the beginning bound of the second expression; otherwise, the function returns FALSE.
- If the first expression is a Period expression and the second expression is a DateTime expression, the function returns TRUE if the ending bound of the first expression is less than or equal to the second expression; otherwise, the function returns FALSE.
- If the first expression is a DateTime expression and the second expression has a Period data type, the function returns TRUE if the first expression is less than the beginning bound of the second expression; otherwise, the function returns FALSE.

- If either expression is NULL, the operator returns UNKNOWN.

Error Conditions

If either expression is other than a Period data type or a DateTime value expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example 1

Assume the following query is executed on the *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period1 PRECEDES period2;
```

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	period1	period2
Jones	('2004-01-02','2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02','2004-03-05')	('2004-03-07', '2004-10-07')

Example 2

Assume in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
   eid INTEGER NOT NULL,
   name VARCHAR(100) NOT NULL,
   deptno INTEGER NOT NULL,
   jobst1 DATE NOT NULL,
   jobend1   DATE NOT NULL,
   PERIOD FOR jobdur1(jobst1, jobend1),
   jobst2 DATE NOT NULL,
   jobend2   DATE NOT NULL,
   PERIOD FOR jobdur2(jobst2, jobend2)
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, PRECEDES is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 PRECEDES jobdur2;
```

The result is:

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'

PRIOR

Purpose

Returns the preceding value of the argument so that there is one granule of the argument type between the returned value and the argument.

Syntax

```
── PRIOR (datetime_expression) ──────────────────────
                                                1101A578
```

where:

Syntax element...	Specifies...
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.

Return Value

The return data type is the same as that of the argument. If the value of the argument is NULL, the result is NULL.

Format and Title

The format is the default format for the argument's data type.

Restrictions

The built-in function DATE is not supported as an argument to the PRIOR function. For example, PRIOR(DATE) is no valid. You must use PRIOR(CURRENT_DATE) instead.

Error Conditions

If the argument does not have a DateTime data type, an error is reported.

If the result is outside the permissible range of the argument's data type, an error is reported. For example, if PRIOR(DATE '0001-01-01') is specified, an error is reported.

Example

Assume the following query is executed on the *employee* table where period1 is a PERIOD(DATE) column:

```
SELECT *
FROM employee
```

```
WHERE PRIOR(END(period1)) = DATE '2004-03-04';
ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-03-05')
Simon   Sales         ?
```

The result is:

```
ename   dept          period1
-----   -----------   ----------------------------
Jones   Sales         ('2004-01-02', '2004-03-05')
```

LDIFF

Purpose

Returns the portion of the first Period expression that exists before the beginning of the second Period expression when the Period expressions overlap.

When the Period expressions overlap but there is no portion of the first Period expression before the beginning of the second Period expression or the Period expressions do not overlap, the function returns NULL. If either Period expression is NULL, LDIFF returns NULL.

Syntax

—— *period_expression* —— LDIFF —— *period_expression* ——

1101A592

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type. Note: The Period expressions specified must be comparable. Implicit casting to a Period data type is not supported.

Result Value

- Assuming p1 and p2 are comparable Period expressions, p1 LIDFF p2 returns PERIOD(BEGIN(p1), BEGIN(p2)) if p1 OVERLAPS p2 is TRUE and BEGIN(p1) is less than BEGIN(p2). If either Period expression is NULL, p1 OVERLAPS p2 is FALSE, or BEGIN(p1) is not less than BEGIN(p2), the result is NULL.
- If the Period expressions have PERIOD(TIME(n) [WITH TIME ZONE]) or PERIOD(TIMESTAMP(n) [WITH TIME ZONE]) data types but have different precisions, the result has the higher of the two precisions. If one of the Period expressions contains time zones and the other does not, the result contains a time zone for each element. The result time zones are evaluated using the following rules:
  - If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the expressions as defined by the Period value constructor that follows.
  - If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming p1 and p2 are Period expressions and the result element type as determined above is rt, the result of p1 LDIFF p2 is as follows if p1 OVERLAPS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(BEGIN(p1) AS rt) < CAST(BEGIN(p2) AS rt)
     THEN CAST(BEGIN(p1) AS rt)
     ELSE NULL END,
  CASE WHEN CAST(BEGIN(p1) AS rt) < CAST(BEGIN(p2) AS rt)
     THEN CAST(BEGIN(p2) AS rt)
     ELSE NULL END)
```

Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the LDIFF operation is performed if they overlap.

Format and Title

The format is the default format for the resulting Period data type.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

LDIFF is used to find the left difference of the first Period expression with the second Period expression.

```
SELECT ename, period2 LDIFF period1 FROM employee;
```

Assume the query is executed on the following *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
ename   period1                          period2
-----   ----------------------------     ----------------------------
Adams   ('2005-02-03', '2006-02-03')     ('2005-02-03', '2006-02-03')
Mary    ('2005-04-02', '2006-01-03')     ('2005-02-03', '2006-02-03')
Jones   ('2004-01-02', '2004-03-05')     ('2002-03-05', '2004-10-07')
Randy   ('2006-01-02', '2007-03-05')     ('2004-03-07', '2005-10-07')
Simon   ?                                ('2005-02-03', '2005-07-27')
```

The result is:

```
ename   (period2 LDIFF period1)
-----   ----------------------------
Adams   ?
Mary    ('2005-02-03', '2005-04-02')
Jones   ('2002-03-05', '2004-01-02')
Randy   ?
Simon   ?
```

RDIFF

Purpose

Returns the portion of the first Period expression that exists from the end of the second Period expression when the Period expressions overlap. When the Period expressions overlap but there is no portion of the first Period expression from the end of the second Period expression or if the Period expressions do not overlap, RDIFF returns NULL. If either Period expression is NULL, RDIFF returns NULL.

Syntax

```
── period_expression ── RDIFF ── period_expression ──
```

1101A593

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type. Note: The Period expressions specified must be comparable. Implicit casting to a Period data type is not supported.

Result Value

- Assuming p1 and p2 are comparable Period expressions, p1 RDIFF p2 returns PERIOD(END(p2), END(p1)) if p1 OVERLAPS p2 is TRUE and END(p1) is greater than END(p2). If either Period expression is NULL, p1 OVERLAPS p2 is FALSE, or END(p1) is not greater than END(p2), the result is NULL.
- If the Period expressions have PERIOD(TIME[(n)] [WITH TIME ZONE]) or PERIOD(TIMESTAMP[(n)] [WITH TIME ZONE]) data types but have different precisions, the result has the higher of the two precisions. If one of the Period expressions contains time zones and the other does not, the result contains a time zone for each element. The result time zones are evaluated using the following rules:
  - If both Period expressions have a time zone, the time zone displacement of a result bound is obtained from the corresponding bound of the Period expressions as defined by the Period value constructor that follows.
  - If only one of the Period expressions has a time zone, the other Period expression is considered to be at the current session time zone and the result is computed as follows.

Assuming p1 and p2 are Period expressions and the result element type as determined above is rt, the result of p1 RDIFF p2 is as follows if p1 OVERLAPS p2 is TRUE:

```
PERIOD(
  CASE WHEN CAST(END(p1) AS rt) > CAST(END(p2) AS rt)
     THEN CAST(END(p2) AS rt)
     ELSE NULL END,
  CASE WHEN CAST(END(p1) AS rt) > CAST(END(p2) AS rt)
     THEN CAST(END(p1) AS rt)
     ELSE NULL END)
```

Internally, Period values are saved in UTC and the OVERLAPS operator is evaluated using these UTC represented formats and the RDIFF operation is performed if they overlap.

Format and Title

The format is the default format for the resulting Period data type.

Error Conditions

If either expression is not a Period expression, an error is reported.

If the Period expressions are not comparable, an error is reported.

Example

RDIFF is used to find the right difference of the first Period expression with the second Period expression.

Assume the query is executed on the following *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT ename, period2 RDIFF period1 FROM employee;
ename   period1                        period2
-----   ----------------------------   ----------------------------
Adams   ('2005-02-03', '2006-02-03')   ('2005-02-03', '2006-02-03')
Mary    ('2005-04-02', '2006-01-03')   ('2005-02-03', '2006-02-03')
Jones   ('2001-01-02', '2003-03-05')   ('2002-03-05', '2004-10-07')
Randy   ('2006-01-02', '2007-03-05')   ('2004-03-07', '2005-10-07')
Simon   ?                              ('2005-02-03', '2005-07-27')
```

The result is:

```
ename   (period2 RDIFF period1)
-----   ----------------------------
Adams   ?
Mary    ('2006-01-03', '2006-02-03')
Jones   ('2003-03-05', '2004-10-07')
Randy   ?
Simon   ?
```

SUCCEEDS

Purpose

Evaluates two period expressions, or derived periods, or DateTime expressions to TRUE, FALSE, or UNKNOWN.

Syntax

where:

Syntax element...	Specifies...
period_expression	any expression that evaluates to a Period data type. Note: The Period expression specified must be comparable with the other expression. Implicit casting to a Period data type is not supported.
datetime_expression	any expression that evaluates to a DATE, TIME, or TIMESTAMP data type.
derived_period	any expression that evaluates to a derived period.

Result Value

- If both expressions have a Period data type or a derived period, the function returns TRUE if the beginning bound of the first expression is greater than or equal to the ending bound of the second expression; otherwise, the function returns FALSE.
- If the first expression is a Period expression and the second expression is a DateTime expression, the function returns TRUE if the beginning bound of the first expression is greater than the second expression; otherwise, the function returns FALSE.
- If the first expression is a DateTime expression and the second expression is a Period expression, the function returns TRUE if the DateTime expression is greater than or equal to the ending bound of the second expression; otherwise, the function returns FALSE.
- If either expression is NULL, the operator returns UNKNOWN.

Error Conditions

If either expression is other than a Period data type or a DateTime value expression, an error is reported.

If the expressions are not comparable types, an error is reported.

Example 1

Assume the query is executed on the following *employee* table where period1 and period2 are PERIOD(DATE) columns:

```
SELECT * FROM employee WHERE period2 SUCCEEDS period1;
```

ename	period1	period2
Adams	('2005-02-03', '2006-02-03')	('2005-02-03', '2006-02-03')
Mary	('2005-04-02', '2006-01-03')	('2005-02-03', '2006-02-03')
Jones	('2004-01-02', '2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02', '2004-03-05')	('2004-03-07', '2004-10-07')
Simon	?	('2005-02-03', '2005-07-27')

The result is:

ename	period1	period2
Jones	('2004-01-02','2004-03-05')	('2004-03-05', '2004-10-07')
Randy	('2004-01-02','2004-03-05')	('2004-03-07', '2004-10-07')

Example 2

Assume in the *employee* table, created by the following CREATE TABLE statement, jobdur1 and jobdur2 are derived period columns.

```
CREATE TABLE employee (
   eid INTEGER NOT NULL,
   name VARCHAR(100) NOT NULL,
   deptno INTEGER NOT NULL,
   jobst1 DATE NOT NULL,
   jobend1   DATE NOT NULL,
   PERIOD FOR jobdur1(jobst1, jobend1),
   jobst2 DATE NOT NULL,
   jobend2   DATE NOT NULL,
   PERIOD FOR jobdur2(jobst2, jobend2)
   ) PRIMARY INDEX(eid);
```

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
1	Tom	101	DATE'2001-01-01'	DATE'2004-01-01'	DATE'2005-01-01'	DATE'2006-01-01'
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
3	Joo	301	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2006-01-01'	DATE'2007-01-01'
4	Tam	401	DATE'2001-01-01'	DATE'2006-01-01'	DATE'2002-01-01'	DATE'2004-01-01'
5	Pat	501	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2006-01-01'	DATE'2008-01-01'
6	Jack	601	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2007-01-01'	DATE'2008-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'
8	Tim	801	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2005-01-01'	DATE'2007-01-01'

In the following SQL statement, SUCCEEDS is used with derived period columns of the employee table:

```
SELECT eid, name, jobst1, jobend1, jobst2, jobend2
FROM employee
WHERE jobdur1 SUCCEEDS jobdur2;
```

The result is:

EID	Name	DeptNo	JobSt1	JobEnd1	JobSt2	JobEnd2
2	Rick	201	DATE'2005-01-01'	DATE'2006-01-01'	DATE'2001-01-01'	DATE'2004-01-01'
7	Yu	701	DATE'2005-01-01'	DATE'2007-01-01'	DATE'2004-01-01'	DATE'2005-01-01'

TD_NORMALIZE_OVERLAP

Purpose

Combines the rows whose Period values overlap so that the resulting normalized row contains the earliest beginning bound and the latest ending bound from the Period values of all the rows involved.

Syntax

```
┬──────────────────┬─ TD_NORMALIZE_OVERLAP ── (grouping_column_list, period_column) ──
└─ TD_SYSFNLIB. ───┘
```

1101A739

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_NORMALIZE_OVERLAP is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_NORMALIZE_OVERLAP is a table function that takes two arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is the Period column where you want to find the Period values that overlap.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You can specify an optional INTEGER output column at the end of the RETURNS clause to contain a count of the rows that were normalized.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, duration) AS
   (SELECT flight_id, duration FROM FlightExp)
SELECT *
FROM TABLE (TD_SYSFNLIB.TD_NORMALIZE_OVERLAP(NEW VARIANT_TYPE(subtbl.flight_id),
                                             subtbl.duration)
RETURNS (flight_id INT, duration PERIOD(TIMESTAMP(6) WITH TIME ZONE), NrmCount INT)
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, duration, NrmCount) ORDER BY 1,2;
```

TD_NORMALIZE_MEET

Purpose

Combines the rows whose Period values meet so that the resulting normalized row contains the earliest beginning bound and the latest ending bound from the Period values of all the rows involved.

Syntax

```
┬──────────────────┬─ TD_NORMALIZE_MEET ── (grouping_column_list, period_column) ──────
└─ TD_SYSFNLIB. ─┘
```

1101A740

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_NORMALIZE_MEET is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_NORMALIZE_MEET is a table function that takes two arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is the Period column where you want to find the Period values that meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You can specify an optional INTEGER output column at the end of the RETURNS clause to contain a count of the rows that were normalized.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, duration) AS
   (SELECT flight_id, duration FROM FlightExp)
SELECT *
FROM TABLE (TD_SYSFNLIB.TD_NORMALIZE_MEET(NEW VARIANT_TYPE(subtbl.flight_id),
                                          subtbl.duration)
RETURNS (flight_id INT, duration PERIOD(TIMESTAMP(6) WITH TIME ZONE), NrmCount INT)
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, duration, NrmCount) ORDER BY 1,2;
```

TD_NORMALIZE_OVERLAP_MEET

Purpose

Combines the rows whose Period values either meet or overlap so that the resulting normalized row contains the earliest beginning bound and the latest ending bound from the Period values of all the rows involved.

Syntax

```
┌──────────────────┐
└─ TD_SYSFNLIB. ───┘─ TD_NORMALIZE_OVERLAP_MEET ── (grouping_column_list, period_column) ─
```

1101A741

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_NORMALIZE_OVERLAP_MEET is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_NORMALIZE_OVERLAP_MEET is a table function that takes two arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You can specify an optional INTEGER output column at the end of the RETURNS clause to contain a count of the rows that were normalized.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, duration) AS
   (SELECT flight_id, duration FROM FlightExp)
SELECT *
FROM TABLE (TD_SYSFNLIB.TD_NORMALIZE_OVERLAP_MEET(NEW VARIANT_TYPE(subtbl.flight_id),
                                                  subtbl.duration)
RETURNS (flight_id INT, duration PERIOD(TIMESTAMP(6) WITH TIME ZONE), NrmCount INT)
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, duration, NrmCount) ORDER BY 1,2;
```

TD_SUM_NORMALIZE_OVERLAP

Purpose

Finds the sum of a column for all the rows that were normalized because their Period values overlap.

Syntax

```
┌──────────────────────┐
│                      ├─ TD_SUM_NORMALIZE_OVERLAP ── (grouping_column_list, numeric_column, period_column) ──────
└─ TD_SYSFNLIB. ───────┘
```

1101A742

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
numeric_column	a numeric column on which SUM() is requested. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_SUM_NORMALIZE_OVERLAP is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_SUM_NORMALIZE_OVERLAP is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is a numeric column on which SUM() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.

- The third argument is the Period column where you want to find the Period values that overlap.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You must include a numeric output column to contain the sum result value. The data type of this column should be the same data type as the corresponding input column. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, charges, duration) AS
   (SELECT flight_id, charges, duration FROM FlightExp)
SELECT *
FROM TABLE (TD_SYSFNLIB.TD_SUM_NORMALIZE_OVERLAP(NEW VARIANT_TYPE(subtbl.flight_id),
                                                 NEW VARIANT_TYPE(subtbl.charges),
                                                 subtbl.duration)
RETURNS (flight_id INT, charges FLOAT,
         duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SUM_NORMALIZE_MEET

Purpose

Finds the sum of a column for all the rows that were normalized because their Period values meet.

Syntax

```
┌──────────────────────────┐
│                          │
└─ TD_SYSFNLIB. ───────────┘─ TD_SUM_NORMALIZE_MEET ── (grouping_column_list, numeric_column, period_column) ──────────
```

1101A743

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
numeric_column	a numeric column on which SUM() is requested. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_SUM_NORMALIZE_MEET is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_SUM_NORMALIZE_MEET is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is a numeric column on which SUM() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.

- The third argument is the Period column where you want to find the Period values that meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You must include a numeric output column to contain the sum result value. The data type of this column should be the same data type as the corresponding input column. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, charges, duration) AS
   (SELECT flight_id, charges, duration FROM FlightExp)
SELECT *
FROM TABLE (TD_SYSFNLIB.TD_SUM_NORMALIZE_MEET(NEW VARIANT_TYPE(subtbl.flight_id),
                                              NEW VARIANT_TYPE(subtbl.charges),
                                              subtbl.duration)
RETURNS (flight_id INT, charges FLOAT,
         duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SUM_NORMALIZE_OVERLAP_MEET

Purpose

Finds the sum of a column for all the rows that were normalized because their Period values either overlap or meet.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘─ TD_SUM_NORMALIZE_OVERLAP_MEET ── (grouping_column_list, numeric_column, period_column) ──
```

1101A744

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
numeric_column	a numeric column on which SUM() is requested. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_SUM_NORMALIZE_OVERLAP_MEET is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_SUM_NORMALIZE_OVERLAP_MEET is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is a numeric column on which SUM() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.

- The third argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- You must specify the output columns to be the same as the columns specified in the input arguments, including the Period column.
- You must specify the output columns with the same data types and in the same order as the corresponding input columns.
- You must include a numeric output column to contain the sum result value. The data type of this column should be the same data type as the corresponding input column. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, charges, duration) AS
   (SELECT flight_id, charges, duration FROM FlightExp)
SELECT * FROM TABLE (
   TD_SYSFNLIB.TD_SUM_NORMALIZE_OVERLAP_MEET(NEW VARIANT_TYPE(subtbl.flight_id),
                                             NEW VARIANT_TYPE(subtbl.charges),
                                                              subtbl.duration)
RETURNS (flight_id INT, charges FLOAT,
         duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SEQUENCED_SUM

Purpose

Finds the sum of a column for all adjacent periods in normalized rows whose Period values either meet or overlap.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘─ TD_SEQUENCED_SUM ── (grouping_column_list, numeric_column, period_column) ──
```

1101A745

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
numeric_column	a numeric column on which SUM() is requested. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_SEQUENCED_SUM is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_SEQUENCED_SUM is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is a numeric column on which SUM() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.

- The third argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- The output columns must include all of the grouping columns with the same data type and in the same order as the input columns.
- You must include a numeric output column to contain the sum result value. The data type of this column should be the same data type as the corresponding input column. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.
- A Period column with the same Period data type as the input Period column.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, charges, duration) AS
   (SELECT flight_id, charges, duration FROM FlightExp)
SELECT * FROM  TABLE (
   TD_SYSFNLIB.TD_SEQUENCED_SUM(NEW VARIANT_TYPE(subtbl.flight_id),
                                NEW VARIANT_TYPE(subtbl.charges),
                                                 subtbl.duration)
RETURNS (flight_id INT, charges FLOAT,
         duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SEQUENCED_AVG

Purpose

Finds the average of a column for all adjacent periods in normalized rows whose Period values either meet or overlap.

Syntax

```
┬──────────────────┬ TD_SEQUENCED_AVG ── (grouping_column_list, numeric_column, period_column) ──────────────
└─ TD_SYSFNLIB. ─┘
```

1101A746

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
numeric_column	a numeric column on which AVG() is requested. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_SEQUENCED_AVG is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_SEQUENCED_AVG is a table function that takes three arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is a numeric column on which AVG() is requested. All numeric data types are supported. You must specify this argument as a dynamic UDT where the column is an attribute of the UDT.

- The third argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- The output columns must include all of the grouping columns with the same data type and in the same order as the input columns.
- You must include a numeric output column to contain the average result value. The data type of this column can be FLOAT or the same data type as the corresponding input column; however, to avoid possible rounding of the result value, it is recommended that you use FLOAT. To prevent a possible overflow error, you can use the CAST function to convert the data type of the input column to a larger numeric data type.
- A Period column with the same Period data type as the input Period column.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, charges, duration) AS
   (SELECT flight_id, charges, duration FROM FlightExp)
SELECT * FROM  TABLE (
   TD_SYSFNLIB.TD_SEQUENCED_AVG(NEW VARIANT_TYPE(subtbl.flight_id),
                                NEW VARIANT_TYPE(subtbl.charges),
                                                 subtbl.duration)
RETURNS (flight_id INT, charges FLOAT,
         duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, charges, duration) ORDER BY 1,3;
```

TD_SEQUENCED_COUNT

Purpose

Finds the count of a column for all adjacent periods in normalized rows whose Period values either meet or overlap.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘─ TD_SEQUENCED_COUNT ── (grouping_column_list, period_column) ──────────────────────
```

1101A747

where:

Syntax element...	Specifies...
grouping_column_list	one or more grouping columns, not including the Period column. You must specify the input as a dynamic UDT.
period_column	a column with a data type of PERIOD(DATE), PERIOD(TIMESTAMP), or PERIOD(TIMESTAMP WITH TIME ZONE).

Invocation

TD_SEQUENCED_COUNT is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Usage Notes

TD_SEQUENCED_COUNT is a table function that takes two arguments. The arguments passed to the function are the specified columns in a subtable derived from using the WITH Request Modifier as follows:

- The first argument is one or more grouping columns, not including the Period column. You must specify this argument as a dynamic UDT, where each column is an attribute of the UDT. For more information, see "NEW VARIANT_TYPE" on page 1321.
- The second argument is the Period column where you want to find the Period values that overlap or meet.

Input to the table function must be columns that are hash-redistributed on the grouping columns and sorted by the grouping columns and the Period values as follows:

- You must specify a LOCAL ORDER BY clause that includes all of the grouping columns and the Period column in the same order that was specified in the input arguments. The sort order must be ascending.
- You must include a HASH BY clause with at least one of the grouping columns. The HASH BY clause cannot include the Period column or any columns that are not part of the grouping columns.

You must invoke the function with a RETURNS clause that specifies the output columns as follows:

- The output columns must include all of the grouping columns with the same data type and in the same order as the input columns.
- You must include an INTEGER output column to contain the count result.
- A Period column with the same Period data type as the input Period column.

Result

This function returns result rows with the columns specified in the RETURNS clause as follows:

- The grouping columns specified in the input argument.
- The Period column with normalized Period values.
- An optional INTEGER column containing the count of the rows that were normalized because their Period values meet.

Example

```
WITH subtbl(flight_id, duration) AS
   (SELECT flight_id, duration FROM FlightExp)
SELECT * FROM  TABLE (
   TD_SYSFNLIB.TD_SEQUENCED_COUNT(NEW VARIANT_TYPE(subtbl.flight_id),
                                                  subtbl.duration)
RETURNS (flight_id INT, cnt INT,
         duration PERIOD(TIMESTAMP(6) WITH TIME ZONE))
HASH BY flight_id     /* input data is redistributed on column, flight_id */
LOCAL ORDER BY flight_id, duration)     /* input data is sorted on these columns */
AS DT(flight_id, cnt, duration) ORDER BY 1,3;
```

CHAPTER 23 Regular Expression Functions

This chapter describes regular expression functions.

REGEXP_SUBSTR

Purpose

Extracts a substring from *source_string* that matches a regular expression specified by *regexp_string*.

REGEXP_SUBSTR supports 2, 3, 4, or 5 parameters.

Syntax

```
┌──────────────────────┐
│                      │
└── TD_SYSFNLIB. ──────┘── REGEXP_SUBSTR ── ( source_string , regexp_string , position_arg ──── (A)

(A) ── , occurrence_arg , match_arg ) ──────────────────────────────────────────────────────
```

1145-051A

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character argument. If *source_string* is NULL, NULL is returned.
regexp_string	a character argument. If *regexp_string* is NULL, NULL is returned.
position_arg	a numeric argument. *position_arg* specifies the position in *source_string* from which to start searching (default is 1). If *position_arg* is greater than the input string length, NULL is returned. If *position_arg* is NULL, NULL is returned.
occurrence_arg	a numeric argument. *occurrence_arg* specifies the number of the occurrence to be returned (default is 1). For example, if *occurrence_arg* is 2, the function matches the first occurrence in *source_string* and starts searching from the character following the first occurrence in *source_string* for the second occurrence in *source_string*. If *occurrence_arg* is greater than the number of matches found, NULL is returned. If *occurrence_arg* is NULL, a NULL result is returned. If *occurrence_arg* is omitted, the default value (1) is used.

Syntax element ...	Specifies ...
match_arg	a character argument. Valid values are: 'i' = case-insensitive matching. 'c' = case sensitive matching. 'n' = the period character (match any character) can match the newline character. 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option, the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*. 'l' = if *source_string* exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail. If *match_arg* is not specified: • The match is case sensitive. • A period does not match the newline character. • *source_string* is treated as a single line. If *match_arg* is NULL, an error is returned. If *match_arg* is not valid, it is ignored.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

REGEXP_SUBSTR is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR, or CLOB
- *regexp_string* = CHAR or VARCHAR (maximum size of 512 bytes)
- *position_arg* = NUMBER
- *occurrence_arg* = NUMBER
- *match_arg* = VARCHAR

source_string parameters that are CLOBs can be a maximum of 16 MB.

REGEXP_SUBSTR returns an error if this size is exceeded unless *match_arg* = 'l', in which case, it returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

REGEXP_SUBSTR is a scalar function whose return value data type depends on the data type associated with *source_string* input parameter that is passed into the function.

A *source_string* of:

- CHAR, VARCHAR returns VARCHAR in the same character set as *source_string*.
- CLOB returns CLOB in the same character set as *source_string*.

Example 1

The following query:

```
SELECT REGEXP_SUBSTR('mint chocolate chip', 'ch(i|o)p', 1, 1, 'c');
```

returns 'chip'.

Example 2

The following query:

```
SELECT REGEXP_SUBSTR('mint chocolate chip chop', ' ch(i|o)p', 1, 2,
'i');
```

returns 'chop' because it is the second occurrence of the match.

REGEXP_REPLACE

Purpose

Replaces portions of *source_string* that match *regexp_string* with *replace_string*.

REGEXP_REPLACE supports 3, 4, or 6 parameters.

Syntax

```
┬──────────────────┬─ REGEXP_REPLACE ─ ( source_string, regexp_string, replace_string, ──Ⓐ
└─ TD_SYSFNLIB. ───┘

Ⓐ── position_arg, occurrence_arg, match_arg ) ──────────────────────────────────────
```

1101A923

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character argument. If *source_string* is NULL, NULL is returned.
regexp_string	a character argument. If *regexp_string* is NULL, NULL is returned.
replace_string	a character argument. If *replace_string* is not specified, is NULL or is an empty string, the matches are removed from the result.
position_arg	a numeric argument. *position_arg* specifies the position in *source_string* from which to start searching (default is 1). If *position_arg* is greater than the input string length, nothing is replaced and *source_string* is returned. If *position_arg* is NULL, NULL is returned.

Syntax element ...	Specifies ...
occurrence_arg	a numeric argument. *occurrence_arg* specifies the occurrence to replace the match with *replace_string*. • If a value of 0 is specified, all occurrences are replaced. • If the value is not 0, *occurrence_arg* searches for the *n*th occurrence beginning from *position_arg* of *regexp_string*. If *occurrence_arg* is greater than the number of matches found, nothing is replaced and *source_string* is returned. If *occurrence_arg* is NULL, a NULL result is returned. If *occurrence_arg* is omitted, the default value (1) is used. This is different from the Oracle implementation of REGEXP_REPLACE, where if *occurrence_arg* is omitted, the default value (0) is used.
match_arg	a character argument. Valid values are: 'i' = case-insensitive matching. 'c' = case sensitive matching. 'n' = the period character (match any character) can match the newline character. 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option, the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*. 'l' = if *source_string* exceeds the current maximum allowed source string size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail. If *match_arg* is not specified: • The match is case sensitive. • A period does not match the newline character. • *source_string* is treated as a single line. If *match_arg* is NULL, an error is returned. If *match_arg* is not valid, it is ignored.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

REGEXP_REPLACE is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR, or CLOB
- *regexp_string* = CHAR or VARCHAR (maximum size of 512 bytes)
- *replace_string* = CHAR, VARCHAR CLOB (truncated to 32 KB)
- *position_arg* = NUMBER
- *occurrence_arg* = NUMBER
- *match_arg* = VARCHAR

source_string parameters that are CLOBs can be a maximum of 16 MB. The function returns an error if this size is exceeded unless *match_arg* = 'l', in which case, it returned NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

REGEXP_REPLACE is a scalar function whose return value data type depends on the data type associated with *source_string* input parameter that is passed into the function.

A *source_string* of:

- CHAR, VARCHAR returns VARCHAR in the same character set as *source_string*.
- CLOB returns CLOB in the same character set as *source_string*.

Example 1

The following query:

```
SELECT REGEXP_REPLACE('Hello World World', '(world)$', 'My', 1, 1, 'i');
```

returns the result "Hello World My".

Example 2

The following query:

```
SELECT REGEXP_REPLACE('Friday is the best day of the week.', 'of the
week', 'EVER', 1, 1, 'c');
```

returns the result 'Friday is the best day EVER'.

Example 3

The following query:

```
SELECT REGEXP_REPLACE('Hello Santa says ho ho', 'ho', 'HO!', 1, 2, 'c');
```

returns the result 'Hello Santa says ho HO!'.

REGEXP_INSTR

Purpose

Searches *source_string* for a match to *regexp_string*.

REGEXP_INSTR supports 2 or 6 parameters.

Syntax

```
┬──────────────────┬─ REGEXP_INSTR ── ( source_string, regexp_string, position_arg, ──Ⓐ
└─ TD_SYSFNLIB. ─┘

Ⓐ── occurrence_arg, return_opt, match_arg ) ──────────────────────────────────────
```

1145-061A

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character argument. If *source_string* is NULL, NULL is returned.
regexp_string	a character argument. If *regexp_string* is NULL, NULL is returned.
position_arg	a numeric argument. *position_arg* specifies the position in *source_string* from which to start searching (default is 1). If *position_arg* is greater than the input string length, NULL is returned. If *position_arg* is NULL, NULL is returned.
occurrence_arg	a numeric argument. *occurrence_arg* specifies the number of the occurrence to be returned (default is 1). For example, if *occurrence_arg* is 2, the function matches the first occurrence in *source_string* and starts searching from the character following the first occurrence in *source_string* for the second occurrence in *source_string*. If *occurrence_arg* is greater than the number of matches found, zero is returned. If *occurrence_arg* is NULL, a NULL result is returned. If *occurrence_arg* is omitted, the default value (1) is used.

Syntax element ...	Specifies ...
return_opt	a numeric argument. Valid values are: 0 = function returns the beginning position of the match (default). 1 = function returns the end position (character following the occurrence) of the match If *return_opt* is not valid, an error is returned. If *return_opt* is NULL, NULL is returned.
match_arg	a character argument. Valid values are: 'i' = case-insensitive matching. 'c' = case sensitive matching. 'n' = the period character (match any character) can match the newline character. 'm' = *source string* is treated as multiple lines instead of as a single line. With this option the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*. 'l' = if *source_string* exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail. If *match_arg* is not specified: • The match is case sensitive. • A period does not match the newline character. • *source_string* is treated as a single line. If there is no match, zero is returned. If *match_arg* is not valid or omitted, an error is returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

REGEXP_INSTR is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string*= CHAR, VARCHAR, or CLOB
- *regexp_string* = CHAR or VARCHAR (maximum size of 512 bytes)

- *position_arg* = NUMBER
- *occurrence_arg* = NUMBER
- *return_opt* = NUMBER
- *match_arg* = VARCHAR

source_string parameters that are CLOBs can be a maximum of 16 MB. The function returns an error if this size is exceeded unless *match_arg* = 'l' is specified. If this is specified, a NULL is returned instead of an error.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

REGEXP_INSTR is a scalar function whose return value data type is:

- INTEGER if source_string is a CHAR or VARCHAR
- BIGINT if source_string is a CLOB

Example 1

The following query:

```
SELECT REGEXP_INSTR('Hello Santa says ho ho','Hello Santa says ho ho', 1,
1, 1, 'c');
```

returns the result 23.

Example 2

The following query:

```
SELECT REGEXP_INSTR('Hello Santa says ho ho', 'Hello Santa says ho ho',
1, 1, 0, 'c');
```

returns the result 1.

REGEXP_SIMILAR

Purpose

Compares *source_string* to *regexp_string* and returns integer value:

REGEXP_SIMILAR provides semantics comparable to the SIMILAR predicate in ANSI-SQL-99.

REGEXP_SIMILAR supports 3 parameters.

Syntax

```
┬──────────────┬─ REGEXP_SIMILAR ─── ( source_string, regexp_string, match_arg ) ───
└─ TD_SYSFNLIB. ┘
```

1101A925

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character argument. If *source_string* is NULL, NULL is returned.
regexp_string	a character argument. If *regexp_string* is NULL, NULL is returned.

Syntax element ...	Specifies ...
match_arg	a character argument. Valid values are: 'i' = case-insensitive matching. 'c' = case sensitive matching. 'n' = the period character (match any character) can match the newline character. 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*. 'l' = if *source_string* exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail. If *match_arg* is not specified, the default is as follows: • The match is case sensitive. • A period does not match the newline character. • *source_string* is treated as a single line. If there is no match, NULL is returned. If *match_arg* is not valid, an error is returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

REGEXP_SIMILAR is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string*= CHAR, VARCHAR, CLOB
- *regexp_string* = CHAR, VARCHAR (maximum size of 512 bytes)
- *match_arg* = VARCHAR

source_string parameters that are CLOBs can be a maximum of 16 MB. The function returns an error if this size is exceeded unless *match_arg* = 'l' is specified. If this is specified, a NULL is returned instead of an error.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

REGEXP_SIMILAR is a scalar function whose return value data type in an integer value:

- 1 (true) if the entire string matches *regexp_arg*
- 0 (false) if the entire string does not match *regexp_arg*

Example

The following query:

```
SELECT name FROM customers WHERE REGEXP_SIMILAR(name, '(Mike B(i|y)rd)|
(Michael B(i|y)rd)', 'c') = 1;
```

returns the names from the customers table that match:

- 'Mike Bird'
- 'Mike Byrd'
- 'Michael Bird'
- 'Michael Byrd'

The matching is case sensitive.

REGEXP_SPLIT_TO_TABLE

Purpose

Splits *source_string* into a table of strings using *regexp_string* as the delimiter.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── REGEXP_SPLIT_TO_TABLE ── ( source_string, regexp_string, match_arg ) ──
```

1101A926

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character argument. If *source_string* is NULL, NULL is returned.
regexp_string	a character argument. If *regexp_string* is NULL, NULL is returned.
match_arg	a character argument. Valid values are: 'i' = case-insensitive matching. 'c' = case sensitive matching. 'n' = the period character (match any character) can match the newline character. 'm' = *source_string* is treated as multiple lines instead of as a single line. With this option the '^' and '$' characters apply to each line in *source_string* instead of the entire *source_string*. 'l' = if *source_string* exceeds the current maximum allowed *source_string* size (currently 16 MB), a NULL is returned instead of an error. This is useful for long-running queries where you do not want long strings causing an error that would make the query fail. If *match_arg* is not specified: • The match is case sensitive. • A period does not match the newline character. • *source_string* is treated as a single line. If there is no match, NULL is returned. If *match_arg* is not valid, an error is returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

REGEXP_SPLIT_TO_TABLE is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR
- *regexp_str*ing = CHAR, VARCHAR (maximum size of 512 bytes)
- *match_arg* = VARCHAR

REGEXP_SPLIT_TO_TABLE does not support CLOB for the *source_string* type.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

REGEXP_SPLIT_TO_TABLE is a table function that returns a table with a single column of type VARCHAR.

The character set of the return value is the same as the character set of *source_str*.

Example

If you create the following table:

```
CREATE TABLE split_table_latin(id integer, src varchar(100) character
set latin, pattern varchar(100) character set latin, match varchar(100)
character set latin);
```

and insert the following values:

```
INSERT into split_table_latin(3,
'The2134quick234brown234fox987jumps8743over342the03487lazy089734red97238
4dog2343.', '[0-9]*','c');
```

Then the following SELECT statement:

```
SELECT * from table(
regexp_split_to_table(split_table_latin.src,split_table_latin.pattern,sp
lit_table_latin.match)
returns (res varchar(100) character set latin))  as t1;
```

returns a table with the following rows:

- The
- quick
- brown
- fox
- jumps
- over
- the
- lazy
- red
- dog

CHAPTER 24 Set Operators

This chapter describes SQL set operators.

Overview of Set Operators

The SQL set operators manipulate the results sets of two or more queries by combining the results of each individual query into a single results set.

Teradata SQL Set Operators

Teradata SQL supports the following set operators:

Set Operator	Function
INTERSECT	Returns result rows that appear in all answer sets generated by the individual SELECT statements.
MINUS / EXCEPT	Result is those rows returned by the first SELECT except for those also selected by the second SELECT. MINUS is the same as EXCEPT.
UNION	Combines the results of two or more SELECT statements.

Set operators appear in query expressions. A query expression is a set of queries combined by the set operators INTERSECT, MINUS/EXCEPT, and UNION.

Syntax for *query_term*

```
┬─ SELECT ── statement ─┬
└─ (query_expression ) ─┘
```

FF07D178

Syntax for *query_factor*

```
┬─ query_term ──────────────────────────────────┬
└─ query_factor ── INTERSECT ┬──────┬── query_term ─┘
                             └─ ALL ─┘
```

HH01A061

Syntax for *query_expression*

where:

Syntax Element ...	Specifies ...
query_term	
SELECT *statement*	a SELECT statement. For details, see *SQL Data Manipulation Language*.
query_expression	an optional expression that might or might not include set operators, other expressions, and an ORDER BY clause.
query_factor	
INTERSECT	a set operator returning the result rows appearing in all answer sets.
ALL	an optional keyword, allowing duplicate rows to be returned.
query_expression	
UNION	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.
MINUS	
EXCEPT	
ALL	an optional keyword, allowing duplicate rows to be returned.
ORDER BY	the ORDER BY clause to order the result rows returned. For details, see *SQL Data Manipulation Language*.
expression	an expression used in the ORDER BY clause to determine the sort order of returned rows in the result.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.

Syntax Element ...	Specifies ...
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
NULLS FIRST	that NULL results are to be listed first.
NULLS LAST	that NULL results are to be listed last.

ANSI Compliance

INTERSECT, EXCEPT, and UNION are ANSI SQL:2011 compliant.

MINUS and the ALL option are Teradata extensions to the ANSI standard.

Rules for Set Operators

Duplicate Rows

By default, duplicate rows are not returned.

To permit duplicate rows to be returned, specify the ALL option. For an example, see "Retaining Duplicate Rows Using the ALL Option" on page 1129.

Operations That Support Set Operators

You can use set operators within the following operations:

- Simple queries
- Derived tables

  Note: You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.
- Subqueries
- INSERT … SELECT clauses
- View definitions

SELECT statements connected by set operators can include all of the normal clause options for SELECT except the WITH clause.

SELECT AND CONSUME Statement

Set operations do not operate on SELECT AND CONSUME statements.

Support for ORDER BY Clause

A query expression can include only one ORDER BY specification, at the end.

Restrictions on the Data Types Involved in Set Operations

The following restrictions apply to CLOB, BLOB, and UDT types involved in set operations:

Data Type	Restrictions
BLOB	You cannot use set operators with CLOB or BLOB types.
CLOB	
UDT	• Multiple UDTs involved in set operations must be identical types because Teradata Database does not perform implicit type conversion on UDTs involved in set operations. A workaround for this restriction is to use CREATE CAST to define casts that cast between the UDTs and then explicitly invoke the CAST function within the set operation. • UDTs involved in set operations must have ordering definitions. Teradata Database generates ordering functionality for distinct UDTs where the source types are not LOBs. To create an ordering definition for structured UDTs or distinct UDTs where the source types are LOBs, or to replace system-generated ordering functionality, use CREATE ORDERING. For more information on CREATE CAST and CREATE ORDERING, see *SQL Data Definition Language*.

Precedence of Set Operators

The precedence for processing set operators is as follows:

1 INTERSECT

2 UNION and MINUS/EXCEPT

The set operators evaluate from left to right if no parentheses explicitly specify another order.

Example

For example, consider the following query.

```
SELECT statement_1
UNION
SELECT statement_2
EXCEPT
SELECT statement_3
INTERSECT
SELECT statement_4;
```

The operations are performed in the following order:

1 Intersect the results of statement_3 and statement_4.

2 Union the results of statement_1 and statement_2.

3 Subtract the intersected rows from the union.

Using Parentheses to Customize Precedence

To override precedence, use parentheses. Operations in parentheses are performed first.

For example, consider the following form:

```
( ( SELECT statement_1
    UNION
    SELECT statement_2 )
EXCEPT
  ( SELECT statement_3
    UNION
    SELECT statement_4 )
)
EXCEPT
SELECT statement_5
INTERSECT
SELECT statement_6;
```

The following list explains the precedence of operators for this example.

1. UNION SELECT `statement_1` and SELECT `statement_2`.
2. UNION SELECT `statement_3` and SELECT `statement_4`.
3. Subtract the result of the second UNION from the result of the first UNION.
4. INTERSECT SELECT `statement_5` and SELECT `statement_6`.
5. Subtract the INTERSECT result from the remainder of the UNION operations.

Retaining Duplicate Rows Using the ALL Option

Unless you specify the ALL option, duplicate rows are eliminated from the final result. The ALL option retains duplicate rows for the result set to which it is applied.

Example

The following query returns duplicate rows for each result set, including the final:

```
SELECT statement_1
UNION ALL
SELECT statement_2
MINUS ALL
SELECT statement_3
INTERSECT ALL
SELECT statement_4
```

Attributes of a Set Result

The data type, title, and format clauses contained in the first SELECT statement determine the data type, title, and format information that appear in the final result.

Attributes for all other SELECT statements in the query are ignored.

Example 1

```
SELECT level, param, 'GMKSA' (TITLE 'OWNER')
FROM gmksa
WHERE cycle = '03'
UNION
SELECT level, param, 'GMKSA CONTROL'
FROM gmksa_control
WHERE cycle = '03'
ORDER BY 1, 2;
```

The query returns the following results set:

```
***QUERY COMPLETED. 5 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL  PARAM  OWNER
-----  -----  -----
00     A      GMKSA
00     T      GMKSA
85     X      GMKSA
SF     A      GMKSA
SF     T      GMKSA
```

The first SELECT specifies GMKSA, which is CHAR(5)—that data type is then forced on the second SELECT. As a result, GMKSA_CONTROL entries are dropped because the first five characters are the same.

Because this query does not specify the ALL option, duplicate rows are dropped.

Example 2

In the next query, the SELECT order is reversed:

```
SELECT level, param 'GMKSA CONTROL' (TITLE 'OWNER')
FROM gmksa_control
WHERE cycle = '03'
UNION
SELECT level, param, 'GMKSA'
FROM gmksa
WHERE cycle = '03'
ORDER BY 1, 2;
```

This query returns the following answer set:

```
***QUERY COMPLETED.10 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL   PARAM     OWNER
-----   -----     -------------
00      A         GMKSA
00      A         GMKSA CONTROL
00      T         GMKSA
00      T         GMKSA CONTROL
85      X         GMKSA
85      X         GMKSA CONTROL
SF      A         GMKSA
SF      A         GMKSA CONTROL
SF      T         GMKSA
SF      T         GMKSA CONTROL
```

In this case, because the first SELECT specified 'GMKSA CONTROL', the rows were not duplicates and were included in the answer set.

Example 3

This example demonstrates how a poorly formed query can cause truncation of the results.

```
SELECT level, param, 'GMKSA       ' (TITLE 'OWNER')
FROM gmksa
WHERE cycle = '03'
UNION
SELECT level, param,'GMKSA CONTROL'
FROM gmksa_control
WHERE cycle = '03'
ORDER BY 1, 2;
```

This query returns the following answer set:

```
***QUERY COMPLETED.10 ROWS FOUND. 3 COLUMNS RETURNED.
LEVEL   PARAM   OWNER
-----   -----   ------------
00      A       GMKSA
00      A       GMKSA CONTRO
00      T       GMKSA
00      T       GMKSA CONTRO
85      X       GMKSA
85      X       GMKSA CONTRO
SF      A       GMKSA
SF      A       GMKSA CONTRO
SF      T       GMKSA
SF      T       GMKSA CONTRO
```

This query returned the expected rows; note, however, that because of the way the name was specified in the first SELECT, there was some truncation.

Example of How the Character Set is Determined for the Query

The character set of the expression in the first SELECT statement determines the character set of the entire query. In the following example, the character set of <*char_expression_1*> is used as the character set for the entire query:

```
SELECT <char_expression_1> FROM <table_1>
UNION
SELECT <char_expression_2> FROM <table_2>;
```

If <*char_expression_2*> contains characters not included in the character set of <*char_expression_1*>, an error can result.

For example, if <*char_expression_1*> is CHARACTER SET LATIN, but <*char_expression_2*> is CHARACTER SET UNICODE, attempting to translate <*char_expression_2*> from UNICODE to LATIN results in an error if multi-byte characters are present in <*char_expression_2*>.

Set Operators With Derived Tables

Derived tables support set operators, as demonstrated in the following example:

Example

```
SELECT x1
FROM table_1,
(SELECT x2
FROM table_2
UNION
SELECT x3
FROM table_3
) derived_table;

SELECT x1,y1
FROM table_1,
(SELECT *
FROM table_2) derived_table(column_1, column_2)
WHERE column_2 = 1 ;
```

Restrictions

You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators. The following example returns an error.

Example

The following table function "add2int" takes two integers as input and returns the two integers and their summation.

```
CREATE TABLE t1 (a1 INTEGER, b1 INTEGER);
CREATE TABLE t2 (a2 INTEGER, b2 INTEGER);

REPLACE FUNCTION add2int
   (a INTEGER,
    b INTEGER)
RETURNS TABLE
   (addend1 INTEGER,
    addend2 INTEGER,
    mysum INTEGER)
SPECIFIC add2int
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'CS!add3int!add2int.c';

/* Query Q1 */
WITH dt(a1, b1) AS
( SELECT a1, b1
  FROM t1
  UNION ALL
  SELECT a2, b2
  FROM t2
)
SELECT *
FROM TABLE (add2int(dt.a1, dt.b1)
HASH BY b1
LOCAL ORDER BY b1) tf;
```

Set Operators in Subqueries

Set operators are permitted in subqueries. The following examples demonstrate their correct use.

Example 1

```
SELECT x1
FROM table_1
WHERE (x1,y1) IN
(SELECT * FROM table_2
UNION
SELECT * FROM table_3);
```

Example 2

```
SELECT *
FROM table_1
WHERE table_1.x1 IN
(SELECT x2
FROM table_2
UNION
(SELECT x3
FROM table_3
UNION
SELECT x4
FROM table_4));
```

Example 3

```
SELECT *
FROM table_1
WHERE x1 IN
(SELECT SUM(x2)
FROM table_2
UNION
SELECT x3
FROM table_3);
```

Example 4

```
SELECT *
FROM table_1
WHERE x1 IN
(SELECT MAX(x2)
FROM table_2
UNION
SELECT MIN(x3)
FROM table_3);
```

Example 5

```
SELECT *
FROM table_1
WHERE X1 IN
```

```
(SELECT x2 FROM table_2
UNION
SELECT x3 FROM table_3
UNION
SELECT x4 FROM table_4);
```

Example 6

```
SELECT x1
FROM table_1
WHERE x1 IN ANY
(SELECT x2 FROM table_2
INTERSECT
SELECT x3 FROM table_3
MINUS
SELECT x4 FROM table_4);
```

Example 7

```
UPDATE table_1
SET x1=1
WHERE table_1.x1 IN
(SELECT x2
FROM table_2
UNION
SELECT x3
FROM table_3
UNION
SELECT x4
FROM table_4);
```

Set Operators in INSERT … SELECT Statements

Set operators are permitted in INSERT … SELECT statements. The following examples demonstrate their correct use.

Example 1

The first example demonstrates a simple INSERT … SELECT using set operators.

```
INSERT table1 (x1,y1)
SELECT *
FROM table_2
UNION
SELECT x3,y3
FROM table_3;
```

Example 2

The second example demonstrates an INSERT … SELECT from a view that uses set operators.

```
REPLACE VIEW v AS
SELECT *
FROM table_1
UNION
```

```
SELECT *
FROM table_2;

INSERT table_3(x3,y3)
SELECT *
FROM v;
```

Example 3

This example demonstrates an INSERT … SELECT from a derived table with set operators.

```
INSERT table_1
SELECT *
FROM
(SELECT x2,y2
FROM table_2
UNION
SELECT *
FROM table_3 DerivedTable
);
```

Set Operators in View Definitions

Set operators are permitted within view definitions.

For example, the following REPLACE VIEW statement uses UNION within a view definition:

```
REPLACE VIEW view_1 AS
   SELECT x1,y1
   FROM table_1
   UNION
   SELECT x2,y2
   FROM table_2;
```

Support for the GROUP BY Clause

GROUP BY can be used within views with set operators. For details, see "GROUP BY and ORDER BY Clauses" on page 1138.

Restrictions

The following limitations apply to view definitions that specify set operators:

- UPDATE, DELETE, and INSERT are not applicable. The following example does *not* work:

  ```
  REPLACE VIEW V AS
  SELECT X
  FROM TABLE_1
  UNION
  SELECT Y FROM
  TABLE_1;

  UPDATE V
  SET X=0;
  ```

  An attempt to perform this sequence of statements produces the following error message:

  ```
  ***Failure 3823 VIEW 'v' may not be used for Help Index/
  Constraint/Statistics, Update, Delete or Insert.
  ```

- WITH CHECK OPTION is not applicable. The following example does *not* work:

  ```
  REPLACE VIEW ERRV( c ) AS
  SELECT *
  FROM TABLE_1
  UNION
  SELECT *
  FROM TABLE_2
  WHERE TABLE_2.X=2 WITH CHECK OPTION;
  ```

  An attempt to perform this statement causes the following error message:

  ```
  ***Failure 3847 Illegal use of a WITH clause.
  ```

- Column level privileges cannot be granted. The following example does *not* work:

  ```
  GRANT UPDATE ( c ) ON TABLE_VIEW TO USER_NAME;
  ```

  An attempt to perform this statement causes the following error message:

  ```
  ***Failure 3499: GRANT cannot be used on views with set operators.
  ```

- A view definition that uses set operators cannot specify an ORDER BY clause, but a SELECT statement applied on the view can use ORDER BY. For details, see "GROUP BY and ORDER BY Clauses" on page 1138.

Examples

The following examples provide correct uses of set operators within view definitions.

Example 1

```
REPLACE VIEW v AS
   SELECT x1
   FROM TABLE_1
   UNION
   SELECT x2
   FROM TABLE_2
   UNION
```

```
   SELECT x3
   FROM TABLE_3;

SELECT * FROM v;
```

Example 2

```
REPLACE VIEW view_2 AS
   SELECT *
   FROM view_1
   UNION
   SELECT *
   FROM table_3
   UNION
   SELECT *
   FROM table_4;

SELECT *
FROM view_2
ORDER BY 1,2;
```

Example 3

```
REPLACE VIEW v AS
   SELECT x1
   FROM table_1
   WHERE x1 IN
    (SELECT x2
     FROM table_2
     UNION
     SELECT x3
     FROM table_3
    );

SELECT * FROM v;
```

Queries Connected by Set Operators

Certain rules and restrictions apply to SELECT statements connected by set operators that might not apply elsewhere.

Number of Expressions in SELECT Statements

All SELECT statements must have the same number of expressions.

If the first SELECT statement contains three expressions, all succeeding SELECT statements must contain three expressions.

You can use a null expression in a SELECT statement as a place holder for a missing expression.

In the following example, the second expression is null.

```
SELECT EmpNo, NULL (CHAR(5))
FROM Employee;
```

WITH Clause

WITH clauses cannot be used in SELECT statements connected by set operators.

GROUP BY and ORDER BY Clauses

GROUP BY clauses are allowed in individual SELECT statements of a query expression but apply only to that SELECT statement and not to the result set.

ORDER BY clauses are allowed only in the last SELECT statement of a query expression and specify the order of the result set.

ORDER BY clauses can contain only numeric literals.

For example, to order by the first column in your result set, specify ORDER BY 1.

View definitions with set operators can use GROUP BY but cannot use ORDER BY. A SELECT statement applied to a view definition with set operators can use GROUP BY and ORDER BY. The following examples are correct uses of these operations within a view definition:

```
REPLACE VIEW v AS
SELECT x1,y1
FROM table1
UNION
SELECT x2,y2
FROM table2;

SELECT *
FROM v
ORDER BY 1;

SELECT SUM(x1), y1
FROM v
GROUP BY 2;
```

You can also apply independent GROUP BY operations to each unioned SELECT. The following example demonstrates how to do this:

```
REPLACE VIEW v(column_1,column_2) AS
SELECT MIN(x1),y1
FROM table_1
GROUP BY 2
UNION ALL
SELECT MIN(x2),y2
FROM table_2
GROUP BY 2
UNION ALL
SELECT x3,y3 FROM table_3;

SELECT SUM(v.column_1) (NAMED sum_c1),column_2
GROUP BY 2
ORDER BY 2;

SELECT *
FROM table_1
```

```
WHERE (x1,y1) IN
(SELECT SUM(x2), y2
FROM table_2
GROUP BY 2
UNION
SELECT SUM(x3), y3
FROM table_3
GROUP BY 2
);
```

Table Name in SELECT Statements

Each SELECT statement must identify the table that the data is to come from even if all SELECT statements reference the same table.

Data Type Compatibility

Corresponding fields in each SELECT statement must have data types that are compatible. For example, if the first field in the first SELECT statement is a character data type, then the first field in each succeeding SELECT statement must be a character data type.

Corresponding numeric types do not have to be the same, but they must be compatible. For example, a field in one SELECT statement can be defined as INTEGER and the corresponding field in another SELECT statement can be defined as SMALLINT.

The data types in the first SELECT statement determine the data types of corresponding columns in the result set.

The following table provides details about data type compatibility.

Data Type	Details
Character	Character types in the first SELECT statement determine the length of character strings in the result set. This can lead to truncation of character strings in the result set if the length of a character type in the first SELECT statement is less than the length of corresponding character types in succeeding SELECT statements. The character set of the expression in the first SELECT statement determines the character set for the entire query. For more information, see "Example of How the Character Set is Determined for the Query" on page 1131.
Numeric	Numeric types in the first SELECT statement determine the size of numeric types in the result set. All corresponding numeric fields in succeeding SELECT statements are converted to the numeric data type in the first SELECT statement. This can lead to a numeric overflow error if the size of a numeric type in the first SELECT statement is smaller than the size of corresponding numeric types in succeeding SELECT statements and the values returned by the succeeding statements do not fit into the smaller data type.

Data Type	Details
TIME TIMESTAMP PERIOD(TIME) PERIOD(TIMESTAMP)	TIME, TIMESTAMP, PERIOD(TIME), and PERIOD(TIMESTAMP) types in the first SELECT statement determine the precision of corresponding columns in the result set. All corresponding fields in succeeding SELECT statements are implicitly converted to the data type in the first SELECT statement. If a corresponding field does not have a time zone and the data type in the first SELECT statement does, the time zone is set to the current session time zone displacement. If the precision of a corresponding field is lower than the precision of the data type in the first SELECT statement, trailing zeros are appended to the fractional digits as needed. If the precision of corresponding fields in succeeding SELECT statements is higher than the precision of the data type in the first SELECT statement, an error is reported.

For examples that show how the length of the character type in the first SELECT statement affects the result set, see "Attributes of a Set Result" on page 1129. For examples that show how the numeric data type in the first SELECT statement affects the result set, see "Example 6: Effect of the Order of SELECT Statements on Data Type" on page 1152.

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 1125.
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 1125.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

The ALL option is a Teradata extension to the ANSI standard.

Rules for INTERSECT

The following rules apply to the use of INTERSECT:

- In addition to using INTERSECT within simple queries, you can use INTERSECT within the following operations:
  - Derived tables

    Note: You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.
  - Subqueries
  - INSERT … SELECT statements
  - View definitions
- Each query connected by INTERSECT is executed to produce a result consisting of a set of rows. The intersection must include the same number of columns from each table in each

SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible.

- INTERSECT cannot be used within the following:
  - SELECT AND CONSUME statements.
  - WITH RECURSIVE clause
  - CREATE RECURSIVE VIEW statements

Attributes of a Set Result

The data type, title, and format clauses contained in the first SELECT statement in the intersection determine the data type, title, and format information that appear in the final result.

Attributes for all other SELECT statements in the query are ignored.

Data Type of Nulls

When you specify an explicit NULL for any intersection operation, its data type is INTEGER. For an example of this principle using the UNION operator, see "Example 5: Effect of Explicit NULLs on Data Type of a UNION" on page 1151.

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

Duplicate Row Handling

Unless the ALL option is used, duplicate rows are eliminated from the final result.

If the ALL option is specified, duplicate rows are retained. The ALL option can be specified for as many INTERSECT operators as are used in a multistatement query.

Example

Assume that two tables contain the following rows:

SPart table

SuppNo	PartNo
100	P2
101	P1
102	P1
103	P2

SLocation table

SuppNo	SuppLoc
100	London
101	London
102	Toronto
103	Tokyo

To then select supplier number (SuppNo) for suppliers located in London (SuppLoc) who supply part number P1 (PartNo), use the following request:

```
SELECT SuppNo FROM SLocation
WHERE SuppLoc = 'London'
INTERSECT
SELECT SuppNo FROM SPart
WHERE PartNo = 'P1';
```

The result of this request is:

```
SuppNo
------
   101
```

MINUS/EXCEPT Operator

Purpose

Returns the results rows that appear in *query_expression_1* and *not* in *query_expression_2*.

Syntax

where:

Syntax element ...	Specifies ...
query_expression_1	a complete SELECT statement whose results table is to be MINUSed with *query_expression_2*.
ALL	that duplicate rows are to be retained for the MINUS operation.
query_expression_2	a complete SELECT statement to be MINUSed from *query_expression_1*.

ANSI Compliance

EXCEPT is ANSI SQL:2011 compliant.

MINUS and the ALL option are Teradata extensions to the ANSI SQL:2011 standard.

Usage Notes

Besides simple queries, MINUS or EXCEPT can be used within the following operations:

- Derived tables

  Note: You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.
- Subqueries
- INSERT … SELECT statements
- View definitions

MINUS and EXCEPT cannot be used within the following operations:

- SELECT AND CONSUME statements.
- WITH RECURSIVE clause
- CREATE RECURSIVE VIEW statements

Each query connected by MINUS or EXCEPT is executed to produce a result consisting of a set of rows. The exception must include the same number of columns from each table in each SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible. All the result sets are then combined into a single result set, which has the data types of the columns specified in the *first* SELECT statement in the exception.

MINUS/EXCEPT and NULL

When you specify an explicit NULL for any exception operation, its data type is INTEGER. For an example of this principle using the UNION operator, see "Example 5: Effect of Explicit NULLs on Data Type of a UNION" on page 1151.

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

Duplicate Rows

Unless the ALL option is used, duplicate rows are eliminated from the final result.

If the ALL option is specified, duplicate rows are retained. The ALL option can be specified for as many MINUS operators as are used in a multistatement query.

UNION Operator

Purpose

Combines two or more SELECT results tables into a single result.

Syntax

where:

Syntax element ...	Specifies ...
query_expression_1	a complete SELECT statement to be unioned with *query_expression_2*. For details, see "Syntax for query_expression" on page 1126.
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 1125.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Valid UNION Operations

Besides simple queries, UNION can be used within the following operations:

- Derived tables

  Note: You cannot use the HASH BY or LOCAL ORDER BY clauses in derived tables with set operators.
- Subqueries
- INSERT … SELECT statements
- Non-recursive CREATE VIEW statements

UNION ALL is the only valid set operator in a WITH RECURSIVE clause or CREATE RECURSIVE VIEW statement that defines a recursive query.

Unsupported Operations

UNION cannot be used within the following:

- SELECT AND CONSUME statements.
- WITH RECURSIVE clause (unless the ALL option is also specified)
- CREATE RECURSIVE VIEW statements (unless the ALL option is also specified)

Description of a UNION Operation

Each query connected by UNION is performed to produce a result consisting of a set of rows. The union must include the same number of columns from each table in each SELECT statement (more formally, they must be of the same degree), and the data types of these columns should be compatible. All the result sets are then combined into a single result set that has the data type of the columns specified in the *first* SELECT statement in the union. For an example, see "Example 6: Effect of the Order of SELECT Statements on Data Type" on page 1152.

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 1151.

On the other hand, column data defined as NULL has neither value nor data type and evaluates like any other null in a scalar expression.

Duplicate Rows

Unless the ALL option is used, duplicate rows are eliminated from each result set and from the final result.

If the ALL option is used, duplicate rows are retained for the applicable result set.

You can specify the ALL option for each UNION operator in the query to retain every occurrence of duplicate rows in the final result.

Unexpected Row Length Errors: Sorting Rows for UNION

Before performing the sort operation used to check for duplicates in some union operations, Teradata Database creates a sort key and appends it to the rows to be sorted. If the length of this temporary data structure exceeds the system limit of 64 KB, the operation fails and returns an error to the requestor. Depending on the situation, the message text is one of the following:.

- A data row is too long.
- Maximum row length exceeded in *database_object_name*.

See *Messages* for explanations of these messages.

Example 1

To select the name, project, and the number of hours spent by employees assigned to project OE1-0001, plus the names of employees not assigned to a project, the following query could be used:

```
SELECT Name, Proj_Id, Hours
FROM Employee,Charges
WHERE Employee.Empno = Charges.Empno
AND Proj_Id IN ('OE1-0001')
UNION
SELECT Name, NULL (CHAR (8)), NULL (DECIMAL (4,2))
FROM Employee
WHERE Empno NOT IN
(SELECT Empno
FROM Charges);
```

This query returns the following rows:

Name	Project Id	Hours
Aguilar J	?	?
Brandle B	?	?
Chin M	?	
Clements D	?	?
Kemper R		
Marston A	?	?
Phan A	?	?
Regan R	?	?
Russell S	?	?
Smith T		
Watson L		
Inglis C	0E1-0001	30.0
Inglis C	0E1-001	30.5
Leidner P	0E1-001	10.5
Leidner P	0E1-001	23.0
Moffit H	0E1-001	12.0
Moffit H	0E1-001	35.5

In this example, null expressions are used in columns 2 and 3 of the second SELECT statement. The null expressions are used as place markers so that both SELECT statements in the query contain the same number of expressions.

Example 2

To determine the department number and names of all employees in departments 500 and 600, the UNION operator could be used as follows:

```
SELECT DeptNo, Name
FROM Employee
WHERE DeptNo = 500
UNION
SELECT DeptNo, Name
FROM Employee
WHERE DeptNo = 600 ;
```

This query returns the following rows:

DeptNo	Name
500	Carter J
500	Inglis C
500	Marston A
500	Omura H
500	Reed C
500	Smith T
500	Watson L
600	Aguilar J
600	Kemper R
600	Newman P
600	Regan R

The same results could have been returned with a simpler query, such as the following:

```
SELECT Name, DeptNo
FROM Employee
WHERE (DeptNo = 500)
OR (DeptNo = 600);
```

The advantage to formulating the query using the UNION operator is that if the DeptNo column is the primary index for the Employee table, then using the UNION operator guarantees that the basic selects are prime key operations. There is no guarantee that a query using the OR operation will make use of the primary index.

Example 3

In addition, the UNION operator is useful if you must merge lists of values taken from two or more tables.

For example, if departments 500 and 600 had their own Employee tables, the following query could be used to select data from two different tables and merge that data into a single list:

```
SELECT Name, DeptNo
FROM Employee_dept_500
UNION
SELECT Name, DeptNo
```

```
FROM Employee_dept_600 ;
```

Example 4

Suppose you want to know the number of man-hours charged by each employee who is working on a project. In addition, suppose you also wanted the result to include the names of employees who are not working on a project.

To do this, you would have to perform a union operation as illustrated in the following example.

```
SELECT Name, Proj_Id, Hours
FROM Employee, Charges
WHERE Employee.EmpNo = Charges.EmpNo
UNION
SELECT Name, Null (CHAR(8)), Null (DECIMAL(4,2)),
FROM Employee
WHERE EmpNo NOT IN
(SELECT EmpNo
FROM Charges
)
UNION
SELECT Null (VARCHAR(12)), Proj_Id, Hours
FROM Charges
WHERE EmpNo NOT IN
(SELECT EmpNo
FROM Employee
);
```

The first portion of the statement joins the Employee table with the Charges table on the EmpNo column. The second portion accounts for the employees who might be listed in the Employee table, but not the Charges table. The third portion of the statement accounts for the employees who might be listed in the Charges table and not in the Employee table. This ensures that all the information asked for is included in the response.

UNION Operator and the Outer Join

"Example 4" on page 1150 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 NULLs.

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 NULLs.

```
SELECT Offering.CourseNo, Offerings.Location, Enrollment.EmpNo
FROM Offerings, Enrollment
WHERE Offerings.CourseNo = Enrollment.CourseNo
UNION
SELECT Offerings.CourseNo, Offerings.Location, NULL
FROM Offerings, Enrollment
WHERE Offerings.CourseNo <> Enrollment.CourseNo;
```

The above UNION operation returns results equivalent to the results of the left outer join example shown above.

O.CourseNo	O.Location	E.EmpNo
C100	El Segundo	235
C100	El Segundo	668
C200	Dayton	?
C400	El Segundo	?

Example 5: Effect of Explicit NULLs on Data Type of a UNION

Set operator results evaluate to the data type of the columns defined in the first SELECT statement in the operation. When a column in the first SELECT is defined as an explicit NULL, the data type of the result is not intuitive.

Consider the following two examples, which you might intuitively think would evaluate to the same result but do not.

In the first, an explicit NULL is selected as a column value.

```
SELECT 'p', NULL
FROM TableVM
UNION
SELECT 'q', 145.87
FROM TableVM;
```

BTEQ returns the result as follows.

```
'p'         Null
---  -----------
p              ?
q            145
```

The expected value for the second row of the Null column probably differs from what you might expect—a decimal value of 145.87.

What if the order of the two SELECTs in the union is reversed?

```
SELECT 'q', 145.87
FROM TableVM
UNION
SELECT 'p', NULL
FROM TableVM;
```

BTEQ returns the result as follows.

```
'q'       145.87
---  -----------
p              ?
q         145.87
```

The value for q is now reported as its true data type—DECIMAL—and without truncation. Why the difference?

In the first union example, the explicit NULL is specified for the second column in the first SELECT statement. The second column in the second SELECT statement, though specified as a DECIMAL number, evaluates to an integer because in this context, NULL, though having no value, *does* have the data type INTEGER, and that type is retained for the result of the union.

The second union example carries the data type for the value 145.87—DECIMAL—through to the result.

You can confirm the unconverted data type for NULL and 145.87 by performing the following SELECT statement.

```
SELECT TYPE(NULL), TYPE(145.87)
```

BTEQ returns the result as follows.

```
Type(Null)         Type(145.87)
-----------------  ----------------------
INTEGER            DECIMAL(5,2)
```

Example 6: Effect of the Order of SELECT Statements on Data Type

The result of any UNION is always expressed using the data type of the selected value of the first SELECT. This means that SELECT A UNION SELECT B does not always return the same result as SELECT B UNION SELECT A unless you explicitly convert the output data type to ensure the same result in either case.

Consider the following complex unioned queries:

```
SELECT MIN(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND X8.i1 IN
(SELECT COUNT(*)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND X8.i1 = ANY
(SELECT COUNT(*)
FROM t7 X7
WHERE X7.i1 = ANY
(SELECT AVG(X1.i1)
FROM t1 X1)))
UNION
SELECT AVG(X4.i1)
FROM t4 X4
WHERE X4.i1 = ANY
(SELECT (X8.i1)
FROM t1 X1
RIGHT JOIN t8 X8 ON X8.i1=X1.i1
AND X8.i1 = IN
(SELECT MAX(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1=X1.i1
AND
(SELECT (X4.i1)
FROM t6 X6
RIGHT JOIN t4 X4 ON X6.i1=i1))));
```

The result is the following report.

```
Minimum(i1)
-----------
         -2
          0
```

You might intuitively expect that reversing the order of the queries on either side of the UNION would produce the same result. Because the data types of the selected value of the first SELECT can differ, this is not always true, as the following query on the same database demonstrates.

```
SELECT AVG(X4.i1)
FROM t4 X4
WHERE X4.i1 = ANY
(SELECT (X8.i1)
FROM t1 X1
RIGHT JOIN t8 X8 ON X8.i1 = X1.i1
AND X8.i1 = ANY
(SELECT MAX(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND
(SELECT (X4.i1)
FROM t6 X6
RIGHT JOIN t4 X4 ON X6.i1 = i
)
)
)
UNION
SELECT MIN(X8.i1)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND X8.i1 IN
(SELECT COUNT(*)
FROM t8 X8
LEFT JOIN t1 X1 ON X8.i1 = X1.i1
AND X8.i1 = ANY
(SELECT COUNT(*)
FROM t7 X7
WHERE X7.i1 = ANY
(SELECT AVG(X1.i1)
FROM t1 X1
)
);
```

The result is the following report.

```
Average(i1)
-----------
         -2
          1
```

The actual average is < 0.5. Why the difference when the order of SELECTs in the UNION is reversed? The following table explains the seemingly paradoxical results.

WHEN the first SELECT specifies this function ...	The result data type is ...	AND the value returned as the result is ...
AVG	REAL	1
MIN	INTEGER	truncated to 0

CHAPTER 25 String Operators and Functions

This chapter describes string operators and functions.

String Function Overview

SQL provides a concatenation operator and string functions to translate, concatenate, and perform other operations on strings.

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 literal and literal data strings, the term *string* is frequently used here to refer to all three of these data type families.

Data Types on Which String Functions can Operate

The following table lists all the data types that can be processed as strings. Note that not all types are acceptable to all functions. See the individual functions for the types they can process.

Data Type Grouping		
Character	Byte	Numeric
• CHARACTER • VARCHAR • CLOB	• BYTE • VARBYTE • BLOB	• BYTEINT • DECIMAL • FLOAT • INTEGER • NUMERIC • SMALLINT

ANSI Equivalence of Teradata SQL String Functions

Several of the Teradata SQL string functions are extensions to the ANSI SQL:2011 standard.

To maintain ANSI compatibility, use the ANSI equivalent functions instead of Teradata SQL string functions, when available.

Change this Teradata string function ...	To this ANSI string function in new applications ...
INDEX	POSITION
MINDEX†	

Change this Teradata string function ...	To this ANSI string function in new applications ...
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 5: "Attribute Functions."
comparison operators	Chapter 10: "Comparison Operators and Functions."
the LIKE predicate	Chapter 19: "Logical Predicates."

Effects of Server Character Sets on Character String Functions

String functions that operate on character data follow the rules listed below.

Uppercase Character Conversion for LATIN

For the LATIN server character set, the method of converting to uppercase characters is based on ISO 8859 Latin1.

Logical Characters vs. Physical Characters

For UNICODE, GRAPHIC and KANJISJIS server character sets, the functions operate on a logical character basis, except for the functions that are sensitive to the ANSI mode vs. Teradata mode switch.

Although the storage space for KANJISJIS is allocated on a physical basis and is not ANSI compatible, all string operations on this type operate on a character basis as dictated by ANSI.

Untranslatable KANJI1 Characters

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see “KANJI1 Character Set” in *International Character Set Support*.

Character string functions do not work on all characters in the KANJI1 server character set when the session character set is UTF8 or UTF16, because the KANJI1 server character set is ambiguous in regard to multibyte characters and some single-byte characters.

Recommendation: Unless the KANJI1 server character set is required, use the UNICODE server character set with the UTF8 and UTF16 session character sets for best results.

The following single-byte characters in KanjiEBCDIC to KANJI1 translations are mapped to the following Unicode character names.

Hexadecimal Value	Character	Unicode Character Name
0x10	¢	CENT SIGN
0x11	£	POUND SIGN
0x12	¬	NOT SIGN
0x13	\	REVERSE SOLIDUS
0x14	~	TILDE

However, with a KanjiSJIS character set, these hexadecimal values map to control characters.

Implicit Server Character Set Translation

For functions that operate on more than one argument, if the arguments have different server character sets, implicit translation rules take effect.

For details, see "Implicit Character-to-Character Translation" on page 599.

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:2011 standard. Do not use them as concatenation operators.

Solid and broken VERTICAL LINE character pairs (||) are ANSI SQL:2011 compliant forms of the concatenation operator.

Argument Types and Rules

Use the concatenation operator on strings and string expressions of type:

- Byte

  If any argument is a byte type, all other arguments must also be byte types.
- Numeric

  A numeric argument is converted to a character string using the format for the numeric value. For details about implicit numeric to character data type conversion, see "Implicit Numeric-to-Character Conversion" on page 661
- Character

  When the arguments are both character types, but have different server character sets, then implicit string conversion occurs. For details, see "Implicit Character-to-Character Translation" on page 599.

- UDTs that have implicit casts to a predefined character type.

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

  Implicit type conversion of UDTs for system operators and functions, including the concatenation operator, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

  For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Result Type and Attributes

The result of a concatenation operation is a string formed by concatenating the arguments in a left-to-right direction.

Here are the default result type and attributes for *arg1* || *arg2*:

- If the arguments are byte strings, the result is a byte string.
- If the arguments are numeric, character strings, or UDTs that are implicitly cast to character strings, the result is a character string.

If either argument is null, the result is null.

The data types and attributes of the arguments determine whether the result type of a concatenation operation is a fixed length or varying length string. Result types appear in the following table, where *n* is the sum of the lengths of all arguments:

IF this argument ...	Is this data type or attribute ...	THEN the result is this data type or attribute ...
either	VARBYTE	VARBYTE(*n*)
	VARCHAR	VARCHAR(*n*)
	numeric	
	UDT that is implicitly cast to VARCHAR	
	CLOB	CLOB(*n*)
	BLOB	BLOB(*n*)

IF this argument ...	Is this data type or attribute ...	THEN the result is this data type or attribute ...
both	BYTE	BYTE(*n*)
	CHARACTER (with same server character set)	CHARACTER(*n*)
	UDT that is implicitly cast to CHARACTER (with the same server character set)	
	CHARACTER (with different server character sets)	VARCHAR(*n*)
	UDT that is implicitly cast to CHARACTER (with different server character sets)	
	numeric	

When either argument is a character string that specifies the CASESPECIFIC attribute, the result also specifies the CASESPECIFIC attribute.

Example 1: Using Concatenation to Create More Readable Results

Literals, spaces, and the TITLE phrase can be included in the operation definition to format the result heading and improve readability.

For example, the following definition returns side titles, evenly spaced result strings, and a blank heading.

```
SELECT ('Sex ' || sex ||', Marital Status ' || mstat)(TITLE ' ')
FROM Employee ;

Sex M, Marital Status S
Sex F, Marital Status M
Sex M, Marital Status M
Sex F, Marital Status M
Sex F, Marital Status M
Sex M, Marital Status M
Sex F, Marital Status W
   ...
```

Example 2: Concatenating First Name With Last Name

Consider a table called names that contains last and first names columns, defined as VARCHAR, as listed here:

```
lname         fname
------------  ------------
Ryan          Loretta
Villegas      Arnando
Kanieski      Carol
Brown         Alan
```

Use string concatenation and a space separator to combine first and last names:

```
SELECT fname ||' '|| lname
FROM names
ORDER BY lname ;
```

The result is:

```
((fname||' ')||lname)
---------------------
Alan Brown
Carol Kanieski
Loretta Ryan
Arnando Villegas
```

Example 3: Concatenating Last Name With First Name

Change the SELECT and the separator to obtain last and first names:

```
SELECT lname||', '||fname
FROM names
ORDER BY lname;
```

The result is:

```
((lname||', ')||fname)
----------------------
Brown, Alan
Kanieski, Carol
Ryan, Loretta
Villegas, Arnando
```

Example 4: Concatenating Byte Strings

This example shows how to concatenate byte strings. Consider the following table definition:

```
CREATE TABLE tsttbla
  (column_1 BYTE(2)
  ,column_2 VARBYTE(10)
  ,column_3 BLOB(128K) );
```

The following values are inserted into table tsttbla:

```
INSERT tsttbla ('4142'XB, '7A7B7C'XB, '1A1B1C2B2C'XB);
```

The following SELECT statement concatenates column_2 and column_1 and column_3:

```
SELECT (column_2 || column_1 || column_3) (FORMAT 'X(20)')
FROM tsttbla ;
```

The result is:

```
((column_2||column_1)||column_3)
--------------------------------
7A7B7C41421A1B1C2B2C
```

The resulting data type is BLOB.

Concatenating Character Strings Having Different Server Character Sets

There are special considerations for the concatenation of character strings that specify different server character sets in the CHARACTER SET attribute.

Implicit translation rules apply. For details, see "Implicit Character-to-Character Translation" on page 599.

If the strings are fixed strings, then the result is varying with length equal to the sum of the lengths of the strings being concatenated.

This is true regardless of whether the string lengths are defined in terms of bytes or characters. So, a fixed *n*-byte KANJISJIS character string concatenated with a fixed *m*-character UNICODE string produces a VARCHAR(m+n) CHARACTER SET UNICODE result.

Consider the following table definition:

```
CREATE TABLE tab1
   (cunicode  CHARACTER(4)  CHARACTER SET UNICODE
   ,clatin    CHARACTER(3)  CHARACTER SET LATIN
   ,csjis     CHARACTER(3)  CHARACTER SET KANJISJIS);
```

The following values are inserted into table tab1:

```
INSERT tab1 ('abc', 'abc', 'abc');
```

The following table illustrates these concatenation properties.

Concatenation	Result	Type of Result
cunicode || clatin	'abcΔabc'	VARCHAR(7) CHARACTER SET UNICODE
clatin || csjis	'abcabc'	VARCHAR(6) CHARACTER SET UNICODE
cunicode || csjis	'abcΔabc'	VARCHAR(7) CHARACTER SET UNICODE

With the exception of KanjiEBCDIC, concatenation of KANJI1 character strings acts as described above. Under KanjiEBCDIC, any adjacent shift-out (<) and shift-in (>) characters within the resulting expression are removed. In this case, the result string is padded as necessary with trailing <single-byte space> characters.

Examples for Japanese Character Sets

The following tables show the results of concatenating string expressions under each of the Kanji character sets supported by Teradata Database.

These examples assume that the string expressions follow the rules defined in the chapter "SQL Data Definition" in *SQL Data Types and Literals.*

For an explanation of symbols and other notation in the examples, see "Character Shorthand Notation Used In This Book" on page 1334.

Example 1: KanjiEBCDIC

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
<ABC>	**<DEF>G**	**<ABCDEF>G**
<ABC>	**<>**	**<ABC>**

string_expression_1	string_expression_2	Result
<ABC>a	**<DEF>**	**<ABC>**a**<DEF>**

Example 2: KanjiEUC

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
ABCm	**DEF**g	**ABC**m**DEF**g
ss$_3$**A** ss$_2$**B** m	ss$_3$**C**	ss$_3$**A** ss$_2$**B** m ss$_3$**C**

Example 3: KanjiShift-JIS

```
string_expression_1 || string_expression_2
```

string_expression_1	string_expression_2	Result
mn**ABCX**	**B**	mn**ABCXB**
mn**ABCX**	g	mn**ABCX**g

ASCII

Purpose

Returns the decimal representation of the first character in *string_expr* as a NUMBER value. The decimal representation will reflect the character set of the input string.

Syntax

```
┬──────────────────┬─ ASCII ── ( string_expr ) ──────────────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A897

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
string_expr	a character string or string expression. If *string_expr* is NULL, the function returns NULL.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

ASCII is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

CHAR, VARCHAR, or CLOB

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result data type is NUMBER. ASCII returns the decimal representation in the character set of the input argument. For example, if a Unicode string is passed to the function, the decimal representation of the Unicode character is returned.

Example

The following query returns the result 121.

```
SELECT ASCII('y');
```

CHAR2HEXINT

Purpose

Returns the hexadecimal representation for a character string.

Syntax

```
── CHAR2HEXINT── ( character_string_expression ) ──
```

1101E173

where:

Syntax element ...	Specifies ...
character_string_expression	a character string or character string expression for which the hexadecimal representation is to be returned.

Argument Types

Use CHAR2HEXINT on character strings or character string expressions.

By default, Teradata Database performs implicit type conversion on a UDT argument that has an implicit cast that casts between the UDT and a predefined character type.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including CHAR2HEXINT, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

CHAR2HEXINT is not supported for CLOBs.

Result Type and Attributes

Here are the default attributes for CHAR2HEXINT(*character_string_expression*):

Data Type	Heading
CHARACTER	Char2HexInt(*character_string_expression*)

The length of the result is twice the length of *character_string_expression*.

The server character set of the result is LATIN.

CHAR2HEXINT and Literal Strings

You can apply CHAR2HEXINT to a string literal to determine its hexadecimal equivalent.

Character literals are treated as VARCHAR(*n*) CHARACTER SET UNICODE, where *n* is the length of the literal.

The following statement and results illustrate how CHAR2HEXINT operates on literal strings:

```
SELECT CHAR2HEXINT('123');

Char2HexInt('123')
------------------------
003100320033
```

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
----------------------------------------   ----------------
41636365737352696768747 3                   AccessRights
4163634C6F6752756C6554626C                  AccLogRuleTbl
4163634C6F6754626C                          AccLogTbl
4163636F756E7473                            Accounts
4163637467                                  Acctg
416C6C                                      All
436F70496E666F54626C                        CopInfoTbl
```

CHR

Purpose

Returns the Latin ASCII character given a numeric code value.

Syntax

```
┬──────────────────┬── CHR ── ( numeric_expr ) ──────────────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A896

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
numeric_expr	a numeric expression. *numeric_expr* must be zero or greater. If *numeric_expr* is greater than 255, an operation of *numeric_expr* mod 256 is executed to return a value between 0 and 255. If *numeric_expr* is NULL, the function returns NULL.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

CHR is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have a data type of NUMBER.

You can also pass arguments with data types such as BYTEINT, SMALLINT, INTEGER, or BIGINT that can be converted to NUMBER using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result data type is CHAR(1) CHARACTER SET LATIN.

Example

The following query returns the result 'B'.

```
SELECT CHR(66);
```

CSV

Purpose

CSV (Comma-Separated Value Data Unloading) returns input row column values in text format separated by a user-specified delimiter character.

Syntax

```
┌──────────────────┐
└ TD_SYSFNLIB.     ┘ CSV ( NEW VARIANT_TYPE ( ┌ ,─(8)─┐ value ┘ ) , delimit_string_value , quote_string_value ) —
```

1145-048A

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
NEW VARIANT_TYPE *value*	a numeric or character expression. NEW VARIANT_TYPE can support up to 8 row column values. Each row column value can have a maximum of 128 columns of any supported data type.
delimit_string_value	a character expression. A comma (,) is the default delimiter character.
quote_string_value	a character expression. If you specify a quotation mark character, for example **“”**, then columns defined as string data types are returned within quotation marks.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

CSV is an embedded services system function. For information on activating and invoking embedded services functions, see “Embedded Services System Functions” on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- NEW VARIANT_TYPE *value* = BYTEINT, SMALLINT, BIGINT, INTEGER, DECIMAL, FLOAT, DATE, TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, INTERVAL, CHAR, VARCHAR

  Note: *value* cannot be CLOB or GRAPHIC.
- *delimit_string_value* = VARCHAR
- *quote_string_value* = VARCHAR

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

CSV is a table function whose return value data type is VARCHAR in either the LATIN or UNICODE character set.

Example 1

The following query:

```
SELECT * FROM TABLE(CSV(NEW VARIANT_TYPE(dt.c1, dt.c2, dt.c3), ',', '"')
RETURNS (op varchar(64000) character set LATIN)) as t1;
```

CSV returns a string where column values are separated by commas and columns with string data types are enclosed in quotation marks.

Example 2

The following query:

```
SELECT * FROM TABLE (CSV(NEW VARIANT_TYPE(dt.c1, dt.c2, dt.c3), ',', '')
RETURNS (op varchar(32000) character set UNICODE)) as t1;
```

CSV returns a string where column values are separated by comma and columns with string data types are not enclosed by any characters.

CSVLD

Purpose

CSVLD (Comma-Separated Value Data Loading) takes in a comma-separated string produced through the CSV table function, parses the string, and returns VARCHAR columns.

Syntax

```
┬──────────────┬─ CSVLD ( data_string_value, delim_string_value, quote_string_value ) ──────
└─ TD_SYSFNLIB. ─┘
```

1101A919

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
data_string_value	a string argument.
delim_string_value	a character argument. A comma (,) is the default delimiter character. *delim_string_value* is used to determine the boundaries of the *data_string_value*.
quote_string_value	a character argument. If you specify a quotation mark character, for example **“”**, then columns defined as string data types are returned within quotation marks.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

CSVLD is an embedded services system function. For information on activating and invoking embedded services functions, see “Embedded Services System Functions” on page 26.

Argument Types and Rules

Expressions passed to this function must have a VARCHAR data type.

You can also pass arguments with data types that can be converted to the above type using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

CSVLD is a table function can return up to 1024 VARCHAR columns in either the LATIN or UNICODE character set.

The number of output columns specified in this function must match the comma separated values in the input string.

Example

The following query:

```
SELECT * FROM TABLE (CSVLD(load_date.data, ',', '"')
RETURNS (p1 varchar(100), p2 varchar(100))) as T1;
```

CSVLD parses the comma-separated strings and returns two VARCHAR columns.

EDITDISTANCE

Purpose

Returns the minimum number of edit operations (insertions, deletions, substitutions and transpositions) required to transform *string1* into *string2*.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── EDITDISTANCE ── ( string1, string2 ┌──────────────────┐ ) ──────────
                                                              │                  │
                                                              └─ , ci, cd, cs, ct ─┘
                                                                                  1101A885
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
string1	a character string or string expression.
string2	a character string or string expression.
ci	the relative cost of an insert operation. *ci* cannot be a negative value. If not specified, a default value of 1 is used.
cd	the relative cost of a delete operation. *cd* cannot be a negative value. If not specified, a default value of 1 is used.
cs	the relative cost of a substitute operation. *cs* cannot be a negative value. If not specified, a default value of 1 is used.
ct	the relative cost of a transpose operation. *ct* cannot be a negative value. If not specified, a default value of 1 is used.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

EDITDISTANCE is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *string1* = CHAR, VARCHAR, or CLOB
- *string2* = CHAR, VARCHAR, or CLOB
- *ci* = INTEGER
- *cd* = INTEGER
- *cs* = INTEGER
- *ct* = INTEGER

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result data type is INTEGER.

Usage Notes

EDITDISTANCE measures the similarity between two strings. A low number of deletions, insertions, substitutions or transpositions implies a high similarity. The insertions, deletions, substitutions, and transpositions are based on the Damerau-Levenshtein Distance algorithm with modifications for costed operations.

If either *string1* or *string2* is NULL, the function returns NULL.

Example 1

The following query returns a result of 9.

```
SELECT EDITDISTANCE('Jim D. Swain', 'John Smith');
```

Example 2

The following query returns a result of 0 since the strings are the same.

```
SELECT EDITDISTANCE('John Smith', 'John Smith');
```

Example 3

The following query returns a result of 9.

```
SELECT EDITDISTANCE('Jim D. Swain', 'John Smith', 2, 1, 1, 2);
```

Example 4

The following query returns a result of 11.

```
SELECT EDITDISTANCE('John Smith', 'Jim D. Swain', 2, 1, 1, 2);
```

INDEX

Purpose

Returns the position in *string_expression_1* where *string_expression_2* starts.

Syntax

```
── INDEX ── ( string_expression_1 ,string_expression_2 ) ──
```

FF07D253

where:

Syntax element ...	Specifies ...
string_expression_1	a full string to be searched.
string_expression_2	a substring to be searched for its position within the full string.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Use POSITION instead of INDEX for ANSI SQL:2011 compliance.

Argument Types and Rules

INDEX operates on the following types of arguments:

- Character
- Byte

  If one string expression is of type BYTE, then both string expressions must be of type BYTE.
- Numeric

  If any string expression is numeric, then it is converted implicitly to CHARACTER type.
- UDTs that have implicit casts that cast between the UDT and any of the following predefined types:
  - Numeric
  - Character
  - DATE
  - Byte

  To define an implicit cast for a UDT, use CREATE CAST and specify AS ASSIGNMENT. For details on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including INDEX, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

INDEX does not support CLOBs or BLOBs.

For more information on implicit type conversion, see Chapter 13: "Data Type Conversions."

Result Type and Attributes

Here are the default result type and attributes for INDEX(*arg1*, *arg2*):

Data Type	Heading
INTEGER	Index(*arg1*, *arg2*)

Expected Values

The following rules apply to the value that INDEX returns:

- If *string_expression_2* is not found in *string_expression_1*, then the result is zero.
- If *string_expression_2* is null, then the result is null.
- If the arguments are character types, INDEX returns a logical character position, not a byte position, except when the server character set of the arguments is KANJI1 and the session client character set is KanjiEBCDIC.

  For details, see "Rules for KANJI1 Server Character Set" on page 1181.

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 599.

The CASESPECIFIC attribute affects whether characters are considered to be a match.

IF the session mode is ...	THEN the default case specification for character columns and literals is ...
ANSI	CASESPECIFIC.
Teradata	NOT CASESPECIFIC. The exception is character data of type GRAPHIC, which is always CASESPECIFIC.

To override the default case specification, you can apply the CASESPECIFIC or NOT CASESPECIFIC phrase to a character column in CREATE TABLE or ALTER TABLE.

Or, you can apply the CASESPECIFIC or NOT CASESPECIFIC phrase to the INDEX character string arguments.

IF ...	THEN ...
either argument has a CASESPECIFIC attribute (either by default or specified explicitly)	simple Latin letters are considered to be matching only if they are the same letters and the same case.
both arguments have a NOT CASESPECIFIC attribute (either by default or specified explicitly)	before the operation begins, some characters are converted to uppercase. If the character is a lowercase simple Latin letter, the character is converted to uppercase before the operation begins. If the character is a non-Latin single byte character, a multibyte character, or a byte indicating a transition between single-byte and multibyte character data, the character is not converted to uppercase.

Using the rules for character type arguments, if you want INDEX to match letters only if they are the same letters in the same case, specify the CASESPECIFIC phrase with at least one of the arguments. For example:

```
SELECT Name
FROM Employee
WHERE INDEX(Name, 'X' (CASESPECIFIC)) = 1;
```

If you want INDEX to match letters without considering the case, specify the NOT CASESPECIFIC phrase with both of the arguments.

Rules for KANJI1 Server Character Set

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see “KANJI1 Character Set” in *International Character Set Support*.

When the server character set is KANJI1 and the client character set is KanjiEBCDIC, the offset count includes Shift-Out/Shift-In characters, but they are not matched. They are treated only as an indication of a transition from a single byte character and an multibyte character.

The nonzero position of the result is reported as follows:

IF the character set is ...	THEN the result is the ...
KanjiEBCDIC	position of the first byte of the logical character offset (including Shift-Out/Shift-In in the offset count) within *string_expression_1*.
other than KanjiEBCDIC	logical character offset within *string_expression_1*.

Relationship Between INDEX and POSITION

INDEX and POSITION behave identically, except on character type arguments when the client character set is KanjiEBCDIC, the server character set is KANJI1, and an argument contains a multibyte character.

For an example of when the two functions return different results for the same data, see "How POSITION and INDEX Differ" on page 1210.

Example 1

The following table shows examples of simple INDEX expressions and their results.

Expression	Result
INDEX('catalog','log')	5
INDEX('catalog','dog')	0
INDEX('41424344'XB,'43'XB)	3

Example 2

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for *string_1* and the server character set for *string_2* differ. In these cases, both arguments are converted to UNICODE (if needed) and the characters are matched logically.

IF *string_1* is ...		AND *string_2* is ...		THEN the result is ...
Character Set	Data	Character Set	Data	
UNICODE	92年abc	LATIN	abc	4
UNICODE	abc	UNICODE	c	3
KANJISJIS	92年04	UNICODE	0	4

Example 3

The following examples show how INDEX(*string_1*, *string_2*) operates when the server character set for both arguments is KANJI1 and the client character set is KanjiEBCDIC.

Note that for KanjiEBCDIC, results are returned in terms of physical units, making INDEX DB2-compliant in that environment.

IF *string_1* contains ...	AND *string_2* contains ...	THEN the result is ...
MN<**AB**>	<**B**>	6
MN<**AB**>	<**A**>	4
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 concatenation, SUBSTRING, and INDEX might be more instructive. Suppose the employee table contains the following values.

```
empno      name
---------- -----------
10021      Smith T
10007      Aguilar J
10018      Russell S
10011      Chin M
10019      Newman P
```

You can transpose the form of the names from the name column selected from the employee table and change the punctuation in the report using the following query:

```
SELECT empno,
SUBSTRING(name FROM INDEX(name,' ')+1 FOR 1)||'. '||
SUBSTRING(name FROM 1 FOR INDEX(name, ' ')-1)
(TITLE 'Emp Name')
FROM employee ;
```

The system returns the following report.

```
empno      Emp Name
---------- --------------
10021      T. Smith
10007      J. Aguilar
10018      S. Russell
10011      M. Chin
10019      P. Newman
```

INITCAP

Purpose

Modifies a string argument and returns the string with the first character in each word in uppercase and all other characters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Syntax

```
┬──────────────┬─ INITCAP ── ( string ) ──────────────────────────────────
└─ TD_SYSFNLIB. ─┘
```

1101A894

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
string	a character string or string expression. If *string* is NULL, the function returns NULL.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

INITCAP is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

CHAR, VARCHAR, or CLOB

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The result data type and character set are the same as those of the input string. For example, if the input string has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Example

The following query returns the result 'Hello World'.

```
SELECT INITCAP('hello WORLD');
```

INSTR

Purpose

Searches the *source_string* argument for occurrences of *search_string*.

Syntax

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character string or string expression.
search_string	a string of characters that the function searches for in *source_string*.
position	that the search will begin at this position in *source_string*. If *position* is not specified, the search starts at the beginning of *source_string*. If *position* is negative, the function counts and searches backwards from the end of *source_string*. *position* cannot have a value of zero.
occurrence	which occurrence of *search_string* to find in *source_string*. If *occurrence* is not specified, the function searches for the first occurrence. If *occurrence* is greater than 1, the function searches for additional occurrences beginning with the second character in the previous occurrence. *occurrence* cannot be zero or a negative value.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

INSTR is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *source_string* = CHAR, VARCHAR, or CLOB
- *search_string* = CHAR, VARCHAR, or CLOB
- *position* = INTEGER, BIGINT, or NUMBER
- *occurrence* = INTEGER, BIGINT, or NUMBER

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

If a match is found, the function returns the position (starting from 1) in the *source_string* for the match; otherwise the function returns 0. If any of the input arguments are NULL, the function returns NULL.

Result Type

The result data type is NUMBER.

Example 1

The following query returns the result 20 indicating the position of 'ch' in 'chip'. This is the second occurrence of 'ch' with the search starting from the second character of the source string.

```
SELECT INSTR('choose a chocolate chip cookie','ch',2,2);
```

Example 2

The following query returns the result 2, which indicates the position of the first occurrence of 'N' in the source string with the search starting from the beginning of the string.

```
SELECT INSTR('INSTR FUNCTION','N');
```

LENGTH

Purpose

Returns the number of characters in the string argument.

Syntax

```
┬──────────────────┬─ LENGTH ── ( string ) ──────────────────────────────
└─ TD_SYSFNLIB. ───┘
```

1101A895

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
string	a character string or string expression. If *string* is NULL, the function returns NULL.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

LENGTH is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

CHAR, VARCHAR, or CLOB

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result data type is NUMBER.

Example

The following query returns the result 7.

```
SELECT LENGTH('astring');
```

LOWER

Purpose

Returns a character string identical to *character_string_expression*, except that all uppercase letters are replaced with their lowercase equivalents.

Syntax

```
— LOWER — (character_string_expression) ————————————
```

FF07D091

where:

Syntax element ...	Specifies ...
character_string_expression	a character string or character string expression for which all uppercase characters are to be replaced with their lowercase equivalents.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Argument Types

LOWER is valid only for character strings or character string expressions. The function does not accept CLOBs and non-character arguments.

By default, Teradata Database performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including LOWER, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 13: “Data Type Conversions.”

Result Type and Attributes

Here are the default result type and attributes for LOWER(*arg*):

Data Type	Heading
Same type as *arg*	Lower(*arg*)

The LOWER function returns the result in the same character set as the input argument. The exception is when the input is KANJI1 data; LOWER returns the result in the LATIN server character set.

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see "KANJI1 Character Set" in *International Character Set Support*.

Usage Notes

See *International Character Set Support* for the internal mappings that Teradata Database uses for the LOWER function.

Teradata SQL has the type attribute NOT CASESPECIFIC that allows case blind comparisons, but the type attributes CASESPECIFIC and NOT CASESPECIFIC are Teradata extensions to the ANSI standard.

For ANSI portability, use the UPPER function for case blind comparisons with ANSI-compliant syntax. For details, see "UPPER" on page 1252.

Example

The use of LOWER to return and store values is shown in the following example.

```
SELECT LOWER (last_name)
FROM names;

INSERT INTO names
SELECT LOWER(last_name),LOWER(first_name)
FROM newnames;
```

The identical result is achieved with a USING phrase.

```
USING (last_name CHAR(20),first_name CHAR(20))
INSERT INTO names (LOWER(:last_name), LOWER(:first_name));
```

LPAD

Purpose

Returns the *source_string* padded to the left with the characters in *fill_string* so that the resulting string is *length* characters.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── LPAD ── ( source_string, length ┌──────────────┐ ) ──────────────
                                                           │              │
                                                           └─ , fill_string ─┘
```

1101A893

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character string or string expression. If the length of *source_string* is greater than *length*, *source_string* is truncated to *length* characters.
length	the number of characters in the resulting string.
fill_string	an optional string of characters used to pad the *source_string*. If *fill_string* is not specified, *source_string* will be padded to the left with space characters. The sequence of characters in *fill_string* is replicated as necessary.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

LPAD is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *source_string* = VARCHAR or CLOB
- *length* = INTEGER, BIGINT, or NUMBER

- *fill_string* = CHAR, VARCHAR, or CLOB

If any of the input arguments are NULL, the function returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result data type and character set are the same as those of the *source_string* argument. For example, if the *source_string* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Example 1

The following query returns the result 'yzybuilding'.

```
SELECT LPAD('building', 11, 'yz');
```

Example 2

The following query returns the result 'build'.

```
SELECT LPAD('building', 5, 'yz');
```

Example 3

The following query returns the result ' building'. The space character is used by default to pad the source string.

```
SELECT LPAD('building', 11);
```

LTRIM

Purpose

Returns the string argument *string1*, with its left-most characters removed up to the first character that is not in the string argument *string2*.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── LTRIM ── ( string1 ┌──────────────┐ ) ──────────────────────────
                                              │              │
                                              └─ , string2 ──┘
```

1101A891

where:

Syntax element…	Specifies…
TD_SYSFNLIB	the name of the database where the function is located.
string1	a character string or string expression.
string2	an optional string of characters that will be removed from *string1*. If *string2* is not specified, the default is to use a single space character.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

LTRIM is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *string1* = VARCHAR or CLOB
- *string2* = VARCHAR

If any of the input arguments are NULL, the function returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The result data type and character set are the same as those of the *string1* argument. For example, if the *string1* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Example 1

The following query returns the result 'XxyLAST WORD'. LTRIM removes the individual occurrences of 'x' and 'y', and it stops removing characters when it encounters 'X' because 'X' is not in the *string2* argument.

```
SELECT LTRIM('xyxXxyLAST WORD','xy');
```

Example 2

The following query returns the result 'LEFT TRIM'. LTRIM removes the spaces from the left, and it stops removing characters when it encounters 'L' because 'L' is not in the *string2* argument. Since the *string2* argument is not explicitly specified, the default of a single space is used.

```
SELECT LTRIM('  LEFT TRIM');
```

NGRAM

Purpose

Returns the number of n-gram matches between *string1* and *string2*.

A high number of matching n-gram patterns implies a high similarity between the two strings.

Syntax

```
┌──────────────────────┐
└─ TD_SYSFNLIB. ───────┘── NGRAM ── ( string1, string2, length ┌──────────────────┐ ) ──
                                                               └─ , position ─────┘
                                                                                    1101A886
```

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
string1	a character string or string expression. If *string1* is NULL, the function returns NULL.
string2	a character string or string expression. If *string2* is NULL, the function returns NULL.
length	the value *n* in *n*-gram, which is the comparison length.
position	that the n-gram is a positional n-gram match.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

NGRAM is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *string1* = CHAR, VARCHAR, or CLOB
- *string2* = CHAR, VARCHAR, or CLOB
- *length* = INTEGER

- *position* = INTEGER

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

If the data type of *string1* is CHAR or VARCHAR, the result data type is INTEGER.

If the data type of *string1* is CLOB, the result data type is BIGINT.

Usage Notes

For positional n-gram matching, the position as well as the pattern must match when measuring similarity. The *position* value indicates how far away positionally the match may be between the 2 strings as follows:

- If *position* is set to a value of zero, the match must be at the same position in the 2 strings.
- If *position* is set to a value of *x*, the match must be within *x* positions in the 2 strings. For example, if *position* = 2, then the match must be within 2 positions in the 2 strings.

As an example, for a string of 'abc', the 1-grams (length =1) are 'a', 'b', and 'c'. The 2-grams (length =2) are 'ab' and 'bc'. The 3-gram (length = 3) is 'abc'. By definition, there are no 4-grams or greater.

The function returns zero in the following cases:

- If the *length* argument is greater than the length of either *string1* or *string2*.
- If the *length* argument is <= 0 or if either *string1* or *string2* is an empty string.

Patterns beyond the length of 255 are ignored.

Example 1

The following query returns a result of 2. The 3-grams 'mit' and 'ith' match. Note that 'Smi' and 'smi' do not match because of the difference in case.

```
SELECT NGRAM('John Smith','Allen smith 1',3);
```

Example 2

The following query returns a result of zero. There are no 3-grams in the first string expression of '' since the length of the string is less than 3.

```
SELECT NGRAM ('','str1 empty',3);
```

Example 3

The following query returns a result of zero. There are no 0-grams in the strings.

```
SELECT NGRAM ('test with zero length', 'test with zero length',0);
```

Example 4

The following query returns a result of 3. The 1-grams 'a', 'b', and 'c' match.

```
SELECT NGRAM ('abc','yyabc',1);
```

Example 5

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match.

```
SELECT NGRAM ('abc','yyabc',2);
```

Example 6

The following query returns a result of zero. The 2-grams 'ab' and 'bc' match, but they are not within 1 position of each other.

```
SELECT NGRAM ('abc','yyabc',2, 1);
```

Example 7

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match, and they are within 2 positions of each other.

```
SELECT NGRAM ('abc','yyabc',2, 2);
```

Example 8

The following query returns a result of 2. The 2-grams 'ab' and 'bc' match, and they are at the same position in each string.

```
SELECT NGRAM ('abc','abc',2, 0);
```

Example 9

The following query returns a result of zero. There are no 5-grams since the length of either input string is less than 5.

```
SELECT NGRAM ('abc','abc',5,0);
```

NVP

Purpose

Extracts the value of a name-value pair where the name in the pair matches the name and the number of the occurrence specified.

Syntax

```
┌──────────────────┐
└ TD_SYSFNLIB.─┘ NVP ── (instring, name_to_search ┌──────────────────────────┐ ┌──────────────────────────┐ ┌──────────────────────────┐ ) ──
                                                  └ , name_delimiters ─┘ └ , value_delimiters ─┘ └ , occurrence ─┘
                                                                                                                  1101A986
```

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
instring	the name-value pairs separated by multibyte delimiters. If *instring* is VARCHAR, the maximum size of *instring* is 32,000. If *instring* is CLOB, the maximum size of *instring* is 2,000,000.
name_to_search	the name whose *instring* value NVP returns.
name_delimiters	the multibyte delimiters used to separate name-value pairs. Delimiters can contain any characters. They are separated from each other in the string by spaces. If a space is used as part of a delimiter, it must be escaped using a backslash (\). The maximum length of any delimiter is 10. This parameter is optional and if not specified, the default value is & ampersand).
value_delimiters	the multibyte delimiters used to associate a name to its value in a name-value pair. Delimiters can contain any characters. They are separated from each other in the string by spaces. If a space is used as part of a delimiter, it must be escaped using a backslash (\). The maximum length of any delimiter is 10. This parameter is optional and if not specified, the default value is = (equal sign).
occurrence	the number of occurrences of *name_to_search* that NVP searches for. This parameter is optional and if not specified, the default value is 1.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

NVP is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *instring* = VARCHAR or CLOB
- *name_to_search* = VARCHAR
- *name_delimiters* = VARCHAR
- *value_delimiters* = VARCHAR
- *occurrence* = INTEGER

The character set of *instring*, *name_to_search*, *name_delimiters*, and *value_delimiters* can be LATIN or UNICODE.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

If a delimiter is part of a longer delimiter, the longer delimiter has precedence in the matching process.

Adjacent delimiters are treated as a single delimiter.

Result Type

The result data type is VARCHAR with the same character set as that of *instring*.

Example 1

The following query:

```
SELECT NVP ('entree:orange chicken#entree2:honey salmon', 'entree','#',
':', 1);
```

returns `'orange chicken'`.

Example 2

The following query:

```
SELECT NVP('store = whole foods&&store: ?Bristol farms','store', '&&',
'\ =\  :\ ?', 2);
```

returns `'Bristol farms'`.

In this example, *occurrence* = 2 instructs NVP to search for the second occurrence of `'store'`.

Example 3

The following query:

```
SELECT NVP('entree=orange chicken&entree2=honey salmon', 'entree', 1)
```

returns `'orange chicken'`.

In this example, *name_delimiters* is & (default value) and *value_delimiters* is = (default value).

Example 4

The following query:

```
SELECT NVP('entree=orange chicken&entree2=honey salmon', 'entree');
```

returns `'orange chicken'`.

In this example, *name_delimiters* is & (default value), *value_delimiters* is = (default value), and *occurrence* is 1 (default value).

OREPLACE

Purpose

Replaces every occurrence of *search_string* in the *source_string* with the *replace_string*. Use this function either to replace or remove portions of a string.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── OREPLACE ── ( source_string, search_string ┌──────────────────────┐ ) ──
                                                                      │                      │
                                                                      └─ , replace_string ───┘
```

1101A889

where:

Syntax element…	Specifies…
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character string or string expression. If *source_string* is NULL, the function returns NULL.
search_string	a string of characters that will be replaced or removed from *source_string*. If *search_string* is NULL, the function returns *source_string*.
replace_string	an optional string of characters that replaces the characters specified by *search_string*. If *replace_string* is NULL or is an empty string, or is omitted, all occurrences of *search_string* are removed from the *source_string*.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

OREPLACE is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

CHAR, VARCHAR, or CLOB

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result data type and character set depend on those of the *source_string* argument.

- If the *source_string* is CHAR, the result data type is VARCHAR.
- If the *source_string* is VARCHAR, the result data type is VARCHAR.
- If the *source_string* is CLOB, the result data type is CLOB.

For example, if the *source_string* argument has a data type of CHAR CHARACTER SET UNICODE, then the result data type will be VARCHAR CHARACTER SET UNICODE.

The maximum length of a VARCHAR or CLOB result value is the maximum length that Teradata supports for these data types. An error is returned if the result string is larger than the maximum result string size.

Usage Notes

OREPLACE provides a superset of the functionality provided by the OTRANSLATE function. OTRANSLATE provides single character, 1-to-1 substitution while OREPLACE allows you to substitute 1 string for another, as well as to remove character strings.

For information about the OTRANSLATE function, see "OTRANSLATE" on page 1206.

Example 1

The following query returns the string 'TD14.0 is the current version'. The string '13.1' in the source string was replaced by the string '14.0'.

```
SELECT OREPLACE('TD13.1 is the current version', '13.1', '14.0');
```

Example 2

The following query returns the string 'This chair is a brown chair'. Both occurrences of the search string 'bag' in the source string were replaced by the string 'chair'.

```
SELECT OREPLACE('This bag is a brown bag', 'bag', 'chair');
```

Example 3

The following query returns the string 'TD13.1 is the current version'. The source string is returned unchanged since the search string is NULL. The result would be the same if the search string is an empty string or a string that has no matches in the source string.

```
SELECT OREPLACE('TD13.1 is the current version', NULL, '14.0');
```

Example 4

The following query returns the string 'We removed the extra word'. The occurrence of the search string 'superfluous' was removed from the source string.

```
SELECT OREPLACE('We removed the superfluous extra word',
   'superfluous', NULL);
```

Example 5

The result set from the following query will have an ADDRESS column that is the concatenation of the ADDRESS1 and ADDRESS2 columns from the CUSTOMER table, with every occurrence of 'st.' replaced with ' street'.

```
SELECT OREPLACE(ADDRESS1||ADDRESS2, ' st.', ' street') AS ADDRESS
   from CUSTOMER;
```

OTRANSLATE

Purpose

Returns *source_string* with every occurrence of each character in *from_string* replaced with the corresponding character in *to_string*.

Syntax

```
┬──────────────────┬── OTRANSLATE ── ( source_string, from_string, to_string ──────── ) ──
└── TD_SYSFNLIB. ──┘
```

1101A887

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character string or string expression. If *source_string* is NULL, the function returns NULL.
from_string	a string of characters that will be replaced in *source_string*. If *from_string* is NULL, the function returns *source_string*.
to_string	a string of characters that replaces the characters specified by *from_string*. If *to_string* is: • NULL is NULL the function returns NULL. • An empty string, the corresponding character in *from_string* is removed.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

OTRANSLATE is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types: CHAR or VARCHAR

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The result data type is VARCHAR. The character set is the same as that of the *source_string* argument.

Usage Notes

If the first character in *from_string* occurs in the *source_string*, all occurrences of it are replaced by the first character in *to_string*. This repeats for all characters in *from_string* and for all characters in *from_string*. The replacement is performed character-by-character, that is, the replacement of the second character is done on the string resulting from the replacement of the first character.

If *from_string* contains more characters than *to_string*, the extra characters are removed from the *source_string*. If an extra character was involved in a mapping, it is not removed, just ignored.

If *from_string* contains fewer characters than *to_string*, the extra characters in *to_string* have no effect.

If the same character occurs more than once in *from_string*, only the replacement character from the *to_string* corresponding to the first occurrence is used.

Example 1

The following query returns the string 'TD14.0 is the current database version'. The occurrence in *source_string* of the character in *from_string* ('3') is replaced by the character in *to_string* ('4').

```
SELECT OTRANSLATE('TD13.0 is the current database version',
   '3', '4');
```

Example 2

In the following query, the characters 'T' and 'h' are replaced with 'S' and 'p', resulting in the string 'Spin and Spick'. Next, the extra character 'k' in the *from_string* (where there is no corresponding character in the *to_string*) is removed from the *source_string*. The resulting string is 'Spin and Spic'.

```
SELECT OTRANSLATE('Thin and Thick', 'Thk', 'Sp');
```

Example 3

In the following query, the characters 'T' and 'h' are replaced with 'S' and 'p'. The character 'T' occurs twice in the *from_string*, but only the replacement character in the *to_string* that corresponds to the first occurrence of 'T' is used. That is, only 'S' is used to replace 'T'. Next, the character 'k' is replaced with 'x', and the extra characters 'y' and 'z' in the *to_string* are ignored. The resulting string is 'Spin and Spicx'.

```
SELECT OTRANSLATE('Thin and Thick', 'ThTk', 'Sptxyz');
```

Example 4

The following query returns the *source_string* unchanged since *from_string* is NULL.

```
SELECT OTRANSLATE('Thin and Thick', NULL, 'Spt');
```

POSITION

Purpose

Returns the position in *string_expression_2* where *string_expression_1* starts.

Syntax

```
── POSITION ──(string_expression_1 ── IN ── string_expression_2) ──
```

FF07D090

where:

Syntax element ...	Specifies ...
string_expression_1	a substring to be searched for its position within the full string.
string_expression_2	a full string to be searched.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Use POSITION instead of INDEX for ANSI SQL:2011 conformance. POSITION and INDEX behave identically except when the client character set is KanjiEBCDIC and the server character for an argument is KANJI1 and contains multibyte characters.

Use POSITION in place of MINDEX. (MINDEX no longer appears in this book because its use is deprecated and it will not be supported after support for KANJI1 is dropped.)

Argument Types and Rules

POSITION operates on the following types of arguments:

- Character, except for CLOB
- Byte, except for BLOB

  If one string expression is of type BYTE, then both expressions must be of type BYTE.
- Numeric

  Numeric string expressions are converted implicitly to CHARACTER type.
- UDTs that have implicit casts that cast between the UDT and any of the following predefined types:
  - Numeric
  - Character

- DATE
- Byte

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including POSITION, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 13: "Data Type Conversions."

Result Type and Attributes

Here are the default result type and attributes for POSITION(*arg1* IN *arg2*):

Data Type	Heading
INTEGER	Position(*arg1* in *arg2*)

Expected Values

POSITION returns a value according to the following rules.

IF ...	THEN the result is ...
either argument is null	null.
string_expression_1 has length zero	one.
string_expression_1 is a substring within *string_expression_2*	the position in *string_expression_2* where *string_expression_1* starts.
none of the preceding is true	zero.

If the arguments are character types, then regardless of the server character set, the value for POSITION represents the position of a logical character, not a byte position.

How POSITION and INDEX Differ

INDEX and POSITION behave identically except when the session client character set is KanjiEBCDIC, the server character set is KANJI1, and the parent string contains a multibyte character.

This is the only case for which the results of these two functions differ when performed on the same data.

Suppose we create the following table.

```
CREATE TABLE iptest (
 column_1 VARCHAR(30) CHARACTER SET Kanji1
 column_2 VARCHAR(30) CHARACTER SET Kanji1);
```

We then insert the following set of values for the columns.

column_1	column_2
MN<**AC**>	<**C**>
MN<**AC**>P	<**A**>
MN<**AB**>P	P
MN<**AB**>P	<**B**>

The client session character set is KanjiEBCDIC5026_0I. Now we perform a query that demonstrates how INDEX and POSITION return different results in this condition.

```
SELECT column_1, column_2, INDEX(column_1,column_2)
FROM iptest;
```

The result of this query looks like the following:

```
column_1     column_2        Index(column_1,column_2)
-----------  -----------     ------------------------
MN<AC>       <C>                                   6
MN<AC>P      <A>                                   4
MN<AB>P      P                                     9
MN<AB>P      <B>                                   6
```

With the same session characteristics in place, perform the semantically identical query on the table using POSITION instead of INDEX.

```
SELECT column_1, column_2, POSITION(column_2 IN column_1)
FROM iptest;
```

The result of this query looks like the following:

```
column_1     column_2     Position(column_2 in column_1)
-----------  -----------  ------------------------------
MN<AC>       <C>                                      4
MN<AC>P      <A>                                      3
MN<AB>P      P                                        5
MN<AB>P      <B>                                      4
```

The different results are accounted for by the following differences in how INDEX and POSITION operate in this particular case.

- INDEX counts Shift-Out and Shift-In characters; POSITION does not.
- INDEX counts bytes; POSITION counts logical characters. As a result, an **A**, for example, counts as two bytes (two *physical* characters) for INDEX, but only one *logical* character for POSITION.

RPAD

Purpose

Returns the *source_string* padded to the right with the characters in *fill_string* so that the resulting string is *length* characters.

Syntax

```
┬──────────────────┬─ RPAD ── ( source_string, length ┬──────────────┬ ) ──────────────
└─ TD_SYSFNLIB. ───┘                                 └─ , fill_string ─┘
```

1101A892

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
source_string	a character string or string expression. If the length of *source_string* is greater than *length*, *source_string* is truncated to *length* characters.
length	the number of characters in the resulting string.
fill_string	an optional string of characters used to pad the *source_string*. The sequence of characters in *fill_string* is replicated as necessary. If *fill_string* is not specified, *source_string* will be padded to the right with space characters.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

RPAD is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *source_string* = VARCHAR or CLOB
- *length* = INTEGER, BIGINT, or NUMBER
- *fill_string* = CHAR, VARCHAR, or CLOB

If any of the input arguments are NULL, the function returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The result data type and character set are the same as those of the *source_string* argument. For example, if the *source_string* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is ARCHAR CHARACTER SET UNICODE.

Example 1

The following query returns the result 'buildingyzy'.

```
SELECT RPAD('building', 11, 'yz');
```

Example 2

The following query returns the result 'build'.

```
SELECT RPAD('building', 5, 'yz');
```

Example 3

The following query returns the result 'building '. The space character is used by default to pad the source string.

```
SELECT RPAD('building', 11);
```

RTRIM

Purpose

Returns the string argument *string1*, with its right-most characters removed up to the first character that is not in the string argument *string2*.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── RTRIM ── ( string1 ┌──────────────┐ ) ──────────────────────────
                                              │              │
                                              └─ , string2 ──┘
```

1101A890

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
string1	a character string or string expression.
string2	an optional string of characters that will be removed from *string1*. If *string2* is not specified, the default is to use a single space character.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

RTRIM is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have one of the following data types:

- *string1* = VARCHAR or CLOB
- *string2* = VARCHAR

If any of the input arguments are NULL, the function returns NULL.

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see “Compatible Types” in *SQL External Routine Programming*.

Result Type

The result data type and character set are the same as those of the *string1* argument. For example, if the *string1* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Example 1

The following query returns the result 'TURNERyxX'. RTRIM removes the individual occurrences of 'x' and 'y', and it stops removing characters when it encounters 'X' because 'X' is not in the *string2* argument.

```
SELECT RTRIM('TURNERyxXxy,'xy');
```

Example 2

The following query returns the result ' RIGHTT TRIM' because the leading spaces are not removed. It stops removing characters when it encounters 'M' because 'M' is not in the *string2* argument. Since the *string2* argument is not explicitly specified, the default of a single space is used.

```
SELECT RTRIM('  RIGHTT TRIM   ');
```

SOUNDEX

Purpose

Returns a character string that represents the Soundex code for *string_expression*.

Syntax

```
SOUNDEX — ( — string_expression — ) —
```

KO01A060

where:

Syntax element ...	Specifies ...
string_expression	a character string or expression that contains a surname to be evaluated in simple Latin characters. Soundex is case insensitive. Embedded or trailing pad characters within *character_string* return an error to the requestor.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Argument Types

Use SOUNDEX on character strings or character string expressions that use the LATIN or UNICODE server character set.

SOUNDEX does not accept CLOB types.

By default, Teradata Database performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including SOUNDEX, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 13: “Data Type Conversions.”

Definition: Simple Latin Characters

A simple Latin character is one that does not have diacritical marks such as tilde (~) or acute accent (´).

There are 26 uppercase simple Latin characters and 26 lowercase simple Latin characters.

Definition: Soundex

Soundex is a system that codes surnames having the same or similar sounds, but variant spellings. The Soundex system was first used by the National Archives in 1880 to index the United States census.

Soundex codes begin with the first letter of the surname followed by a three-digit code. Zeros are added to names that do not have enough letters.

Soundex Coding Guide

The following process outlines the Soundex coding guide:

1. Retain the first letter of the name.
2. Drop all occurrences of the following letters:

   A, E, I, O, U, Y, H, W

   in other positions.
3. Assign the following number to the remaining letters after the first letter:

   1 = B, F, P, V

   2 = C, G, J, K, Q, S X, Z

   3 = D, T

   4 = L

   5 = M, N

   6 = R
4. If two or more letters with the same code are adjacent in the original name or adjacent except for any intervening H or W, omit all but the first.
5. Convert the form “letter, digit, digit, digit,” by adding trailing zeros if less than three digits.
6. Drop the rightmost digits if more than three digits.
7. Names with adjacent letters having the same equivalent number are coded as one letter with a single number

   Surname prefixes are generally not used.

Example 1

The following SELECT statement returns the result that follows.

```
SELECT SOUNDEX ('ashcraft');

Soundex('ashcraft')
-------------------
a261
```

The surname “ashcraft” initially evaluates to “a2h2613,” but the following Soundex rules convert the result to a261.

- “h” is dropped because it occurs in the third position. Soundex drops all occurrences of the following characters in any position other than the first.

  A, E, I, O, U, Y, H, W
- “2” is dropped because it represents the second occurrence of one of the following characters:

  C, G, J, K, Q, S X, Z

  If two or more characters with the same code are adjacent in the original name, or adjacent except for any intervening H or W, Soundex omits all but the code for the first occurrence of the character in the returned code.
- “3” is dropped because Soundex drops the rightmost digits if *character_string* evaluates to more than three digits following the initial simple Latin character.

Example 2

“Example 2” and “Example 3” on page 1219 use the following table data:

```
SELECT family_name FROM family;

family_name
-----------
John
Joan
Joey
joanne
michael
Bob
```

Here are the results of the SOUNDEX function on the data in the family_name column:

```
SELECT SOUNDEX(TRIM(family.family_name));

Soundex(TRIM(BOTH FROM family_name))
------------------------------------
J500
J500
B100
J000
m240
j500
```

Example 3

Find all family names in Family that sound like “Joan”.

```
SELECT family_name
FROM family
WHERE SOUNDEX(TRIM(family.family_name)) = SOUNDEX('Joan');

family_name
-----------
John
Joan
Joanne
```

Examples of 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.

STRING_CS

Purpose

Returns a heuristically derived integer value that you can use to help determine which KANJI1-compatible client character set was used to encode *string_expression*.

Caution: The result is not guaranteed correct, but should work for most strings likely to be encountered.

Syntax

```
— STRING_CS —( string_expression )—
```

1101A515

where:

Syntax element ...	Specifies ...
string_expression	a CHAR or VARCHAR character string or expression.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Argument Types

Use STRING_CS on character strings or character string expressions that use the KANJI1 server character set. (Non-KANJI1 character strings will be coerced to KANJI1, but the results are unlikely to be useful.)

STRING_CS does not accept CLOB or UDT types.

Result Value

STRING_CS returns a heuristically derived INTEGER value that you can use to help determine the client character set that was used to encode the KANJI1 character string or expression. The result value can also help determine which client character set to use to interpret the character data.

IF the result value is ...	THEN the heuristic found that *string_expression* ...
-1	most likely uses a single-byte client character set encoding, but it may also contain a mix of encodings.

IF the result value is ...	THEN the heuristic found that *string_expression* ...
0	does not contain anything distinguishable from any particular character set, so any character set that you use to interpret *string_expression* provides the same result. Not all translations use the same interpretation for the characters represented by 0x5C and 0x7E, however. If *string_expression* contains: 0x5C and you want it to be interpreted as REVERSE SOLIDUS, use a single-byte character set. 0x7E and you want it to be interpreted as TILDE, use a single-byte character set. 0x5C and you want it to be interpreted as YEN SIGN, or 0x7E and you want it to be interpreted as OVERLINE, use any of the following: • KANJISJIS_0S • KANJIEBCDIC5026_0I • KANJIEBCDIC5035_0I • KATAKANAEBCDIC • KANJIEUC_0U
1	uses the encoding of one of the following: • KANJIEBCDIC5026_0I • KANJIEBCDIC5035_0I • KATAKANAEBCDIC
2	uses the encoding of KANJIEUC_0U.
3	uses the encoding of KANJISJIS_0S.

Usage Notes

STRING_CS helps determine which encoding to use when using the TRANSLATE function to translate a string from the KANJI1 server character set to the UNICODE server character set.

IF the result value is ...	THEN substitute the following value for *source_TO_target* in TRANSLATE(*string_expression* USING *source_to_target*) ...
-1	KANJI1_SBC_TO_UNICODE.
0	KANJI1_SBC_TO_UNICODE.
1	KANJI1_KANJIEBCDIC_TO_UNICODE.
2	KANJI1_KANJIEUC_TO_UNICODE.
3	KANJI1_KANJISJIS_TO_UNICODE.

For more information on TRANSLATE, see "TRANSLATE" on page 1234.

Example 1: Using STRING_CS to Determine the Client Character Set

Consider the following table definition:

```
CREATE TABLE SysNames
   (SysID INTEGER
   ,SysName VARCHAR(30) CHARACTER SET KANJI1);
```

Suppose the session character set is KANJIEBCDIC5026_0I. The following statement inserts the mixed single-byte/multibyte character string '<**TEST**>Q' into the SysName column of the SysNames table:

```
INSERT SysNames (101, '0E42E342C542E242E30FD8'XC);
```

Using STRING_CS to determine the client character set that was used to encode the string produces the results that follow:

```
SELECT STRING_CS(SysName) FROM SysNames WHERE SysID = 101;

String_CS(SysName)
------------------
                 1
```

Example 2: Using STRING_CS to Translate a KANJI1 String to UNICODE

Consider the SysNames table from the preceding example, "Example 1: Using STRING_CS to Determine the Client Character Set."

The following statement uses STRING_CS to determine which encoding to use to translate strings in the SysName column from the KANJI1 server character set to the UNICODE server character set:

```
SELECT CASE STRING_CS(SysName)
   WHEN 0 THEN TRANSLATE(SysName USING KANJI1_SBC_TO_UNICODE)
   WHEN 1 THEN TRANSLATE(SysName USING KANJI1_KANJIEBCDIC_TO_UNICODE)
   WHEN 2 THEN TRANSLATE(SysName USING KANJI1_KANJIEUC_TO_UNICODE)
   WHEN 3 THEN TRANSLATE(SysName USING KANJI1_KANJISJIS_TO_UNICODE)
   ELSE TRANSLATE(SysName USING KANJI1_SBC_TO_UNICODE)
   END
FROM SysNames;
```

STRTOK

Purpose

Splits *instring* into tokens based on the specified list of delimiter characters and returns the *n*th token, where *n* is specified by the *tokennum* argument.

Syntax

where:

Syntax element...	Specifies...
TD_SYSFNLIB	the name of the database where the function is located.
instring	a character string or string expression.
delimiter	a list of delimiter characters. Each character in the string is considered a delimiter character. If not specified, the default delimiter character is a space character.
tokennum	the ordinal token to return. If not specified, the default value is 1.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

STRTOK is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *instring* = VARCHAR(32000) or CLOB
- *delimiter* = VARCHAR(64)
- *tokennum* = INTEGER

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result data type is VARCHAR and the character set is the same as that of the *instring* argument. For example, if the *instring* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

Usage Notes

STRTOK has similar semantics as the standard C/C++ library STRTOK function.

If *instring* contains fewer tokens than *tokennum*, the function returns a zero-length string.

If either *instring* or *delimiter* is NULL, the function returns NULL.

Example

Consider the following table and inserted data.

```
   CREATE TABLE t (id INTEGER, str VARCHAR(256));

   INSERT INTO t VALUES (1,'Teradata-Warehouse 13.10 - Combine 2
powerful forms of business intelligence (BI).');

   INSERT INTO t VALUES (2,'http://www.teradata.com/');
```

The following query returns the 3rd token from the t.str string value. There are only 2 tokens in the second string, so the function returns a zero-length string for that row.

```
SELECT id, STRTOK(t.str, ' -/', 3) FROM t;
```

The output from the query is:

```
          id  token
-----------  ------------------------------------------------------------
           1  13.10
           2
```

STRTOK_SPLIT_TO_TABLE

Purpose

Splits strings into a table of tokens based on the provided delimiter(s) string.

Note: STRTOK_SPLIT_TO_TABLE has similar semantics to the standard C/C++ library STRTOK function.

Syntax

```
┌──────────────────────┐
│                      │
└─ TD_SYSFNLIB. ───────┘── STRTOK_SPLIT_TO_TABLE ── ( inkey, instring, delimiters ) ──────── Ⓐ

Ⓐ── RETURNS ── ( outkey, tokennum, token ) ──────────────────────────────────────────────
```

1101A917

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
inkey	a numeric or character argument.
instring	a character or CLOB argument. If *instring* is NULL, the STRTOK_SPLIT_TO_TABLE returns NULL.
delimiters	a character argument. If *delimiters* is not specified, the value defaults to a space character.
outkey	a numeric or character argument. *outkey* is used to identify the instring in the result set.
tokennum	an integer. *tokennum* is the ordinal position of the token in the input string.
token	a character argument. The *token* from the input string in the same character set as *instring*.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Invocation

STRTOK_SPLIT_TO_TABLE is an embedded services system function. For information on activating and invoking embedded services functions, see "Embedded Services System Functions" on page 26.

Argument Types and Rules

Expressions passed to this function must have the following data types:

- *inkey* = NUMERIC or VARCHAR
- *instring* = VARCHAR(32000) or CLOB
- *delimiter* = VARCHAR(64)

You can also pass arguments with data types that can be converted to the above types using the implicit data type conversion rules that apply to UDFs.

Note: The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Teradata Database. If an argument cannot be converted to the required data type following the UDF implicit conversion rules, it must be explicitly cast.

For details, see "Compatible Types" in *SQL External Routine Programming*.

Result Type

The result row type is:

- *outkey* = NUMERIC OR VARCHAR(10)
- *tokennum* = INTEGER
- *token* = VARCHAR(256)

Example

If:

```
CREATE TABLE t (id integer, str varchar(256)character set unicode);
```

and:

```
INSERT INTO t VALUES (1,'Teradata-Warehouse 13.10 - Combine 2 powerful
forms of business intelligence (BI).');
```

and:

```
insert into t values (2,'http://www.teradata.com/');
```

then:

```
SELECT d.* FROM TABLE (strtok_split_to_table(t.id, t.str, ' -/')
RETURNS (outkey integer, tokennum integer, token varchar(20)character
set unicode) ) as d order by 1,2;

 outkey   tokennum  token
-----------  -----------  --------------------------
```

```
1            1  Teradata
1            2  Warehouse
1            3  13.10
1            4  Combine
1            5  2
1            6  powerful
1            7  forms
1            8  of
1            9  business
1           10  intelligence
1           11  (BI).
2            1  http:
2            2  www.teradata.com
```

SUBSTRING / SUBSTR

Purpose

Extracts a substring from a named string based on position.

ANSI Syntax

```
── SUBSTRING ─ (string_expression ─ FROM ─ n1 ┬──────────┬ ) ──
                                              └─FOR n2 ─┘
                                                          FF07D256
```

where:

Syntax Element ...	Specifies ...
string_expression	a string expression from which the substring is to be extracted.
n1	the starting position of the substring to extract from *string_expression*.
FOR	a keyword indicating that the searched substring is bounded on the right by the value *n2*. If you omit FOR *n2*, then you extract the entire right hand portion of the named string or string expression, beginning at the position named by *n1*. If *string_expression* is a BYTE or CHAR type and you omit FOR *n2*, trailing binary zeros or pad characters are trimmed.
n2	the length of the substring to extract from *string_expression*. If *n2* < 0, the function returns an error.

Teradata Syntax

```
─ SUBSTR ─ (string_expression,n1 ┬──────┬ ) ──────────────────
                                └─,n2 ─┘
                                                FF07D257
```

where:

Syntax Element ...	Specifies ...
string_expression	a string expression from which the substring is to be extracted.
n1	the starting position of the substring to extract from *string_expression*.
n2	the length of the substring to be extracted from *string_expression*. If *string_expression* is a BYTE or CHAR type and you omit *n2*, trailing binary zeros or pad characters are trimmed. If *n2* < 0, the function returns an error.

ANSI Compliance

This is ANSI SQL:2011 compliant.

SUBSTR is a Teradata extension to the ANSI SQL:2011 standard.

Argument Types and Rules

SUBSTRING and SUBSTR operate on the following types of arguments:

- Character
- Byte
- Numeric

  If the *string_expression* argument is numeric, it is implicitly converted to CHARACTER type.
- UDTs that have implicit casts to any of the following predefined types:
  - Character
  - Numeric
  - Byte
  - DATE

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

  Implicit type conversion of UDTs for system operators and functions, including SUBSTRING and SUBSTR, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 13: "Data Type Conversions."

Result Type and Attributes

Here are the default result type and attributes for SUBSTR(*string*, *n1*, *n2*) and SUBSTRING(*string* FROM *n1* FOR *n2*):

If the *string* argument is a:

- BLOB, the result type is BLOB(*n*).
- byte string other than BLOB, the result type is VARBYTE(*n*).
- CLOB, the result type is CLOB(*n*).
- numeric or character string other than CLOB, the result type is VARCHAR(*n*).

In ANSI mode, the value of *n* for the resulting BLOB(*n*), VARBYTE(*n*), CLOB(*n*), or VARCHAR(*n*) is the same as the original string. In Teradata mode, the value of *n* for the result type depends on the number of characters or bytes in the resulting string. To get the data type of the resulting string, use the TYPE function.

Result Value

SUBSTRING/SUBSTR extracts *n2* characters or bytes from *string_expression* starting at position *n1*.

To get the number of characters or bytes in the resulting string, use the BYTE function for byte strings and the CHARACTER_LENGTH function for character strings.

If either of the following conditions are true, SUBSTRING/SUBSTR returns a zero length string:

- ($n1 > string_length$) AND ($0 \leq n2$)
- ($n1 < 1$) AND ($0 \leq n2$) AND (($n2 + n1 - 1$) ≤ 0)

Usage Rules for SUBSTRING and SUBSTR

SUBSTRING is the ANSI SQL:2011 syntax. Teradata syntax using SUBSTR is supported for backward compatibility. Use SUBSTRING in place of SUBSTR for ANSI compliance.

Use SUBSTRING in place of MSUBSTR. (MSUBSTR no longer appears in this book because its use is deprecated and it will not be supported after support for KANJI1 is dropped.)

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see “KANJI1 Character Set” in *International Character Set Support*.

Difference Between SUBSTRING and SUBSTR

SUBSTRING and SUBSTR perform identically except when they operate on character strings in Teradata mode where the server character set is KANJI1 and the client character set is KanjiEBCDIC.

In this case, SUBSTR interprets *n1* and *n2* as physical units, making the DB2-compliant SUBSTR operate on a byte-by-byte basis. Shift-Out and Shift-In bytes are significant because the result might be formatted incorrectly. For example, the result string might not contain either the opening Shift-Out character or the closing Shift-In character.

Otherwise, if *string_expression* is character data, then SUBSTRING expects mixed single byte and multibyte character strings and operates on logical characters that are valid for the character set of the session. In this case, *n1* is a positive integer pointing to the first character of the result and *n2* is in terms of logical characters.

Example 1

Suppose sn is a CHARACTER(15) field of Serial IDs for Automobiles and positions 3 to 5 represent the country of origin as three letters.

For example:

```
12JAP3764-35421
37USA9873-26189
11KOR1221-13145
```

To search for serial IDs of cars made in the USA:

```
SELECT make, sn
FROM autos
WHERE SUBSTRING (sn FROM 3 FOR 3) = 'USA';
```

Example 2

If we want the last five characters of the serial ID, which represent manufacturing sequence number, another substring can be accessed.

```
SELECT make, SUBSTRING (sn FROM 11) AS sequence
FROM autos
WHERE SUBSTRING (sn FROM 3 FOR 3) = 'USA';
```

Example 3

Suppose nameaddress is a VARCHAR(120) field, and the application used positions 1 to 30 for name, starting address at position 31. To return address only, but limit the number of characters returned to 50 use:

```
...
SUBSTRING (nameaddress FROM 31 FOR 50)
```

This returns an address of up to 50 characters.

Example 4

The following example shows a SELECT statement requesting substrings from a character field in positions 1 through 4 for every row:

```
SELECT SUBSTRING (jobtitle FROM 1 FOR 4)
FROM employee ;
```

The result is as follows.

```
Substring(jobtitle From 1 For 4)
--------------------------------
Tech
Cont
Sale
Secr
Test
 ...
```

Example 5

Consider the following table:

```
CREATE TABLE cstr
  (c1 CHAR(3) CHARACTER SET LATIN
  ,c2 CHAR(10) CHARACTER SET KANJI1);

INSERT cstr ('abc', '92年abc');
```

Here are some examples of how to use SUBSTR to extract substrings from the KanjiEUC client character set.

Function	Result
`SELECT SUBSTR(c2, 2, 3) FROM cstr;`	'2年a'
`SELECT SUBSTR(c1, 2, 2) FROM cstr;`	'bc'

Example 6

Consider the following table:

```
CREATE TABLE ctable1
  (c1 VARCHAR(11) CHARACTER SET KANJI1);
```

The following table shows the difference between SUBSTR and SUBSTRING in Teradata mode for KANJI1 strings from KanjiEBCDIC client character set.

IF c1 contains ...	THEN this query ...	Returns ...
MN<**ABC**>P	`SELECT SUBSTR(c1,2) FROM ctable1;`	N<**ABC**>P
	`SELECT SUBSTR(c1,3,8) FROM ctable1;`	<**ABC**>
	`SELECT SUBSTR(c1,4) FROM ctable1;`	**ABC**>P Note: The client application might not be able to properly interpret the resulting multibyte characters because the shift out (<) is missing.
	`SELECT SUBSTRING(c1 FROM 2) FROM ctable1;`	N<**ABC**>P
	`SELECT SUBSTRING(c1 FROM 3 FOR 8) FROM ctable1;`	<**ABC**>P
	`SELECT SUBSTRING(c1 FROM 4) FROM ctable1;`	<**BC**>P

Example 7

The following table shows examples for the KanjiEUC client character set, where ctable1 is the table defined in Example 6.

IF c1 contains ...	THEN this query ...	Returns ...
A ss_2**B CD**	`SELECT SUBSTR(c1,2) FROM ctable1;`	ss_2**B CD**
ss_3**A** ss_2**B** ss_3**C** ss_2**D**	`SELECT SUBSTR(c1,2,2) FROM ctable1;`	ss_2**B** ss_3**C**

Example 8

The following table shows examples for KanjiShift-JIS client character set, where ctable1 is the table defined in Example 6.

IF c1 contains ...	THEN this query ...	Returns ...
mn**ABCX**	`SELECT SUBSTR(c1, 6, 1) FROM ctable1;`	**X**
	`SELECT SUBSTR(c1,4) FROM ctable1;`	**BCX**

Example 9

The following statement applies the SUBSTRING function to a CLOB column in table full_text and stores the result in a CLOB column in table sub_text.

```
INSERT sub_text (text)
SELECT SUBSTRING (text FROM 9 FOR 128000)
FROM full_text;
```

TRANSLATE

Purpose

Converts a character string or character string expression from one server character set to another server character set.

Syntax

```
— TRANSLATE — ( — character_string_expression — USING — source_repertoire_name — (A)

(A) ┌─────────────┐ _TO _target_repertoire_name ┌──────────────┐┌──────────────┐ ) ────────
    └ _encoding   ┘                             └ _suffix      ┘└ WITH ERROR    ┘
```

1101E198

where:

Syntax element ...	Specifies ...
character_string_expression	a character string to translate to another server character set. If *character_string_expression* is not a character type, an error is returned.
source_repertoire_name	the source character set of the string to translate. For supported values, see "Supported Translations Between Character Sets" on page 1237. A value of LOCALE can be specified for *source_repertoire_name* to translate a character string from LATIN or KANJI1 to UNICODE using a source repertoire determined by the language support mode of the system and the client character set of the session. For details, see "Supported Translations Between Character Sets" on page 1237.
_encoding	an optional literal for translating from KANJI1 to UNICODE that indicates a specific encoding of KANJI1. The *_encoding* option is not allowed if LOCALE is specified for *source_repertoire_name* or *target_repertoire_name*.

Syntax element ...	Specifies ...
_encoding (continued)	If the translation is from these character sets: • KatakanaEBCDIC • KanjiEBCDIC5026_0I • KanjiEBCDIC5038_0I use the following value for *_encoding*: _KanjiEBCDIC KanjiEUC_0U, use the following value for *_encoding*: _KanjiEUC KanjiShiftJIS_0S, use the following value for *_encoding*: _KANJISJIS ASCII or EBCDIC, use the following value for *_encoding*: _SBC
target_repertoire_name	the target character set of the string to translate. For supported values, see "Supported Translations Between Character Sets" on page 1237. A value of LOCALE can be specified for *target_repertoire_name* to translate a character string from UNICODE to LATIN or KANJI1 using a target repertoire determined by the language support mode of the system and the client character set of the session. For details, see "Supported Translations Between Character Sets" on page 1237.
_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 • _PadGraphic • _KanjiEUC • _Fullwidth • _KANJISJIS • _Halfwidth • _SBC • _FoldSpace • _PadSpace • _VarGraphic The *_suffix* option is not allowed if LOCALE is specified for *source_repertoire_name* or *target_repertoire_name*.
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 1240).

Argument Types

Use TRANSLATE on character strings or character string expressions.

By default, Teradata Database performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including TRANSLATE, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Result Type and Attributes

The default attributes for TRANSLATE (*string* USING *source*_TO_*target*) are as follows.

- If the argument is CHAR or VARCHAR, the result is VARCHAR(*n*) or CHARACTER SET *target*
- If the argument is CLOB, the result is CLOB(*n*) or CHARACTER SET *target*

where source_TO_target determines the character set value of target, according to the supported translations in "Supported Translations Between Character Sets" on page 1237.

Supported Translations for CLOB Strings

The following translations are supported for CLOB strings:

- LATIN_TO_UNICODE
- UNICODE_TO_LATIN

Supported Translations Between Character Sets

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see "KANJI1 Character Set" in *International Character Set Support*.

The following table lists the supported values that you can use for *source_repertoire_name*_TO_*target_repertoire_name* to translate between server character sets.

Value of *source*_TO_*target*	Source Character Set	Target Character Set
GRAPHIC_TO_KANJISJIS	GRAPHIC	KANJISJIS
GRAPHIC_TO_LATIN	GRAPHIC	LATIN
GRAPHIC_TO_UNICODE	GRAPHIC	UNICODE
GRAPHIC_TO_UNICODE_PadSpace	GRAPHIC	UNICODE
KANJI1_KanjiEBCDIC_TO_UNICODE	KANJI1	UNICODE
KANJI1_KanjiEUC_TO_UNICODE	KANJI1	UNICODE
KANJI1_KANJISJIS_TO_UNICODE	KANJI1	UNICODE
KANJI1_SBC_TO_UNICODE	KANJI1	UNICODE
KANJISJIS_TO_GRAPHIC	KANJISJIS	GRAPHIC
KANJISJIS_TO_LATIN	KANJISJIS	LATIN
KANJISJIS_TO_UNICODE	KANJISJIS	UNICODE
LATIN_TO_GRAPHIC	LATIN	GRAPHIC
LATIN_TO_KANJISJIS	LATIN	KANJISJIS
LATIN_TO_UNICODE	LATIN	UNICODE
LOCALE_TO_UNICODE	KANJI1	UNICODE
	LATIN	
UNICODE_TO_GRAPHIC	UNICODE	GRAPHIC
UNICODE_TO_GRAPHIC_PadGraphic	UNICODE	GRAPHIC
UNICODE_TO_GRAPHIC_VarGraphic	UNICODE	GRAPHIC
UNICODE_TO_KANJI1_KanjiEBCDIC	UNICODE	KANJI1
UNICODE_TO_KANJI1_KanjiEUC	UNICODE	KANJI1
UNICODE_TO_KANJI1_KANJISJIS	UNICODE	KANJI1

Value of *source*_TO_*target*	Source Character Set	Target Character Set
UNICODE_TO_KANJI1_SBC	UNICODE	KANJI1
UNICODE_TO_KANJISJIS	UNICODE	KANJISJIS
UNICODE_TO_LATIN	UNICODE	LATIN
UNICODE_TO_LOCALE	UNICODE	KANJI1
		LATIN
UNICODE_TO_UNICODE_FoldSpace	UNICODE	UNICODE
UNICODE_TO_UNICODE_Fullwidth	UNICODE	UNICODE
UNICODE_TO_UNICODE_Halfwidth	UNICODE	UNICODE
UNICODE_TO_UNICODE_NFC	UNICODE	UNICODE
UNICODE_TO_UNICODE_NFD	UNICODE	UNICODE
UNICODE_TO_UNICODE_NFKC	UNICODE	UNICODE
UNICODE_TO_UNICODE_NFKD	UNICODE	UNICODE

If the value specified for *source_repertoire_name*_TO_*target_repertoire_name* is UNICODE_TO_LOCALE or LOCALE_TO_UNICODE, the repertoire that the translation uses for LOCALE is determined by the language support mode for the system and the client character set for the session.

IF the language support mode is ...	AND the session character set is ...	THEN the repertoire that the translation uses for LOCALE is ...
standard	any	LATIN

IF the language support mode is ...	AND the session character set is ...	THEN the repertoire that the translation uses for LOCALE is ...
Japanese	• ASCII • LATIN1252_0A • LATIN1_0A • LATIN9_0A • EBCDIC • EBCDIC037_0E • EBCDIC273_0E • EBCDIC277_0E • any other client character set with a name that has a suffix of _0A or _0E • a single-byte, extended site-defined client character set	KANJI1_SBC
	• KANJIEBCDIC5026_0I • KANJIEBCDIC5035_0I • KATAKANAEBCDIC • any other client character set with a name that has a suffix of _0I	KANJI1_KANJIEBCDIC
	• UTF8 • UTF16 • KanjiShiftJIS_0S • any other client character set with a name that has a suffix of _0S • a multibyte extended site-defined client character set	KANJI1_KANJISJIS
	• KanjiEUC_0U • any other client character set with a name that has a suffix of _0U	KANJI1_KanjiEUC

Source Characters That Generate Errors

The following table lists the characters that generate errors for specific *source_repertoire_name*_TO_*target_repertoire_name* translations. For supported translations that do not appear in the table, only the error character generates errors.

Value of *source*_TO_*target*	Source Characters That Generate Errors
• LATIN_TO_GRAPHIC • KANJISJIS_TO_GRAPHIC • UNICODE_TO_GRAPHIC	non-GRAPHIC

Value of *source*_TO_*target*	Source Characters That Generate Errors
• LATIN_TO_KANJISJIS • KANJI1_KANJISJIS_TO_UNICODE • GRAPHIC_TO_KANJISJIS • UNICODE_TO_KANJI1_KANJISJIS • UNICODE_TO_KANJISJIS • LOCALE_TO_UNICODE or UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is KANJI1_KANJISJIS	non-KANJISJIS
• KANJI1_KanjiEBCDIC_TO_UNICODE • UNICODE_TO_KANJI1_KanjiEBCDIC • LOCALE_TO_UNICODE or UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is KANJI1_KanjiEBCDIC	non-KanjiEBCDIC KANJI1 is very permissive, so there may be characters outside the defined region of the encoding as well as illegal form-of-use errors.
• KANJI1_KanjiEUC_TO_UNICODE • UNICODE_TO_KANJI1_KanjiEUC • LOCALE_TO_UNICODE or UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is KANJI1_KanjiEUC	non-KanjiEUC
• KANJISJIS_TO_LATIN • GRAPHIC_TO_LATIN • UNICODE_TO_LATIN • UNICODE_TO_KANJI1_SBC • UNICODE_TO_LOCALE where the repertoire that the translation uses for LOCALE is LATIN or KANJI1_SBC	non-LATIN

Error Characters Assigned by the WITH ERROR Option

The error characters substituted for offending characters that cannot be translated to a designated target character set are defined in the following table.

Target Character Set	Error Character
LATIN	0x1A
KANJI1	0x1A
KANJISJIS	0x1A
UNICODE	U+FFFD
GRAPHIC	U+FFFD

Suffixes

The *_suffix* variable is used for translations that map source characters to semantically different characters. They indicate the nature of the semantic transformation.

The translations perform minor, yet essential, semantic changes to the data, such as halfwidth/fullwidth conversions, and Space folding modification.

The *_suffix* variable also indicates the form of character data translated from UNICODE to the KANJI1 server character set in one of the four possible encodings, for example Unicode_TO_Kanji1_KanjiEBCDIC. For a list of the encodings, see the definition of _encoding in "Syntax" on page 1234.

This form of translation is also useful for migrating object names. For information, see "Migration" on page 1242.

Translations Between Fullwidth and Halfwidth Character Data

UNICODE has an area known as the compatibility zone. Among other things, this zone includes halfwidth and fullwidth variants of characters that exist elsewhere in the standard.

Translations between fullwidth and halfwidth are provided by the following *source_repertoire_name*_TO_*target_repertoire_name* values.

*source*_TO_*target*	Meaning
UNICODE_TO_UNICODE_Fullwidth	This translation maps any character with a fullwidth variant to that variant. At the same time, it maps any character defined by the standard as a halfwidth variant to its non-halfwidth counterpart outside the compatibility zone. Other characters remain unchanged by the translation.
UNICODE_TO_UNICODE_Halfwidth	This translation maps any character with a halfwidth variant to that variant, and all fullwidth variants to their non-fullwidth counterparts. Other characters remain unchanged by the translation.
UNICODE_TO_GRAPHIC_VarGraphic	This translation is an ANSI equivalent to the VARGRAPHIC function.

Note that these translations are useful for maintaining more information as a step in translating GRAPHIC to LATIN and vice versa.

For details on the mappings, see *International Character Set Support*.

Space Folding

Space folding is performed via UNICODE_TO_UNICODE_FoldSpace. All characters defined as space are converted to U+0020.

All other characters are left unchanged.

For details on which characters are converted to U+0020, see *International Character Set Support*.

UNICODE Normalization Form Translations

Teradata supports translation using the 4 UNICODE normalization forms: NFC, NFD, NFKC, and NFKD, which correspond to the ANSI NORMALIZE function. You can perform these translations using:

- UNICODE_TO_UNICODE_NFC
- UNICODE_TO_UNICODE_NFD
- UNICODE_TO_UNICODE_NFKC
- UNICODE_TO_UNICODE_NFKD

Because normalization functions can cause errors due to not preserving BMP characters, you should use the TRANSLATE_CHK function to verify a clean translation. See

Pad Character Translation

The following translations do not translate the pad character.

*source*_TO_*target*	Pad Character Translation
GRAPHIC_TO_UNICODE	A GRAPHIC string that includes an Ideographic Space is translated to a UNICODE string with an Ideographic Space.
UNICODE_TO_GRAPHIC	A UNICODE string with a Space character generates an error when translated to GRAPHIC.

If you require pad character translation, use one of the following translations.

*source*_TO_*target*	Pad Character Translation
GRAPHIC_TO_UNICODE_PadSpace	Converts all occurrences of Ideographic Space (U+3000) to Space (U+0020).
UNICODE_TO_GRAPHIC_PadGraphic	Converts all occurrences of Space to Ideographic Space.

Other characters are not affected. Note that the position of a character does not affect the translation, so not only trailing pad characters are modified.

Migration

During the migration process, any GRAPHIC data in the old form must be translated to the new canonical form. Note that this involves converting the pad characters from Null (U+0000) to Ideographic Space (U+3000).

Implicit Character Data Type Conversion

TRANSLATE performs implicit conversion if the *string* server character set does not match the type implied by *source_repertoire_name*.

An implicit conversion generates an error if a character from *character_string_expression* has no corresponding character in the *source_repertoire_name* type. This holds regardless of whether you specify the WITH ERROR option.

For example, the following function first translates the string from UNICODE to LATIN, because Teradata Database treats literals as UNICODE, and then translates the string from LATIN to KANJISJIS. However, the translation generates an error because the last character is not in the LATIN repertoire.

```
...
TRANSLATE('abc年' USING LATIN_TO_KanjiSJIS WITH ERROR)
...
```

To circumvent the problem if error character substitution is acceptable, specify two levels of translation, as used in the following example.

```
...
TRANSLATE((TRANSLATE(_UNICODE 'abc年' USING UNICODE_TO_LATIN WITH
ERROR)) USING LATIN_TO_KanjiSJIS WITH ERROR)
...
```

Examples

Function	Result	Type of the Result
TRANSLATE('abc' USING UNICODE_TO_LATIN)	'abc'	VARCHAR(3) CHARACTER SET LATIN
TRANSLATE('abc' USING UNICODE_TO_UNICODE_Fullwidth)	'abc'	VARCHAR(3) CHARACTER SET UNICODE
TRANSLATE('abc年' USING UNICODE_TO_LATIN WITH ERROR) where ε represents the designated error character for LATIN (0x1A).	'abcε'	VARCHAR(4) CHARACTER SET LATIN

Related Topics

For details on the mappings that Teradata Database uses for the TRANSLATE function, see *International Character Set Support*.

TRANSLATE_CHK

Purpose

Determines if a TRANSLATE conversion can be performed without producing errors; returns an integer test result. Use TRANSLATE_CHK to filter untranslatable strings. You can choose to select translatable strings only, or untranslatable strings only, depending on how you form your SELECT statement.

Syntax

```
— TRANSLATE_CHK —( — character_string_expression – USING — source_repertoire_name —(A)

(A)┬──────────┐ _TO _target_repertoire_name ┬──────────┐ )──────────────────────────
   └ _encoding ┘                            └ _suffix  ┘

                                                                          1101E199
```

where:

Syntax element ...	Specifies ...
character_string_expression	a character string to be translated to another server character set. If *character_string_expression* is not a character type, an error is returned.
source_repertoire_name	the source character set of the string to be translated. For supported values, see "Supported Translations Between Character Sets" on page 1237. A value of LOCALE can be specified for *source_repertoire_name* to translate a character string from LATIN or KANJI1 to UNICODE using a source repertoire determined by the language support mode of the system and the client character set of the session. For details, see "Supported Translations Between Character Sets" on page 1237.
_encoding	an optional literal for translating from KANJI1 to UNICODE that indicates a specific encoding of KANJI1. The *_encoding* option is not allowed if LOCALE is specified for *source_repertoire_name* or *target_repertoire_name*.

Syntax element ...	Specifies ...
_encoding (continued)	If the translation is from the following character sets: • KatakanaEBCDIC • KanjiEBCDIC5026_0I • KanjiEBCDIC5038_0I use the following value for *_encoding*: _KanjiEBCDIC KanjiEUC_0U, use the following value for *_encoding*: _KanjiEUC KanjiShiftJIS_0S, us the following value for *_encoding*: _KANJISJIS ASCII or EBCDIC, use the following value for *_encoding*: _SBC
target_repertoire_name	the target character set of the string to translate. For supported values, see "Supported Translations Between Character Sets" on page 1237. A value of LOCALE can be specified for *target_repertoire_name* to translate a character string from UNICODE to LATIN or KANJI1 using a target repertoire determined by the language support mode of the system and the client character set of the session. For details, see "Supported Translations Between Character Sets" on page 1237.
_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 • _PadGraphic • _KanjiEUC • _Fullwidth • _KANJISJIS • _Halfwidth • _SBC • _FoldSpace • _PadSpace • _VarGraphic The *_suffix* option is not allowed if LOCALE is specified for *source_repertoire_name* or *target_repertoire_name*.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Argument Types

Use TRANSLATE_CHK on character strings and character string expressions.

By default, Teradata Database performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including TRANSLATE_CHK, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 13: "Data Type Conversions."

Result Type and Attributes

Default attributes for TRANSLATE_CHK (*string* USING *source*_TO_*target*) are:

Data Type	Heading
INTEGER	Translate_Chk(*string* using *source*_to_*target*)

Result Values

Value	Meaning
0	The string can be translated without error.
anything else	The position of the first character in the string causing a translation error. The value is a logical position for arguments of type LATIN, UNICODE, KANJISJIS, and GRAPHIC. The value is a physical position for arguments of type KANJI1.

Example 1

Function	Result
TRANSLATE_CHK('abc' USING UNICODE_TO_LATIN)	0
TRANSLATE_CHK('abc年' USING UNICODE_TO_LATIN)	4

Example 2

Consider the following table definition:

```
CREATE TABLE table_1
  (cunicode CHARACTER(64) CHARACTER SET UNICODE);
```

To find all values in cunicode that can be translated to LATIN, use the following statement:

```
SELECT cunicode
FROM table_1
```

```
WHERE TRANSLATE_CHK(cunicode USING Unicode_TO_Latin) = 0;
```

Example 3

Consider the following table definitions:

```
CREATE TABLE table_1
  (ckanji1 VARCHAR(20) CHARACTER SET KANJI1);

CREATE TABLE table_2
 (cunicode CHARACTER(20) CHARACTER SET UNICODE);
```

Assume table_1 is populated from the KanjiEUC client character set.

To translate the data in ckanji1 in table_1 to UNICODE, and populate table_2 with translations that have no errors, use the following statement:

```
INSERT INTO table_2
SELECT TRANSLATE(ckanji1 USING Kanji1_KanjiEUC_TO_Unicode)
FROM table_1
WHERE TRANSLATE_CHK(ckanji1 USING Kanji_KanjiEUC_TO_Unicode) = 0;
```

Example 4

After converting column ckanji1 in table_1 to column cunicode in table_2, you want to find all the fields in table_1 that could not be translated.

```
SELECT ckanji1
FROM table_1
WHERE TRANSLATE_CHK(ckanji1 USING Kanji1_KanjiEUC_TO_Unicode) <> 0;
```

Checking UNICODE Normalization Form Translations

When using TRANSLATE_CHK to verify UNICODE normalization form translations, any valid Unicode string can be translated to any of the normalization forms. A successful result (0) is expected unless the compatibility ideographs U+FA6C, U+FACF, U+FAD0, U+FAD1, U+FAD5, U+FAD6, or U+FAD7 are present, because these characters normalize outside the BMP, that is, outside the range U+10000 to U+10FFF.

For all normalization forms, these characters normalize as follows:

Ideograph	Normalized Form
U+FA6C	U+242EE
U+FACF,	U+2284A
U+FAD0	U+22844
U+FAD1	U+233D5
U+FAD5	U+25249
U+FAD6	U+25CD0
U+FAD7	U+27ED3

TRIM

Purpose

Takes a character or byte *string_expression* argument, trims the specified pad characters or bytes, and returns the trimmed string.

Syntax

where:

Syntax Element ...	Specifies ...
BOTH TRAILING LEADING	how to trim the specified trim character or byte from *string_expression*. • BOTH means trim both trailing and leading characters or bytes. • TRAILING means trim only trailing characters or bytes. • LEADING mens trim only leading characters or bytes. If you omit this option, the default is BOTH, and the default trim character is a null byte for byte types and a pad character for character types.
trim_expression	the specific character or byte to trim from the head, tail, or both, of *string_expression*. The expression must evaluate to a single character. You cannot specify *trim_expression* without also specifying BOTH, TRAILING, or LEADING. You cannot specify a *trim_expression* of type KANJI1, nor can you apply a *trim_expression* to a *string_expression* of type KANJI1.
FROM	a keyword required when BOTH, TRAILING, or LEADING are specified.
character_set	the name of the server character set to associate with the string expression.
character_set (*continued*)	Valid values are: _Latin, which is the LATIN server character set _Unicode, which is the UNICODE server character set _KanjiSJIS, which is the KANJISJIS server character set _Graphic, which is the GRAPHIC server character set
string_expression	a byte or character string or string expression to be trimmed.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Argument Types and Rules

The *trim_expression* argument must evaluate to a single byte that has a byte data type or single character that has a character data type.

TRIM operates on the following types of *string_expression* arguments:

- Character, except for CLOB
- Byte, except for BLOB
- Numeric

  If a numeric expression is used as the *string_expression* argument, it is converted implicitly to CHARACTER type.
- UDTs that have implicit casts to any of the following predefined types:
  - Character
  - Numeric
  - Byte
  - DATE

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

  Implicit type conversion of UDTs for system operators and functions, including TRIM, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 13: "Data Type Conversions."

Result Type and Attributes

Here are the default result type and attributes for TRIM(*string_expression*):

- If *string_expression* is a byte string, the result type is VARBYTE.
- If the *string_expression* is a numeric expression or character string, the result type is VARCHAR.

It is possible for the length of the result to be zero.

The server character set of the result is the same as the argument.

If the *string_expression* argument is null, the result is null.

Concatenation With TRIM

The TRIM function is typically used with the concatenation operator to remove trailing pad characters or trailing bytes containing binary 00 from the concatenated string.

If the TRIM function is specified for character data types, leading, trailing, or leading and trailing pad characters are suppressed in the concatenated string, according to which syntax is used.

Example 1

If the Names table includes the columns first_name and last_name, which contain the following information:

```
first_name (CHAR(12)) has a value of 'Mary       '
last_name (CHAR(12)) has a value of  'Jones      '
```

then this statement:

```
SELECT TRIM (BOTH FROM last_name) || ', ' || TRIM(BOTH FROM
first_name)
FROM names ;
```

returns the following string (note that the seven trailing blanks at the end of string Jones, and the eight trailing blanks at the end of string Mary are not included in the result):

```
'Jones, Mary'
```

If the TRIM function is removed, the statement:

```
SELECT last_name || ', ' || first_name
FROM names;
```

returns trailing blanks in the string:

```
'Jones       , Mary        '
```

Example 2

Assume column a is BYTE(4) and column b is VARBYTE(10).

If these columns contained the following values:

```
a             b
------------  ---------
78790000      43440000
68690000      3200
12550000      332200
```

then this function:

```
SELECT TRIM (TRAILING FROM a) || TRIM (TRAILING FROM b) FROM ...
```

returns:

```
78794344
686932
12553322
```

Example 3

The following statement trims trailing SEMICOLON characters from the specified string.

```
SELECT TRIM( TRAILING ';' FROM textfield) FROM texttable;
```

Example 4

The following table illustrates several more complicated TRIM functions:

Function	Result
SELECT TRIM(LEADING 'a' FROM 'aaabcd');	'bcd'
CREATE TABLE t2 (i1 INTEGER, c1 CHAR(6), c2 CHAR(1)); INSERT t2 (1, 'aaabcd', 'a'); SELECT TRIM(LEADING c2 FROM c1) FROM t2;	'bcd'
CREATE TABLE t3 (i1 INTEGER, c1 CHAR(6) CHAR SET UNICODE); INSERT t3 (1, _Unicode '006100610061006200630064'XC); SELECT TRIM(LEADING _Unicode '0061'XC FROM t3.c1);	'bcd'
SELECT TRIM(_Unicode 'ΔΔabc年ΔΔΔ');	'abc年'
SELECT TRIM(_Unicode 'ΔΔabc年ΔΔΔ');	'abc年ΔΔ' Δ (GRAPHIC pad) is not removed.
CREATE TABLE t1 (c1 CHARACTER(6) CHARACTER SET GRAPHIC); INSERT t1 (_Graphic 'abc年ΔΔ'); SELECT TRIM(c1) from t1;	'abc年' Δ (GRAPHIC pad) is removed because the operand of the TRIM function is of type GRAPHIC.

UPPER

Purpose

Returns a character string identical to *character_string_expression*, except that all lowercase letters are replaced by their uppercase equivalents.

Syntax

```
—— UPPER —( character_string_expression) ———
```

FF07D258

where:

Syntax element ...	Specifies ...
character_string_expression	a character string or character string expression for which all lowercase characters are to be replaced by their uppercase equivalents.

ANSI Compliance

This is ANSI SQL:2011 compliant.

Argument Types

UPPER is valid only for character strings and character string expressions. The function does not accept CLOBs and non-character arguments.

By default, Teradata Database performs implicit type conversion on UDT arguments that have implicit casts to predefined character types.

To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

Implicit type conversion of UDTs for system operators and functions, including UPPER, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion of UDTs, see Chapter 13: “Data Type Conversions.”

Result Type and Attributes

Here are the default result type and attributes for UPPER(*arg*):

Data Type	Heading
Same type as *arg*	Upper(*arg*)

Usage Notes

The UPPER function allows users who want ANSI portability to have case blind comparisons with ANSI-compliant syntax.

This function is treated the same as the following obsolete form:

```
expression (UPPERCASE)
```

You can also replace characters with lowercase equivalents. For more information, see "LOWER" on page 1191.

Restrictions

UPPER does not convert multibyte characters to uppercase in the KANJI1 server character set.

Example 1

Consider the following table definition where the character columns have CASESPECIFIC attributes:

```
CREATE TABLE employee
  (last_name CHAR(32) CASESPECIFIC
  ,city      CHAR(32) CASESPECIFIC
  ,emp_id    CHAR(9)  CASESPECIFIC
  ,emp_ssn   CHAR(9)  CASESPECIFIC);
```

To compare on a case blind basis:

```
SELECT emp_id
FROM employee
WHERE UPPER(emp_id) = UPPER(emp_ssn);
```

To compare with a string literal:

```
SELECT emp_id
FROM employee
WHERE UPPER(city) = 'MINNEAPOLIS';
```

Teradata SQL also has the data type attribute NOT CASESPECIFIC, which allows case blind comparisons. Note that the data type attributes CASESPECIFIC and NOT CASESPECIFIC are Teradata extensions to the ANSI standard.

Example 2

The use of UPPER to store values is shown in the following examples:

```
INSERT INTO names
SELECT UPPER(last_name),UPPER(first_name)
   FROM newnames;
```

or

```
USING (last_name CHAR(20),first_name CHAR(20))
INSERT INTO names
(UPPER(:last_name), UPPER(:first_name));
```

Example 3

This example shows that in the KANJI1 server character set, only single byte characters are converted to uppercase.

```
SELECT UPPER('abcd年');
```

The result is 'ABCD年'.

VARGRAPHIC

Purpose

Returns the VARGRAPHIC representation of the character data in *character_string_expression*.

Syntax

```
── VARGRAPHIC ── ( character_string_expression ) ──
```

1101E197

where:

Syntax element ...	Specifies ...
character_string_expression	a character string or character string expression for which the VARGRAPHIC representation is to be returned.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Argument Types

VARGRAPHIC operates on the following types of arguments:

- Character, except for CLOB
- Numeric

  If the argument is numeric, it is implicitly converted to a character type.
- UDTs that have implicit casts to any of the following predefined types:
  - Character
  - Numeric
  - DATE

  To define an implicit cast for a UDT, use the CREATE CAST statement and specify the AS ASSIGNMENT clause. For more information on CREATE CAST, see *SQL Data Definition Language*.

  Implicit type conversion of UDTs for system operators and functions, including VARGRAPHIC, is a Teradata extension to the ANSI SQL standard. To disable this extension, set the DisableUDTImplCastForSysFuncOp field of the DBS Control Record to TRUE. For details, see *Utilities*.

For more information on implicit type conversion, see Chapter 13: “Data Type Conversions.”

Result Type and Attributes

Here are the default result type and attributes for VARGRAPHIC(*arg*):

Data Type	Heading
VARCHAR(*n*) CHARACTER SET GRAPHIC	Vargraphic(*arg*)

Rules

VARGRAPHIC reports an error if the session character set is UTF8 or a single-byte character set, such as ASCII. If the argument is of type KANJI1, the only valid session character set is KanjiEBCDIC.

All characters in the string are converted into one or more graphics that are valid for the character set of the current session. For details, see "VARGRAPHIC Function Conversion Tables" on page 1258.

The argument cannot be of type GRAPHIC.

A result that exceeds the maximum length of a VARCHAR CHARACTER SET GRAPHIC data type generates an error.

VARGRAPHIC cannot appear as the first argument in a user-defined method invocation.

Specific rules apply to the server character set of *character_string_expression*.

IF the string specifies this server character set ...	THEN VARGRAPHIC operates as follows ...
KANJI1	Shift-Out/Shift-In characters in the *character_string_expression* do not appear in the result string. They are required only to indicate the transition between single byte characters and multibyte characters. Improperly placed Shift-Out/Shift-Ins are replaced by the illegal character for the character set of the session. The SPACE CHARACTER translates to the IDEOGRAPHIC SPACE CHARACTER.
UNICODE	• Characters with fullwidth representation in the UNICODE compatibility zone translate to that fullwidth representation. • Halfwidth characters from the compatibility zone translate to the corresponding characters outside the compatibility zone. • The SPACE CHARACTER translates to the IDEOGRAPHIC SPACE CHARACTER. • The control characters U+0000 - U+001F and character U+007F are converted to the VARGRAPHIC error character. • Other characters are left untranslated.
anything else	The result is as if string were first converted to UNICODE and then translated according to the rules listed for UNICODE above.

Notice: In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

For more information, see “KANJI1 Character Set” in *International Character Set Support*.

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**abc**Δ'
VARGRAPHIC('abc')	'**abc**'

Example 2

Consider the following table definition with two character columns that use the KANJI1 server character set:

```
CREATE TABLE t1
  (c1 VARCHAR(12) CHARACTER SET KANJI1
  ,c2 VARCHAR(12) CHARACTER SET KANJI1);
```

Use the KanjiEBCDIC client character set and insert the following strings:

```
INSERT t1 ('def', 'gH<ABC>X');
```

Convert the strings to GRAPHIC data:

Function	Result
SELECT VARGRAPHIC (c1) FROM t1;	'def'
SELECT VARGRAPHIC (c2) FROM t1;	'gH**ABCX**' (The single byte Hankaku Katakana **X** is converted to double byte **X**.)

VARGRAPHIC Function Conversion Tables

The following table shows the translation of a single byte character to its double byte equivalent by the VARGRAPHIC function. Values in columns 2, 3, and 4 are hexadecimal. (Also see the notes following the table.)

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
00		FEFE	FEFE
01		FEFE	FEFE
02		FEFE	FEFE
03		FEFE	FEFE
04		FEFE	FEFE
05		FEFE	FEFE
06		FEFE	FEFE
07		FEFE	FEFE
08		FEFE	FEFE
09		FEFE	FEFE
0A		FEFE	FEFE
0B		FEFE	FEFE
0C		FEFE	FEFE
0D		FEFE	FEFE
0E[a]		N/A	N/A
0F[b]		FEFE	FEFE
10		FEFE	FEFE
11[c]	£	424A	424A
12[d]	¬	425F	FEFE
13	\	43E0	FEFE
14	~	43A1	FEFE
15		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
16		FEFE	FEFE
17		FEFE	FEFE
18		FEFE	FEFE
19		FEFE	FEFE
1A		FEFE	FEFE
1B		FEFE	FEFE
1C		FEFE	FEFE
1D		FEFE	FEFE
1E		FEFE	FEFE
1F		FEFE	FEFE
20		4040	4040
21	!	425A	425A
22	"	4472	4472
23	#	427B	427B
24	$	42E0	42E0
25	%	426C	426C
26	&	4250	4250
27	'	4471	4471
28	(	424D	424D
29	)	425D	425D
2A	*	425C	425C
2B	+	424E	424E
2C	,	426B	426B
2D	-	4260	4260
2E	.	424B	424B
2F	/	4261	4261
30	0	42F0	42F0
31	1	42F1	42F1
32	2	42F2	42F2

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
33	3	42F3	43F3
34	4	42F4	42F4
35	5	42F5	42F5
36	6	42F6	42F6
37	7	42F7	42F7
38	8	42F8	42F8
39	9	42F9	42F9
3A	:	427A	427A
3B	;	425E	425E
3C	<	424C	424C
3D	=	427E	427E
3E	>	426E	426E
3F	?	426F	426F
40	@	427C	427C
41	A	42C1	42C1
42	B	42C2	42C2
43	C	42C3	42C3
44	D	42C4	42C4
45	E	42C5	42C5
46	F	42C6	42C6
47	G	42C7	42C7
48	H	42C8	42C8
49	I	42C9	42C9
4A	J	42D1	42D1
4B	K	42D2	42D2
4C	L	42D3	42D3
4D	M	42D4	42D4
4E	N	42D5	42D5
4F	O	42D6	42D6

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
50	P	42D7	42D7
51	Q	42D8	42D8
52	R	42D9	42D9
53	S	42E2	42E2
54	T	42E3	42E3
55	U	42E4	42E4
56	V	42E5	42E5
57	W	42E6	42E6
58	X	42E7	42E7
59	Y	42E8	42E8
5A	Z	42E9	42E9
5B	[	4444	FEFE
5C	\	425B	425B
5D	]	4445	FEFE
5E	^	4470	425F
5F	_	426D	426D
60	`	4279	FEFE
61	a	4281	FEFE
62	b	4282	FEFE
63	c	4283	FEFE
64	d	4284	FEFE
65	e	4285	FEFE
66	f	4286	FEFE
67	g	4287	FEFE
68	h	4288	FEFE
69	i	4289	FEFE
6A	j	4291	FEFE
6B	k	4292	FEFE
6C	l	4293	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
6D	m	4294	FEFE
6E	n	4295	FEFE
6F	o	4296	FEFE
70	p	4297	FEFE
71	q	4298	FEFE
72	r	4299	FEFE
73	s	42A2	FEFE
74	t	42A3	FEFE
75	u	42A4	FEFE
76	v	42A5	FEFE
77	w	42A6	FEFE
78	x	42A7	FEFE
79	y	42A8	FEFE
7A	z	42A9	FEFE
7B	{	42C0	FEFE
7C	|	424F	424F
7D	}	42D0	FEFE
7E	-[e]	42A1	42A1
7F		FEFE	FEFE
80		FEFE	FEFE
81		FEFE	FEFE
82		FEFE	FEFE
83		FEFE	FEFE
84		FEFE	FEFE
85		FEFE	FEFE
86		FEFE	FEFE
87		FEFE	FEFE
88		FEFE	FEFE
89		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
8A		FEFE	FEFE
8B		FEFE	FEFE
8C		FEFE	FEFE
8D		FEFE	FEFE
8E		FEFE	FEFE
8F		FEFE	FEFE
90		FEFE	FEFE
91		FEFE	FEFE
92		FEFE	FEFE
93		FEFE	FEFE
94		FEFE	FEFE
95		FEFE	FEFE
96		FEFE	FEFE
97		FEFE	FEFE
98		FEFE	FEFE
99		FEFE	FEFE
9A		FEFE	FEFE
9B		FEFE	FEFE
9C		FEFE	FEFE
9D		FEFE	FEFE
9E		FEFE	FEFE
9F		FEFE	FEFE
A0		FEFE	FEFE
A1	f	4341	4341
A2	g	4342	4342
A3	h	4343	4343
A4	i	4344	4344
A5	j	4345	4345
A6	k	4346	4346

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
A7	l	4347	4347
A8	m	4348	4348
A9	n	4349	4349
AA	o	4351	4351
AB	p	4352	4352
AC	q	4353	4353
AD	r	5454	4354
AE	s	4355	4355
AF	t	4356	4356
B0	u	4358	4358
B1	A	4381	4381
B2	I	4382	4382
B3	U	4383	4383
B4	E	4384	4384
B5	O	4385	4385
B6	KA	4386	4386
B7	KI	4387	4387
B8	KU	4388	4388
B9	KE	4389	4389
BA	KO	438A	438A
BB	SA	438C	438C
BC	SHI	438D	438D
BD	SU	438E	438E
BE	SEE	438F	438F
BF	SO	4390	4390
C0	TAI	4391	4391
C1	CHI	4392	4392
C2	TSU	4393	4393
C3	TE	4394	4394

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
C4	TO	4395	4395
C5	NA	4396	4396
C6	NI	4397	4397
C7	NU	4398	4398
C8	NE	4399	4399
C9	NO	439A	439A
CA	HA	439D	439D
CB	HI	439E	439E
CC	FU	439F	439F
CD	HE	43A2	43A2
CE	HO	43A3	43A3
CF	MA	43A4	43A4
D0	MI	43A5	43A5
D1	MU	43A6	43A6
D2	ME	43A7	43A7
D3	MO	43A8	43A8
D4	YA	43A9	43A9
D5	YU	43AA	43AA
D6	YO	43AC	43AC
D7	RA	43AD	43AD
D8	RI	43AE	43AE
D9	RU	43AF	43AF
DA	RE	43BA	43BA
DB	RO	43BB	43BB
DC	WA	43BC	43BC
DD	N	43BD	43BD
DE	v	43BE	43BE
DF	w	43BF	43BF
E0		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
E1		FEFE	FEFE
E2		FEFE	FEFE
E3		FEFE	FEFE
E4		FEFE	FEFE
E5		FEFE	FEFE
E6		FEFE	FEFE
E7		FEFE	FEFE
E8		FEFE	FEFE
E9		FEFE	FEFE
EA		FEFE	FEFE
EB		FEFE	FEFE
EC		FEFE	FEFE
ED		FEFE	FEFE
EE		FEFE	FEFE
EF		FEFE	FEFE
F0		FEFE	FEFE
F1		FEFE	FEFE
F2		FEFE	FEFE
F3		FEFE	FEFE
F4		FEFE	FEFE
F5		FEFE	FEFE
F6		FEFE	FEFE
F7		FEFE	FEFE
F8		FEFE	FEFE
F9		FEFE	FEFE
FA		FEFE	FEFE
FB		FEFE	FEFE
BC		FEFE	FEFE
FD		FEFE	FEFE

Single Byte Character		Double Byte Equivalent	
JIS Internal Code	JIS X 0201 Printable Character	KanjiEBCDIC 5026/ 5035	Katakana EBCDIC
FE		FEFE	FEFE
FF		FEFE	FEFE

a. For KanjiEBCDIC, the SO/SI is not placed in the output of vargraphic function. In particular, a single SO character will not generate any output, or strictly speaking will generate a string with 0 length

b. For KanjiEBCDIC, the SO/SI is not placed in the output of vargraphic function. However, if the SI character appears in the input without matching SO, we will generate FEFE for that SI.

c. Pound Sterling sign

d. Logical NOT

e. Overline

f. Ideographic period

g. Left corner bracket

h. Right corner bracket

i. Ideographic comma

j. Katakana middle dot

k. Katakana letter WO

l. Katakana letter A

m. Katakana letter small I

n. Katakana letter small U

o. Katakana letter small E

p. Katakana letter small O

q. Katakana letter small YA

r. Katakana letter small YU

s. Katakana letter small YO

t. Katakana letter small WO

u. Katakana-Hiragana prolonged sound mark

v. Katakana-Hiragana voiced sound mark

w. Katakana-Hiragana semi-voice sound mark

CHAPTER 26 Table Operators

Table operators accept a table or table expression as input and generate a table as output. They are a type of user-defined function (UDF). Table operators are used only in the FROM clause of SELECT statements.

Table operators enable a simplified in-database MapReduce-style programming model.You cannot use scalar subqueries in table operators with an ON clause using PARTITION BY or HASH BY.

CALCMATRIX

Purpose

Calculates a sum-of-squares-and-cross-products (SSCP) matrix.

Syntax

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB.	the name of the database where the function is located.
tableName	the name of a table with the input data for the function.
query_expression	the SELECT statement with input data for the function.
hashByColList	the column name set by which rows are to be hashed across the AMPs before being passed to the operator.
localOrderByList	the column name set by which rows are to be value-ordered locally on the AMPs before being passed to the operator.
ASC	that the results are to be ordered in ascending sort order. If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session. The default order is ASC.

Syntax element ...	Specifies ...
DESC	that the results are to be ordered in descending sort order. If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.
NULLS FIRST	that NULL results are to be listed first.
NULLS LAST	that NULL results are to be listed last.
PHASE	one of the phases of the function: Valid values are: • 'LOCAL' • 'COMBINE' For a definition of LOCAL and COMBINE, see "Usage Notes" on page 1272.
CALCTYPE	the type of matrix CALCMATRIX creates. Valid values are: • 'SSCP' = sum-of-squares-and-cross-products matrix (the default) • 'ESSCP'= Extended-sum-of-squares-and-cross-products matrix • 'CSSCP' = Corrected-sum-of-squares-and-cross-products matrix • 'COV' = Covariance matrix • 'COR' = Correlation matrix
OUTPUT	the format of the output: Valid values are: • 'COLUMNS' • 'VARBYTE'
NULL_HANDLING	how nulls that appear as in any data columns are handled. Valid values are: • 'ZERO' = any NULL values are considered to be 0, and the row is processed (default). • 'IGNORE' = the row is skipped.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Data Type

The data type of the data columns must be some numeric data type.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

Usage Notes

Note: CALCMATRIX processes data representing a single input matrix or multiple input matrices. In the latter case, the input must have column(s) that identify which matrix a given input row belongs to. These columns are called as *matrix id columns* and the value of one tuple of matrix id columns is called a *matrix id*. All rows from the same matrix must have the same matrix id.

CALCMATRIX works in LOCAL and COMBINE phases.

- **In the LOCAL phase**, the function processes the input data that exists in the source table on each amp separately and outputs rows of summarized data for that amp. No data has to be moved among amps.
- **In the COMBINE phase**, the function processes the summarized data from each amp (that is, the output from the LOCAL phase) and combines it to form the rows that make up final output.

  In order to get one result matrix (or, if there are matrix id columns, one result matrix for each matrix id), the COMBINE phase must use a HASH BY clause and possibLY a LOCAL ORDER BY clause.

LOCAL Phase Details

- The input to the LOCAL phase must consist of zero or more matrix id columns, and 1 or more data columns (call this number n). The matrix id columns, if present, must be placed in the LOCAL ORDER BY list.
- The output from the LOCAL phase consists of the matrix id columns, if any, an INT column named *rownum*, a VARCHAR(128) column named *rowname*, a BIGINT column named *c* (for count), a REAL column named *s* (for sum), and a REAL column for each data column of input.
- The CALCTYPE and OUTPUT custom values in the LOCAL phase are always set to 'SSCP' and 'COLUMNS' respectively. Attempts to use different values for these keys are ignored.
- If any of the data columns in a row have NULL values, the row's handling depends on the setting for NULL_HANDLING.

COMBINE Phase Details

- The input to the COMBINE phase must be exactly the columns that are output from the LOCAL phase: zero or more matrix id columns, an INT *rownum* column, a VARCHAR(128) rowname column, a BIGINT column named *c* (for count), a REAL column names *s* (for sum), and a REAL column for each data column of input.

  If there are matrix id columns in the data, they should be specified in the HASH BY and LOCAL ORDER BY lists. If not, a column with a constant value and an alias name, like "1 as p", should be added to the ON clause and the HASH BY list.
- The custom values for CALCTYPE and OUTPUT control the output columns from the COMBINE phase.
- If the OUTPUT is set to 'COLUMNS', then the columns output from the 'COMBINE' phase are: The matrix id columns, if present, an INT *rownum* column, a VARCHAR(128)

rowname column, a REAL column for each data column of input and, if the CALCTYPE is 'ESSCP', a BIGINT column named *c* (for count), a REAL column named *s* (for sum).

If the OUTPUT is set to 'VARBYTE', then the columns output from the 'COMBINE' phase are: an INT *rowname* column, a VARCHAR(128) *rowname* column, and a VARBYTE column named *v*.

- If there are no matrix id columns, the final output of the COMBINE phase consists of n rows, where *n* is the number of data columns in the input. The rownum field has values from 1 to n. Thus each output row is one row from the resulting matrix. If there are matrix id columns, the final output of the COMBINE phase consists of n rows per matrix id. The rownum field will have values from 1 to n.

VARBYTE Output

When the OUTPUT is 'VARBYTE', the output row consists of the matrix id columns, if any, and these fields:

- INT rownum
- VARCHAR(128) rowname
- VARBTYE *value*

Details for the Both the LOCAL and the COMBINE Phases

- The values for the *rownum* column ranges from 1 to n, the number of data columns in the input. If the *rownum* column has value x, the *rowname* column is the x'th data column name.
- If the columns in the ON clause are expressions, they must have an alias name. Columns in the LOCAL ORDER BY or HASH BY lists must be column or alias names from the ON clause, or ordinal values.

Example 1: One Input Matrix, Single SQL Statement

In the following example, table T has 1 billion rows of weather data. It has 1 geometry column representing the point (latitude/longitude) of the observation, and these REAL columns:

- temperature
- air_pressure
- rainfall.

Use an SQL statement that invokes CALCMATRIX twice, once for the LOCAL phase and once for the COMBINE phase, as follows:

```
select * from CALCMATRIX( ON (
select 1 as p, X.* from CALCMATRIX( ON (select temperature,
   air_pressure, rainfall from T)
USING PHASE('LOCAL') )X )
HASH BY p
USING PHASE('COMBINE')
)Y;
```

The result set would be the SSCP matrix for the weather data. The following table shows the result set. *n* means the numeric result of the calculation.

rownum	rowname	temperature	air_pressure	rainfall
1	temperature	*n*	*n*	*n*
2	air_pressure	*n*	*n*	*n*
3	rainfall	*n*	*n*	*n*

Example 2: One Input Matrix, Multiple SQL Statements

In the following example, table T has the same weather data as table T in "Example 1: One Input Matrix, Single SQL Statement." However, the aim this time is to produce both the covariance and correlation matrices. In order to avoid 2 passes over billion rows of raw data, insert the results of CALCMATRIX using the LOCAL phase into an intermediate table. Then run CALCMATRIX using the COMBINE phase twice, but against the intermediate table, which only has 3 * (the number of amps in the system) rows.

```
CREATE TABLE T_INTERMEDIATE as (
select 1 as p, X.* from CALCMATRIX( ON (select temperature,
   air_pressure, rainfall from T) USING PHASE('LOCAL') )X
) WITH DATA;

select * from CALCMATRIX ( ON T_INTERMEDIATE
HASH BY p
USING PHASE('COMBINE') CALCTYPE('COV')
)Y;

select * from CALCMATRIX( ON T_INTERMEDIATE
HASH BY p
USING PHASE('COMBINE') CALCTYPE('COR')
)Y;
```

Example 3: Multiple Input Matrices, One SQL Statement

In the following example, table T has the same weather data as table T in "Example 1: One Input Matrix, Single SQL Statement" and "Example 2: One Input Matrix, Multiple SQL Statements." In this example, in addition to the point location of the observation, there is an additional VARCHAR column for continent, which is the matrix id. Our aim is to produce 7 SSCP matrices, one for each continent's data.

Use an SQL statement that invokes CALCMATRIX twice, once for the LOCAL phase and once for the COMBINE phase.

```
select * from calcmatrix( ON (
  select * from calcmatrix( ON (select continent, temperature,
  air_pressure, rainfall from T) LOCAL ORDER BY continent USING
  PHASE('LOCAL') )X )
  HASH BY continent
  LOCAL ORDER BY continent
  USING PHASE('COMBINE')
  )Y
  order by continent, rownum;
```

The result set in the following tables is the SSCP matrix for weather data by continent.

Note: Only 2 of the matrices are shown below.

continent	rownum	rowname	temperature	air_pressure	rainfall
Europe	1	temperature	*n*	*n*	*n*
Europe	2	air_pressure	*n*	*n*	*n*
Europe	3	rainfall	*n*	*n*	*n*

continent	rownum	rowname	temperature	air_pressure	rainfall
Asia	1	temperature	*n*	*n*	*n*
Asia	2	air_pressure	*n*	*n*	*n*
Asia	3	rainfall	*n*	*n*	*n*

. . .

LOAD_FROM_HCATALOG (Teradata SQL-H)

Purpose

Retrieves data from a Hadoop distributed database into Teradata Database, where the data can be placed in tables or joined with existing tables.

Syntax

where:

Syntax element ...	Specifies ...
ON *tablename* / ON *query_expression*.	the clause may be passed to LOAD_FROM_HCATALOG; however, for this table operator the clause is ignored
name value	name value pairs that specify parameters for the Hadoop data import, such as the identifier of the HCatalog server for the Hadoop database, database and table names, and specific columns and partitions to be imported. The name value data that can be specified for LOAD_FROM_HCATALOG are listed below. Each name is paired with a single string value enclosed in apostrophe delimiters.

Name Value Pairs

The following name value pairs can be specified with LOAD_FROM_HCATALOG. The names and values are not case-sensitive. Each value constitutes a single string, so must be enclosed within apostrophe characters.

name('*value*')	Description
server('*hcatalog_server*')	The HCatalog server hostname or IP address. The server returns table/file metadata about the column types and splits. This name value pair is required.
port('*port_number*')	The port to use for HCatalog access. This name value pair is required.

name('*value*')	Description
dbname('*hcatalog_dbname*')	The name of the HCatalog database where the data to be imported resides. Note: Database names may not be accessible if they contain supplementary characters. Unicode delimited identifier and Unicode Character String Literal syntax is not supported.
tablename('*tablename*')	The name of the HCatalog table to be imported. Note: Table names may not be accessible if they contain supplementary characters. Unicode delimited identifier and Unicode Character String Literal syntax is not supported.
username('*username*')	The name of the user to use when making the connection to Hadoop. There is no password associated with a Hadoop user. If the Hadoop Distributed File System (HDFS) is not configured for file permissions, this name value pair is optional. Note: Any value can be placed in the username clause. If the proxy logon on the Hadoop side accepts the username, any data under that username is accessible by any Teradata user with permission to execute LOAD_FROM_HCATALOG. To provide a greater level of security, place LOAD_FROM_HCATALOG in a view with a specific username. Access to the view may be granted to specific users, who would then be limited to the static table, columns, and partition filter defined within the view.
columns('*column_list*')	The columns from each imported row that will be added to the output spool area. Only these columns of the imported table will be available to the SELECT statement and WHERE clause. column_list is a list of column names separated by commas. For example: `columns('name,color')` Use a value of `'*'` to retrieve all columns from the specified database. Note: Column names may not be accessible if they contain supplementary characters. Unicode delimited identifier and Unicode Character String Literal syntax is not supported.
column_sizes('*column_size_list*')	Can be used to limit the string and varchar data returned for specified columns. Data in excess of the specified sizes is silently truncated. *column_size_list* is a list of column size specifications that each consist of the column name, an equal sign, and an integer representing the number of characters that should be returned for values of that column. names and the number of characters that should be returned for that column. Column specifications in the list are separated by semicolons. For example: `column_sizes('name=20;color=8')`
partitionfilter('*partitionfilter*')	Limits the data that is imported from a Hadoop partitioned column to only data in a specified partition. The specified column must be a partitioned column. Only string columns can be partitioned. partitionfilter is formatted as the column name, an equal sign, and a double-quote delimited string for the partition value. Multiple partition filters can be separated by operators AND, OR, LIKE, (), = <>, <,>,<=, and >=. Note that the LIKE operator is not full-featured SQL, and does not support % matching. For example: `partitionfilter('doors="4" AND make="Ford"')`

name('*value*')	Description
transformformatting('true')	Reformats Hadoop Array type into a comma separate format compatible with casting into Teradata Database ARRAY data type columns.
templeton__port('*templeton_port*')	The port used to access a Templeton server. Can be used with a REST-like web API to receive metadata about the HCatalog database and tables. Note: The Templeton service must be available during execution of LOAD_FROM_HCATALOG.
hadoop_properties ('<*property1=value*>,<*property3=value1,value2*>')	Sets specific properties used to interact with Hadoop. It is considered a best practice to delimit hadoop_properties name value pairs with left and right angle brackets (< and >). The following rules apply: • If there are multiple arguments separated by commas, you must delimit them with angle brackets. • If you use angle brackets to delimit one hadoop_properties name value pair in the clause, you must use angle brackets to delimit all hadoop_properties name value pairs in the clause. • For the purposes of backwards compatibility, you can omit angle brackets if your name value pairs do not have multiple values and do not contain commas. The syntax for the hadoop_properties clause for a High Availability (HA) target supports an updated syntax where multiple values can be included. In this case, the properties must be delimited by left and right angle brackets. For more information, see "Using hadoop_properties" on page 1281.
isnested ('true')	A flag to indicate that the table operator is being nested. When you use LOAD_FROM_HCATALOG in nested SQL syntax, use this name value pair to indicate that the inner operator is in nested mode. Nested mode supports unbounded data transfer. If you do not use the isnested flag, the amount of data that can be retrieved from Hadoop is limited. Note: Do not use the isNested name value pair when the syntax is not nested (that is, when there is no outer table operator), because the flag causes the table operator to query the meta data and to return results from the meta data spool instead.

At a minimum, you should specify the following name value pairs:

- server
- port
- dbname
- tablename
- templeton_port

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Data Types

The following table shows how Hadoop data types are mapped to Teradata Database data types.

Hadoop Data Type	Teradata Database Data Type
String	VARCHAR UNICODE CHARSET
Boolean	BYTEINT
Integer	INT
BigInt	BIGINT
Float	FLOAT
Double	FLOAT
BINARY	VARBYTE
MAP	VARCHAR UNICODE CHARSET
Struct	VARCHAR UNICODE CHARSET
ARRAY	VARCHAR UNICODE CHARSET
TINYINT	BYTEINT
SMALLINT	SMALLINT

Note: VARCHAR output column size is 2048 characters unless the number of columns and the sizes of the non-VARCHAR data, including the row header, exceed 64 K bytes. You can use the optional column_sizes clause to override the default size. The Hadoop string data type does not include a length, which requires that the mapping take a best-fit approach. Strings are silently truncated.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

Supported File Types and Compression Types

Teradata SQL-H supports the file types and compression types listed in the following table:

Distribution	Supported Import File Types	Compression Supported By File Type

Hortonworks HDP 1.1.0.17, 1.3.2, and 2.1 on Teradata Distribution for Hadoop (TDH)	• TextFile
	• RCFile
	• ORCFile
	
	Note: ORCFile is supported for HDP 1.3.2 and 2.1 only.
		• Compression types supported for use with TextFile:
		  • DefaultCodec
		  • BZip2Codec
		  • GzipCodec
		• Compression type supported for use with RCFile:
		  • Block compression
		• Compression types supported for use with ORCFile:
		  • Block compression
		  • SnappyCodec

Hortonworks HDP 1.3.2 and 2.1 on commodity hardware	• TextFile
	• RCFile
	• ORCFile
		• Compression types supported for use with TextFile:
		  • DefaultCodec
		  • BZip2Codec
		  • GzipCodec
		• Compression type supported for use with RCFile:
		  • Block compression
		• Compression types supported for use with ORCFile:
		  • Block compression
		  • SnappyCodec

Cloudera 4.3 and 5.1 on commodity hardware	• TextFile
	• RCFile
		• Compression types supported for use with TextFile:
		  • DefaultCodec
		  • BZip2Codec
		  • GzipCodec
		• Compression type supported for use with RCFile:
		  • Block compression

MapR 3.1.1 on commodity hardware	• TextFile
	• RCFile
	• ORCFile
		• Compression types supported for use with TextFile:
		  • DefaultCodec
		  • BZip2Codec
		  • GzipCodec
		• Compression type supported for use with RCFile:
		  • Block compression
		• Compression types supported for use with ORCFile:
		  • Block compression
		  • SnappyCodec

Usage Notes

LOAD_FROM_HCATALOG can be specified only in the FROM clause of a SELECT statement. The table that is returned is constrained by the same limits that apply to all Teradata Database tables.

Queries that use LOAD_FROM_HCATALOG can be CPU intensive. Teradata recommends that you use workload management rules to minimize the impact of CPU usage when users submit concurrent queries that use LOAD_FROM_HCATALOG. A user can manually create a rule that is based on the table operator name, SYSLIB.LOAD_FROM_HCATALOG. This value is then used for throttling. Note that if Teradata Dynamic Workload Management does not detect an existing rule for SYSLIB.LOAD_FROM_HCATALOG, then at activate time, it dynamically creates an internal rule with a throttle value of 2. Once an automatic rule is created, the throttle value can only be changed if a user manually creates a rule for SYSLIB.LOAD_FROM_HCATALOG. The recommended throttle value is 2 or less. Query latency increases significantly when running more than two concurrent queries.

For information about using workload management, see *Teradata Viewpoint User Guide*.

To produce a listing of all databases available in the Hadoop HCatalog, omit both the dbname and tablename name value pairs.

To produce a listing of all tables in a particular database, pass the dbname name value pair, but omit the tablename name value pair.

To produce a listing of the columns available in a particular table, pass the dbname and tablename name value pairs, but omit the columns name value pair.

LOAD_FROM_HCATALOG is compatible with the following software:

- On Teradata Distribution for Hadoop: Hortonworks HDP versions 1.1.0.17, 1.3.2, and 2.1
- On commodity hardware: Hortonworks HDP versions 1.3.2 and 2.1, Cloudera CDH 4.3 and 5.1, and MapR 3.1.1

Note: If you use Teradata SQL-H with MapR 3.1.1, you must run Teradata Database on SLES 11 SP1. Teradata SQL-H does not support the use of SLES 10 when used with MapR 3.1.1.

To use Teradata SQL-H, you need the following configuration:

- A network that connects all Teradata nodes to all Hadoop data nodes.
- The Teradata tdatuser configured as a proxy user on the Hadoop namenode.

Note: Release 14.10 of Teradata SQL-H does not support authentication through LDAP or Kerberos on Hadoop systems.

Using hadoop_properties

For Hadoop distributions, High Availability hadoop properties are generally defined based on the name service that is defined on your hadoop server.

Your particular Hadoop setup may require other properties, but the following properties can be used as a general guideline for properties to set when using Teradata SQL-H with Hortonworks 2.1:

```
hadoop_properties('
    <dfs.client.use.datanode.hostname=true>
    ,<dfs.datanode.usedatanode.hostname=true>
    ,<dfs.nameservices=mycluster>
    ,<dfs.ha.namenodes.mycluster=nn1,nn2>
    ,<dfs.namenode.rpc-address.mycluster.nn1=hdp230-2:8020>
    ,<dfs.namenode.rpc-address.mycluster.nn2=hdp230-3:8020>
,<dfs.client.failover.proxy.provider.mycluster=org.apache.
hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider>')
```

Note that you would replace *mycluster* in the hadoop_properties text with your own name service ID.

In most cases, you should set the dfs.client.use.datanode.hostname property to true. If you have a setup where your TPA nodes are on one BYNET, the Hadoop cluster is on another BYNET, and they are communicating with one another via Ethernet, then you should also set the dfs.datanode.usedatanode.hostname property to true.

If you are using Teradata SQL-H with MapR, and you want to set hadoop_properties in the query itself, you should use the 3 properties described in the following table:

Name	Description
dfs.ha.namenodes.TeraMapR	The prefix for a given nameservice, contains a comma-separated list of namenodes for the given nameservice. For Teradata SQL-H, the nameservice is TeraMapR.
dfs.nameservices	The name of a nameservice, or a comma-separated list of nameservices. For Teradata SQL-H, the nameservice is TeraMapR.
dfs.namenode.rpc-address.TeraMapR.cldb*x*	The RPC address that handles all clients requests. In the case of High Availability and Federation, where multiple namenodes exist, the name service id is added to the name, for example, dfs.namenode.rpc-address.ns1 dfs.namenode.rpc-address.EXAMPLENAMESERVICE. The value of this property takes the form of nn-host1:rpc-port.You should replace the *x* with the number of each cluster database.

For example, you would use something similar to the following:

```
hadoop_properties'<dfs.ha.namenodes.TeraMapR=cldb1,cldb2>,
<dfs.nameservices=TeraMapR>,
<dfs.namenode.rpc-address.TeraMapR.cldb1=server1.mycompany.com:7222>,
<dfs.namenode.rpc-address.TeraMapR.cldb2=server1.mycompany.com:7222>')
```

If you do not set the MapR hadoop_properties in the query itself, the query runs using WebHCat.

If you are using Teradata SQL-H with Cloudera, the following properties are not required, but are frequently needed:

- dfs.client.use.datanode.hostname=true
- dfs.datanode.usedatanode.hostname=true

Examples

These examples use the following tables:

- carpricedata is a Hortonworks table that has fourteen columns of various types. Some columns are partitioned.
- arraydata is a Hortonworks table that contains Hadoop columns of complex types.
- customer, a Teradata table that contains some simple customer data columns like last_name, users_make, and users_model.

Example: Using a Simple SELECT

This example shows a simple select from a Hadoop database.

```
SELECT
        CAST(Price AS DECIMAL (8,2))
        , mileage
        , CAST(make AS VARCHAR(20))
        , CAST(model AS VARCHAR(20))
FROM load_from_hcatalog(
        USING
                server('hcatalog_server')
                port('9083')
                username('smith')
                dbname('default')
                tablename('carpricedata')
                columns('*')
                templeton_port('1880')
) as dt;


price     mileage  make    model
--------  -------  ------  --------------------
17314.10  8221     Buick   Century
17542.04  9135     Buick   Century
```

Example: Using Column and Partitionfilter *name value* Pairs

This example shows the use of the columns and partitionfilter name value pairs.

Note: Only the partitionfilter name value pair limits the amount of data that is imported from Hadoop over the network to Teradata Database. The columns name value pair and WHERE clause act only on the data that has been imported from Hadoop.

```
SELECT * FROM load_from_hcatalog( ON empty USING
         server('hcatalog_server')
         port('9083')
         username('smith')
         dbname('default')
         tablename('carpricedata')
         columns('doors,cylinder,liter')
         partitionfilter('doors="4"')
         templeton_port('1880')
) as D1 WHERE liter<4;


  doors     cylinder                  liter
-----------  -----------  ----------------------
```

```
          4            6   3.09999990463257E 000
          4            6   3.09999990463257E 000
```

Example: Using a Join

This example demonstrates joining data from a Hadoop table to a Teradata Database table.

```
SELECT customer.l:ast_name, CAST(lfm.price AS DECIMAL (8,2)) AS 'Price',
lfm.mileage
FROM load_from_hcatalog ( ON empty USING
        server('hcatalog_server')
        port('9083')
        username('smith')
        dbname('default')
        tablename('carpricedata')
        columns('*')
        templeton_port('1880')
      ) as lfm, customer
 WHERE lfm.make = customer.users_make
   AND lfm.model = customer.users_model;


last_name     price       mileage
------------  ----------  ---------
Jones         15709.05    22236
Jones         16339.17    19832
```

Example: Using an INSERT...SELECT

This example demonstrates using an INSERT...SELECT request to create a Teradata Database table from imported Hadoop data.

```
INSERT INTO cpd
SELECT * FROM load_from_hcatalog(
      USING
         server('hcatalog_server')
         port('9083')
         username('smith')
         dbname('default')
         tablename('carpricedata')
         columns('doors,cylinder,liter')
         templeton_port('1880')
) as dt WHERE liter<4;
```

For More Information

For more information about using LOAD_FROM_HCATALOG and additional examples, see the Orange Book *SQL-H for Teradata*, publication number 541-0009812.

TD_UNPIVOT

Purpose

Transforms table columns into rows.

Syntax

where:

Syntax element ...	Specifies ...
TD_SYSFNLIB	the name of the database where the function is located.
tableName	the name of a table with the input data for the function.
query_expression	the SELECT statement with input data for the function.
VALUE _COLUMNS	the destination column or columns for the unpivotted data.
UNPIVOT_COLUMN	the name of the output column identifying the unpivotted data in each row of the value column or columns.
COLUMN_LIST	the input column data for the unpivotting operation.
COLUMN_ALIAS_LIST	alternate names for the values in the unpivotted column.

Syntax element ...	Specifies ...
INCLUDE_NULLS	whether or not to include nulls in the transformation Valid values are: • 'Yes" • 'No' (default)

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Data Type

The data types of all source columns that are mapped into the same value column must be compatible. For the purposes of TD_UNPIVOT, two columns are compatible if they are both numeric, both CHAR or VARCHAR, both BYTE or VARBYTE, or both identical.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

Example 1

In the following example, 12 columns (shown in COLUMN_LIST) are unpivotted to 1 column with 12 rows.

The input table T in the *query_expression* (in the SELECT statement below) has the following columns: id, year, jan_sales, feb_sales, ..., dec_sales

```
SELECT * from TD_UNPIVOT(
        ON( select * from T)
        USING
            VALUE_COLUMNS('monthly_sales')
            UNPIVOT_COLUMN('month')
            COLUMN_LIST('jan_sales', 'feb_sales', ..., 'dec_sales')
            COLUMN_ALIAS_LIST('jan', 'feb', ..., 'dec' )
    )X;
```

The output columns are the following: id, year, month, monthly_sales.

One row of input, for example:

```
 id      year    jan_sales   feb_sales   ... dec_sales
 123     2012    100         200         ... 1200
```

generates 12 rows of output:

```
id  year    month   monthly_sales
123 2012    jan     100
123 2012    feb     200
...
123 2012    dec     1200
```

Example 2

In the following example, there is more than one value column and the COLUMN_LIST takes the form: 'a, x, ..', 'b, y, ..', 'c, z, ...', ... where a, x, b, y, c, z, ... are the names of the columns to be unpivotted into the value columns.

Each individual list, for example, 'a, x, ..', defines which columns are mapped to the value columns for one of the rows of output.

The input table T has columns id, year, jan_sales, jan_expense, feb_sales, feb_expense, ..., dec_sales, dec_expense.

```
SELECT * from UNPIVOT(
       ON( select * from T)
       USING
           VALUE_COLUMNS('monthly_sales', 'monthly_expense')
           UNPIVOT_COLUMN('month')
           COLUMN_LIST('jan_sales, jan_expense', 'feb_sales,
              feb_expense', ..., 'dec_sales, dec_expense')
           COLUMN_ALIAS_LIST('jan', 'feb', ..., 'dec' )
   )X;
```

The output columns are the following: id, year, month, monthly_sales, monthly_expense.

One row of input, for example:

```
 id   year  jan_sales  jan_exp feb_sales feb_exp...dec_sales dec_exp
123   2012  100        50      200       100  ...  1200      600
```

generates 12 rows of output:

```
id  year    month   monthly_sales   monthly_expense
123 2012    jan     100             50
123 2012    feb     200             100
...
123 2012    dec     1200            600
```

Usage Notes for Examples

- Columns in table T that are not included in any of the column lists are referred to as "copy columns," and are included in every row of the output.
- Value column names and the UNPIVOT column name must not exist in table T.
- The names in *column_list_value* must exist in table T.
- The number of VALUE_COLUMNS must be equal to the number of columns in each column list. One column list defines which source columns get mapped to the value columns for one row of output.
- Column alias list values are names that are used for the UNPIVOT_COLUMN to indicate which source column(s) data is in the value columns for that row.

  If specified, the number of alias names must be equal to the number of column lists.

  If not specified, column alias names default to the names of the columns in the column list, concatenated with an '_'.
- The UNPIVOT_COLUMN is a VARCHAR(1000).

- If INCLUDE_NULLS = 'yes', there will be exactly *n* rows of output for every row of input, where *n* is the number of column lists.

  If INCLUDE_NULLS = 'no', then there will be at most *n* rows of output for every row of input, where *n* is the number of column lists. Any row with NULLS for every value column is eliminated.

CHAPTER 27 User-Defined Functions

Teradata Database supports two types of user-defined functions (UDFs) that allow you to extend SQL by writing your own functions:

- SQL UDFs

  SQL UDFs allow you to encapsulate regular SQL expressions in functions and then use them like standard SQL functions.
- External UDFs

  External UDFs allow you to write your own functions in the C, C++ or the Java programming language, install them on the database, and then use them like standard SQL functions. For details on external UDFs, see *SQL External Routine Programming*.

UDFs can be of the following types:

- Scalar

  A scalar UDF can appear almost anywhere a standard SQL scalar function can appear.
- Aggregate or Window Aggregate

  A window aggregate UDF is an aggregate UDF with a window specification applied to it. An aggregate UDF can appear almost anywhere a standard SQL aggregate function can appear.
- Table

  A table UDF can only appear in the FROM clause of an SQL SELECT statement.

SQL UDF

Purpose

As user-defined function written using regular SQL expressions, SQL UDF is used like a standard SQL function.

Syntax

where:

Syntax element ...	Specifies ...
udf_name	the name of the SQL UDF.
argument	a valid SQL expression. See Usage Notes for rules that apply to SQL UDF arguments.

ANSI Compliance

SQL UDFs are partially ANSI SQL:2011 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Restrictions

Self-referencing, forward-referencing, and circular-referencing SQL UDFs are not allowed.

Authorization

You must have EXECUTE FUNCTION privilege on the function or on the database containing the function.

You can specify an SQL SECURITY clause with the DEFINER option in the CREATE/REPLACE FUNCTION statement. This option is the default for an SQL UDF. SQL SECURITY DEFINER means that when an SQL UDF is invoked, Teradata Database verifies that the immediate owner and the creator of the UDF have the appropriate privileges on the underlying database objects referenced in the UDF. If the creator does not exist when the privileges are checked, an error is returned.

To invoke a UDF that takes a UDT argument or returns a UDT, you must have the UDTUSAGE privilege on the SYSUDTLIB database or on the specified UDT.

Usage Notes

An SQL UDF is a function that is defined by a user and is written using SQL expressions. When Teradata Database evaluates an SQL UDF expression, it invokes the function with the arguments passed to it. The following rules apply to the arguments in the function call:

- The arguments must be comma-separated expressions in the same order as the parameters declared in the function.
- The number of arguments passed to the SQL UDF must be the same as the number of parameters declared in the function.
- The data types of the arguments must be compatible with the corresponding parameter declarations in the function and follow the precedence rules that apply to compatible types. For details, see *SQL Data Definition Language*.

  To pass an argument that is not compatible with the corresponding parameter type, use CAST to explicitly convert the argument to the proper type. For information, see "CAST in Explicit Data Type Conversions" on page 586.
- A NULL argument is compatible with a parameter of any data type. You can pass a NULL argument explicitly or by omitting the argument.
- Any form of SQL expression can be used as an argument with three important rules:
  - The SQL expression must not be a Boolean value expression (that is, a conditional expression).
  - If the expression is a nondeterministic SQL expression (expressions involving random functions and/or nondeterministic UDFs), it must not correspond to a parameter that is used more than once in the RETURN statement.
  - The SQL expression must not be a scalar subquery.

When an SQL UDF is invoked, Teradata Database searches for the UDF in the following locations:

- In the database specified if the function call is qualified by a database name.
- In the current database.
- In the SYSLIB database.

For details regarding UDF search resolution, see *SQL Data Definition Language*.

The result type of an SQL UDF is based on the return type specified in the RETURNS clause of the CREATE FUNCTION statement.

The default title of an SQL UDF appears as:

```
UDF_name(argument_list)
```

Example 1

Consider the following function definition and query:

```
CREATE FUNCTION Test.MyUDF (a INT, b INT, c INT)
```

```
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a + b - c;


SELECT Test.MyUDF(t1.a1, t2.a2, t3.a3) FROM t1, t2, t3;
```

The user executing the SELECT statement must have the following privileges:

- SELECT privilege on tables t1, t2, and t3, their containing databases, or on the columns t1.a1, t2.a2, and t3.a3.
- EXECUTE FUNCTION privilege on MyUDF or on the database named Test.

The privileges of the creator or owner are not checked since the UDF does not reference any database objects in its definition.

Example 2

In this example, the SQL UDF named MySQLUDF references an external UDF named MyExtUDF in the RETURN statement.

Consider the following function definition and query:

```
CREATE FUNCTION Test.MySQLUDF (a INT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN a + b * MyExtUDF(a, b) - c;


SELECT Test.MySQLUDF(t1.a1, t2.a2, t3.a3) FROM t1, t2, t3;
```

The user executing the SELECT statement must have the following privileges:

- SELECT privilege on tables t1, t2, and t3, their containing databases, or on the columns t1.a1, t2.a2, and t3.a3.
- EXECUTE FUNCTION privilege on MySQLUDF or on the database named Test.

Because the SQL UDF references MyExtUDF, the following privileges are also checked:

- The creator of MySQLUDF must exist and have the EXECUTE FUNCTION privilege on MyExtUDF or its containing database.
- The database named Test (the immediate owner of MySQLUDF) must have the EXECUTE FUNCTION privilege on MyExtUDF or its containing database.

Example 3

In this example, invocations of the SQL UDF named MyUDF2 are passed as arguments to the SQL UDF named MyUDF1.

```
CREATE FUNCTION test.MyUDF1 (a INT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
COLLATION INVOKER
INLINE TYPE 1
RETURN a * b * c;


CREATE FUNCTION test.MyUDF2 (d INT, e INT, f INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
COLLATION INVOKER
INLINE TYPE 1
RETURN d + e + f;


SELECT test.MyUDF1(test.MyUDF2(t1.a1, 1, 2),
       test.MyUDF2(t1.b1, 2, 3), 5) FROM t1;
```

Example 4

In this example, the UDF invocation argument data type (BYTEINT) is not the same as that of the corresponding UDF parameter data type (INTEGER) since the size of the argument data type is less than the UDF parameter data type. However, because the two data types are compatible and a BYTEINT argument can fit inside an INTEGER parameter, this is allowed.

```
CREATE FUNCTION test.MyUDF (a INT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
COLLATION INVOKER
INLINE TYPE 1
RETURN a * b * c;

CREATE TABLE t1 (a1 BYTEINT, b1 INT);

SELECT test.MyUDF(t1.a1, t1.b1, 2) FROM t1;
```

Example 5

In this example, the UDF invocation argument data type (INTEGER) is not the same as that of the corresponding UDF parameter data type (BYTEINT) since the size of the argument data type is greater than the UDF parameter data type. Although the two data types are compatible, an INTEGER argument cannot fit inside a BYTEINT parameter, so an error is returned.

```
CREATE FUNCTION test.MyUDF (a BYTEINT, b INT, c INT)
RETURNS INT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
COLLATION INVOKER
INLINE TYPE 1
RETURN a * b * c;

CREATE TABLE t1 (a1 INT, b1 INT);

SELECT test.MyUDF(t1.a1, t1.b1, 2) FROM t1;
```

The following error is returned:

```
Failure 5589: Function "test.MyUDF" does not exist.
```

To avoid the error, the caller must explicitly cast the value of t1.a1 to BYTEINT as follows:

```
SELECT test.MyUDF(CAST(t1.a1 AS BYTEINT), t1.b1, 2) FROM t1;
```

Related Topics

FOR more information on ...	SEE ...
• CREATE FUNCTION • REPLACE FUNCTION	• *SQL Data Definition Language*. • *Database Administration*.
EXECUTE FUNCTION and UDTUSAGE privileges	*SQL Data Control Language*.

Scalar UDF

Purpose

Takes input arguments and returns a single value result.

Syntax

For UDFs that are defined with parameters of TD_ANYTYPE data type:

where:

Syntax element ...	Specifies ...
udf_name	the name of the scalar UDF.
argument	a valid SQL expression. See Usage Notes for rules that apply to scalar UDF arguments.
data_type	the desired return type of the TD_ANYTYPE result parameter.
column_expr	a table or a view column reference that is used to determine the return type of the TD_ANYTYPE result parameter.

ANSI Compliance

Scalar UDFs are partially ANSI SQL:2011 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

The RETURNS *data_type* or RETURNS STYLE clauses are Teradata extensions to the ANSI SQL standard.

Restrictions

- Any restrictions that apply to standard SQL scalar functions also apply to scalar UDFs.
- Scalar UDF expressions cannot be used in a partitioning expression of the CREATE TABLE statement.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

To invoke a scalar UDF that takes a UDT argument or returns a UDT, you must have the UDTUSAGE privilege on the SYSUDTLIB database or on the specified UDT.

UDF Arguments

When Teradata Database evaluates a UDF expression, it invokes the function with the arguments passed to it. The following rules apply to the arguments in the function call:

- The arguments must be comma-separated expressions in the same order as the parameters declared in the function.
- The number of arguments passed to the UDF must be the same as the number of parameters declared in the function.
- The data types of the arguments must be compatible with the corresponding parameter declarations in the function and follow the precedence rules that apply to compatible types. For details, see *SQL External Routine Programming*.

  To pass an argument that is not compatible with the corresponding parameter type, use CAST to explicitly convert the argument to the proper type. For information, see "CAST in Explicit Data Type Conversions" on page 586.

Result Data Type

The result type of a scalar UDF is based on the return type specified in the RETURNS clause of the CREATE FUNCTION statement.

When invoking a scalar or aggregate UDF that is defined with a TD_ANYTYPE result parameter, you can use the RETURNS *data type* or RETURNS STYLE *column expression* clauses in the function call to specify the desired return type. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

The RETURNS or RETURNS STYLE clause is not mandatory as long as the function call also includes a TD_ANYTYPE input argument. If you do not specify a RETURNS or RETURNS STYLE clause, then the data type of the first TD_ANYTYPE input argument is used to determine the return type of the TD_ANYTYPE result parameter. For character types, if the character set is not specified as part of the data type, then the default character set is used.

You can use these clauses only with scalar and aggregate UDFs. You cannot use them with table functions. Also, you must enclose the UDF invocation in parenthesis if you use the RETURNS or RETURNS STYLE clauses.

Default Title

The default title of a scalar UDF appears as:

```
UDF_name(argument_list)
```

Example 1

Consider the following table definition and data:

```
CREATE TABLE pRecords (pname CHAR(30),
                       pkey INTEGER);

SELECT * FROM pRecords;
```

The output from the SELECT statement is:

```
pname                                  pkey
------------------------------  -----------
Tom                                       6
Bob                                       5
Jane                                      4
```

The following is the SQL definition of a scalar UDF that calculates the factorial of an integer argument:

```
CREATE FUNCTION factorial (i INTEGER)
RETURNS INTEGER
SPECIFIC factorial
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
NOT DETERMINISTIC
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'ss!factorial!factorial.c!F!fact'
```

The following query uses the scalar UDF expression to calculate the factorial of the pkey column + 1.

```
SELECT pname, factorial(pkey)+1
FROM pRecords;
```

The output from the SELECT statement is:

```
pname                           (factorial(pkey)+1)
------------------------------  -------------------
Tom                                             721
Bob                                             121
Jane                                             25
```

Example 2

Consider the following table and function definitions:

```
CREATE TABLE T1 (intCol INTEGER,
                 varCharCol VARCHAR(40) CHARACTER SET UNICODE);

CREATE TABLE T2 (intCol INTEGER,
                 decimalCol DECIMAL (10, 6));

CREATE FUNCTION myUDF1 (A INTEGER, B TD_ANYTYPE)
```

```
RETURNS TD_ANYTYPE;

CREATE FUNCTION myUDF2 (A TD_ANYTYPE, B TD_ANYTYPE)
RETURNS TD_ANYTYPE;
```

The following invocation of myUDF1 uses the RETURNS *data_type* clause to specify the UDF return type to be DECIMAL(10,6).

```
SELECT (myUDF1 (T1.intCol, T2.decimalCol) RETURNS DECIMAL(10,6));
```

The following invocation of myUDF2 uses the RETURNS STYLE clause to specify the UDF return type to be the data type of the T1.varCharCol column, which is VARCHAR(40) CHARACTER SET UNICODE.

```
SELECT (myUDF2 (T1.varCharCol, T2.decimalCol)
   RETURNS STYLE T1.varCharCol);
```

Related Topics

FOR more information on ...	SEE ...
Implementing external UDFs	*SQL External Routine Programming*.
• CREATE FUNCTION • REPLACE FUNCTION	• *SQL Data Definition Language*. • *Database Administration*.
EXECUTE FUNCTION and UDTUSAGE privileges	*SQL Data Control Language*.
the TD_ANYTYPE data type	*SQL Data Types and Literals*.

Aggregate UDF

Purpose

Takes grouped sets of relational data, makes a pass over each group, and returns one result for the group.

Syntax

For UDFs that are defined with parameters of TD_ANYTYPE data type:

where:

Syntax element ...	Specifies ...
udf_name	the name of the aggregate UDF.
argument	a valid SQL expression. See Usage Notes for rules that apply to aggregate UDF arguments.
data_type	the desired return type of the TD_ANYTYPE result parameter.
column_expr	a table or a view column reference that is used to determine the return type of the TD_ANYTYPE result parameter.

ANSI Compliance

Aggregate UDFs are partially ANSI SQL:2011 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

The RETURNS *data_type* or RETURNS STYLE clauses are Teradata extensions to the ANSI SQL standard.

Restrictions

- Any restrictions that apply to standard SQL aggregate functions also apply to aggregate UDFs.
- Aggregate UDF expressions cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify an aggregate UDF.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

To invoke an aggregate UDF that takes a UDT argument or returns a UDT, you must have the UDTUSAGE privilege on the SYSUDTLIB database or on the specified UDT.

Usage Notes

When Teradata Database evaluates an aggregate UDF expression, it invokes the aggregate function once for each item in an aggregation group, passing the detail values of a group through the input arguments. To accumulate summary information, the context is retained each time the aggregate function is called.

The rules that apply to the arguments in an aggregate function call are the same as those that apply to a scalar function call. See "UDF Arguments" on page 1296.

The result type of an aggregate UDF is based on the return type specified in the RETURNS clause of the CREATE FUNCTION statement. If the result parameter is of TD_ANYTYPE data type, see "Result Data Type" on page 1296 for information on how the return type is determined.

The default title of an aggregate UDF appears as:

```
UDF_name(argument_list)
```

Example

Consider the following table definition and data:

```
CREATE TABLE Product_Life
(Product_ID INTEGER,
 Product_class VARCHAR(30),
 Hours INTEGER);

SELECT * FROM Product_Life;
```

The output from the SELECT statement is:

```
Product_ID  Product_class                        Hours
-----------  ------------------------------  -----------
        100  Bulbs                                   100
        100  Bulbs                                   200
        100  Bulbs                                   300
```

The following is the SQL definition of an aggregate UDF that calculates the standard deviation of the input arguments:

```
CREATE FUNCTION STD_DEV (i INTEGER)
RETURNS FLOAT
CLASS AGGREGATE (64)
SPECIFIC std_dev
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'ss!stddev!stddev.c!f!STD_DEV'
```

The following query uses the aggregate UDF expression to calculate the standard deviation for the life of a light bulb.

```
SELECT Product_ID, SUM(Hours), STD_DEV(Hours)
FROM Product_Life
WHERE Product_class = 'Bulbs'
GROUP BY Product_ID;
```

The output from the SELECT statement is:

```
Product_ID   Sum(hours)          std_dev(hours)
-----------  -----------  ----------------------
        100          600   8.16496580927726E 001
```

Related Topics

FOR more information on ...	SEE ...
SQL aggregate functions	"Chapter 2 Aggregate Functions" on page 31.
window aggregate UDFs	"Window Aggregate UDF" on page 1302.
implementing aggregate UDFs	*SQL External Routine Programming*.
• CREATE FUNCTION • REPLACE FUNCTION	• *SQL Data Definition Language*. • *Database Administration*.
EXECUTE FUNCTION and UDTUSAGE privileges	*SQL Data Control Language*.
the TD_ANYTYPE data type	*SQL Data Types and Literals*.

Window Aggregate UDF

Purpose

Allows an aggregate UDF with a window specification applied to it to operate on a specified window of rows.

Syntax

where:

Syntax element ...	Specifies ...
udf_name	the name of the aggregate UDF on which the window specification is applied.
argument	a valid SQL expression. For rules that apply to aggregate UDF arguments, see "Aggregate UDF" on page 1299.

Syntax element ...	Specifies ...
OVER	how values are grouped, ordered, and considered when computing the cumulative, group, or moving function. Values are grouped according to the PARTITION BY and RESET WHEN clauses, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.
PARTITION BY	in its *column_reference*, or comma-separated list of column references, the group, or groups, over which the function operates. PARTITION BY is optional. If there are no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a single group, or partition. PARTITION BY clause is also called the window partition clause.
ORDER BY	in its *value_expression* the order in which the values in a group, or partition, are sorted.
ASC	ascending sort order. The default is ASC.
DESC	descending sort order.
RESET WHEN	the group or partition, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set, delivered by the FROM clause, constitutes a single partition. If RESET WHEN is specified, then the ORDER BY clause must be specified also.
condition	a conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition. The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions. For more information, see "RESET WHEN Condition Rules" on page 956 and the "QUALIFY Clause" in *SQL Data Manipulation Language*.
ROWS	the starting point for the aggregation group within the partition. The aggregation group end is the current row. The aggregation group of a row R is a set of rows, defined relative to R in the ordering of the rows within the partition. If there is no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

Syntax element ...	Specifies ...
ROWS BETWEEN	the aggregation group start and end, which defines a set of rows relative to the current row in the ordering of the rows within the partition. The row specified by the group start must precede the row specified by the group end. If there is no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.
UNBOUNDED PRECEDING	the entire partition preceding the current row.
UNBOUNDED FOLLOWING	the entire partition following the current row.
CURRENT ROW	the start or end of the aggregation group as the current row.
value PRECEDING	the number of rows preceding the current row. The value for *value* is always a positive integer literal. The maximum number of rows in an aggregation group is 4096 when *value* PRECEDING appears as the group start or group end.
value FOLLOWING	the number of rows following the current row. The value for *value* is always a positive integer literal. The maximum number of rows in an aggregation group is 4096 when *value* FOLLOWING appears as the group start or group end.

ANSI Compliance

Window aggregate UDFs are partially ANSI SQL:2011 compliant.

The requirement that parentheses appear when the argument list of an aggregate UDF is empty is a Teradata extension to preserve compatibility with existing applications.

In the presence of an ORDER BY clause and the absence of a ROWS or ROWS BETWEEN clause, ANSI SQL:2011 window aggregate functions use ROWS UNBOUNDED PRECEDING as the default aggregation group, whereas Teradata SQL window aggregate functions use ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

To invoke an aggregate UDF that takes a UDT argument or returns a UDT, you must have the UDTUSAGE privilege on the SYSUDTLIB database or on the specified UDT.

Arguments to Window Aggregate UDFs

Window aggregate UDFs can take literals, literal expressions, column names (sales, for example), or column expressions (sales + profit) as arguments.

Window aggregates can also take regular aggregates as input parameters to the PARTITION BY and ORDER BY clauses. The RESET WHEN clause can take an aggregate as part of the RESET WHEN condition clause.

The rules that apply to the arguments of the window aggregate UDF are the same as those that apply to aggregate UDF arguments, see "Aggregate UDF" on page 1299.

Supported Window Types for Aggregate UDFs

Window Type	Aggregation Group	Supported Partitioning Strategy
Reporting window	ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	Hash partitioning
Cumulative window	• ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW • ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING	Hash partitioning
Moving window	• ROWS BETWEEN *value* PRECEDING AND CURRENT ROW • ROWS BETWEEN CURRENT ROW AND *value* FOLLOWING • ROWS BETWEEN *value* PRECEDING AND *value* FOLLOWING • ROWS BETWEEN *value* PRECEDING AND *value* PRECEDING • ROWS BETWEEN *value* FOLLOWING AND *value* FOLLOWING	Hash partitioning and range partitioning

Consider the following table definition:

```
CREATE TABLE t (id INTEGER, v INTEGER);
```

The following query specifies a reporting window of rows which the window aggregate UDF MYSUM operates on:

```
SELECT id, v, MYSUM(v) OVER (PARTITION BY id ORDER BY v)
FROM t;
```

The following query specifies a cumulative window of rows which the window aggregate UDF MYSUM operates on:

```
SELECT id, v, MYSUM(v) OVER (PARTITION BY id ORDER BY v
                             ROWS UNBOUNDED PRECEDING)
FROM t;
```

The following query specifies a moving window of rows which the window aggregate UDF MYSUM operates on:

```
SELECT id, v, MYSUM(v) OVER (PARTITION BY id ORDER BY v
                             ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING)
FROM t;
```

Unsupported Window Types for Aggregate UDFs

Window Type	Aggregation Group
Moving window	• ROWS BETWEEN UNBOUNDED PRECEDING AND *value* FOLLOWING • ROWS BETWEEN *value* PRECEDING AND UNBOUNDED FOLLOWING

Partitioning

The range partitioning strategy helps to avoid hot AMP situations where the values of the columns of the PARTITION BY clause result in the distribution of too many rows to the same partition or AMP.

Range and hash partitioning is supported for moving window types. Only hash partitioning is supported for the reporting and cumulative window types because of potential ambiguities that can occur when a user tries to reference previous values assuming a specific ordering within window types like reporting and cumulative, which are semantically not order dependant.

You should use an appropriate set of column values for the PARTITION BY clause to avoid potential skew situations for the reporting or cumulative aggregate cases. For more information, see "Data in Partitioning Column of Window Specification and Resource Impact" on page 950.

Result Type and Format

The result data type of a window aggregate UDF is based on the return type of the aggregate UDF, which is specified in the RETURNS clause of the CREATE FUNCTION statement.

The default format of a window aggregate UDF is the default format for the return type. For information on the default format of data types and an explanation of the formatting characters in the format, see "Data Type Formats and Format Phrases" in *SQL Data Types and Literals*.

Usage Notes

You can apply a window specification to an aggregate UDF. The window feature provides a way to dynamically define a subset of data, or window, and allows the aggregate function to operate on that window of rows. Without a window specification, aggregate functions return one value for all qualified rows examined, but window aggregate functions return a new value for each of the qualifying rows participating in the query.

Problems With Missing Data

Ensure that data you analyze has no missing data points. Computing a moving function over data with missing points produces unexpected and incorrect results because the computation considers *n* physical rows of data rather than *n* logical data points.

Restrictions

- The window feature is *not* supported for aggregate UDFs written in Java.
- Range partitioning for the reporting or cumulative window types is not supported.
- Any restrictions that apply to aggregate UDFs also apply to window aggregate UDFs.
- Any restrictions that apply to the window specification of a standard SQL aggregate function also apply to the window specification of an aggregate UDF.

Example

Consider the following table definition and inserted data:

```
CREATE MULTISET TABLE t
   (id INTEGER,
    v  INTEGER);

INSERT INTO t VALUES (1,1);
INSERT INTO t VALUES (1,2);
INSERT INTO t VALUES (1,2);
INSERT INTO t VALUES (1,4);
INSERT INTO t VALUES (1,5);
INSERT INTO t VALUES (1,5);
INSERT INTO t VALUES (1,5);
INSERT INTO t VALUES (1,8);
INSERT INTO t VALUES (1,);
```

The following is the SQL definition of a window aggregate UDF that performs the dense rank operation:

```
REPLACE FUNCTION dense_rank (x INTEGER)
RETURNS INTEGER
CLASS AGGREGATE (1000)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL;
```

The dense_rank UDF evaluates dense rank over the set of values passed as arguments to the UDF. With dense ranking, items that compare equal receive the same ranking number, and the next item(s) receive the immediately following ranking number. In the following query and result, note the difference in the rank and dense rank value for v=4. The dense rank value is 4 whereas the rank of 4 is 5.

```
SELECT v, dense_rank(v) OVER (PARTITION BY id ORDER BY v
   ROWS UNBOUNDED PRECEDING) as dr,
   rank() OVER (PARTITION BY id ORDER BY v) as r
FROM t ORDER BY dr;
```

The output from the SELECT statement is:

```
           v            dr            r
-----------  -----------  -----------
           ?            1            1
           1            2            2
           2            3            3
           2            3            3
           4            4            5
           5            5            6
           5            5            6
           5            5            6
           8            6            9
```

For a C code example of the dense_rank UDF, see "C Window Aggregate Function" in *SQL External Routine Programming*.

Related Topics

FOR more information on ...	SEE ...
aggregate UDFs	"Aggregate UDF" on page 1299.
ordered analytical functions and the window feature	"The Window Feature" on page 953.
implementing window aggregate UDFs	*SQL External Routine Programming*.
• CREATE FUNCTION • REPLACE FUNCTION	• *SQL Data Definition Language*. • *Database Administration*.
EXECUTE FUNCTION and UDTUSAGE privileges	*SQL Data Control Language*.

Table UDF

As user-defined function invoked in the FROM clause of a SELECT statement, a Table UDF returns a table to the statement.

See the TABLE option of the FROM clause in *SQL Data Manipulation Language*.

ANSI Compliance

Table UDFs are partially ANSI SQL:2011 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Restrictions

A table UDF can only appear in the FROM clause of an SQL SELECT statement. The SELECT statement containing the table function can appear as a subquery.

Authorization

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

To invoke a table UDF that takes a UDT argument or returns a UDT, you must have the UDTUSAGE privilege on the SYSUDTLIB database or on the specified UDT.

Usage Notes

When Teradata Database evaluates a table UDF expression, it invokes the table function which returns a table a row at a time in a loop to the SELECT statement. The function can produce the rows of a table from the input arguments passed to it or by reading an external file or message queue.

A table function can have 128 input parameters. The rules that apply to the arguments in a table function call are the same as those that apply to a scalar function call. See "UDF Arguments" on page 1296.

Table UDFs do not have return values. The columns in the result rows that they produce are returned as output parameters.

The output parameters of a table function are defined by the RETURNS TABLE clause of the CREATE FUNCTION statement. The number of output parameters is limited by the maximum number of columns that can be defined for a regular table.

The number and data types of the output parameters can be specified statically in the CREATE FUNCTION statement or dynamically at runtime in the SELECT statement that invokes the table function.

Table Functions and Table Operators cannot execute against fallback data when an AMP is Down. Once the AMP is returned to service the query can complete as normal.

Example

In this example, the extract_field table UDF is used to extract the *customer ID*, *store number*, and *item ID* from the pending_data column of the raw_cust table.

The raw_cust table is defined as:

```
CREATE SET TABLE raw_cust ,NO FALLBACK ,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL,
   CHECKSUM = DEFAULT
   (
    region INTEGER,
    pending_data VARCHAR(32000) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX (region);
```

The pending_data text field is a string of numbers with the format:

store number, *entries*:*customer ID*, *item ID*, ...; repeat;

where:

- *store number* is the store that sold these items to customers.
- *entries* is the number of items that were sold.
- *customer ID*, *item ID* represent the item each customer bought. *customer ID*, *item ID* is repeated *entries* times ending with a semi-colon ';'.
- The above sequence can be repeated.

The following shows sample data from the raw_cust table:

```
  region  pending_data
--------  ------------------------------------------------------------
       2  7,2:879,3788,879,4500;08,2:500,9056,390,9004;
       1  25,3:9005,3789,9004,4907,398,9004;36,2:738,9387,738,9550;
       1  25,2:9005,7896,9004,7839;36,1:737,9387;
```

The following shows the SQL definition of the extract_field table UDF:

```
CREATE FUNCTION extract_field (Text VARCHAR(32000),
                               From_Store INTEGER)
RETURNS TABLE (Customer_ID INTEGER,
               Store_ID INTEGER,
               Item_ID INTEGER)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME extract_field;
```

The following query extracts and displays the customers and the items they bought from store 25 in region 1.

```
SELECT DISTINCT cust.Customer_ID, cust.Item_ID
FROM raw_cust,
TABLE (extract_field(raw_cust.pending_data, 25))
```

```
AS cust
WHERE raw_cust.region = 1;
```

The output from the SELECT statement is similar to:

```
Customer_ID       Item_ID
------------  -----------
        9005         3789
        9004         4907
         398         9004
        9005         7896
        9004         7839
```

Related Topics

FOR more information on ...	SEE ...
Implementing external UDFs	*SQL External Routine Programming*.
• CREATE FUNCTION • REPLACE FUNCTION	• *SQL Data Definition Language*. • *Database Administration*.
EXECUTE FUNCTION and UDTUSAGE privileges	*SQL Data Control Language*.
the TABLE option in the FROM clause of an SQL SELECT statement	*SQL Data Manipulation Language*.

CHAPTER 28 User-Defined Type Expressions/ Methods

This chapter describes expressions related to user-defined types (UDTs).

UDT Expression

Purpose

Returns a distinct or structured UDT data type.

Syntax

where:

Syntax element ...	Specifies ...
database_name	an optional qualifier for the *column_name*.
table_name	an optional qualifier for the *column_name*.
column_name	the name of a distinct or structured UDT column.
udf_name	the name of a UDF that returns a distinct or structured UDT.
argument	an argument to the UDF.
CAST	a CAST expression that converts a source data type to a distinct or structured UDT. Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Syntax element ...	Specifies ...
expression	an expression that results in a data type that is compatible as the source type of a cast definition for the target UDT.
udt_name	the name of a distinct or structured UDT data type.
NEW	an expression that constructs a new instance of a structured type and initializes it using the specified constructor method. For details on NEW, see "NEW" on page 1318.
SYSUDTLIB.	the database in which the constructor exists. Teradata Database only searches the SYSUDTLIB database for UDT constructors, regardless of whether the database name appears in the expression.
constructor_name	the name of a constructor method associated with a UDT. Constructor methods have the same name as the UDT with which they are associated.
argument	an argument to pass to the constructor. Parentheses must appear even though the argument list may be empty.
method_name	the name of an instance method that returns a UDT. For details on method invocation, see "Method Invocation" on page 1324.
argument	an argument to pass to the method. Parentheses must appear even though the argument list may be empty.

ANSI Compliance

UDT expressions are partially ANSI SQL:2011 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Authorization

To use a UDT expression, you must have the UDTTYPE, UDTMETHOD, or UDTUSAGE on the SYSUDTLIB database or the UDTUSAGE privilege on all of the specified UDTs.

Usage Notes

You can use UDT expressions as input arguments to UDFs written in C or C++. You cannot use UDT expressions as input arguments to UDFs written in Java.

You can also use UDT expressions as IN and INOUT parameters of stored procedures and external stored procedures written in C or C++. However, you cannot use UDT expressions as IN and INOUT parameters of external stored procedures written in Java.

You can use UDT expressions with most SQL functions and operators, with the exception of ordered analytical functions, provided that a cast definition exists that casts the UDT to a

predefined type that is accepted by the function or operator. For details, see other chapters in this book.

Examples

Consider the following statements that create a distinct UDT named *euro* and a structured UDT named *address*:

```
CREATE TYPE euro
AS DECIMAL(8,2)
FINAL;

CREATE TYPE address
AS (street VARCHAR(20)
   ,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE european_sales
  (region INTEGER
  ,location address
  ,sales DECIMAL(8,2));
```

Example 1: Column Name

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE italian_sales
  (location address
  ,sales DECIMAL(8,2));
```

The *location* column reference in the following statement returns an *address* UDT expression.

```
INSERT INTO italian_sales
   SELECT location, sales
   FROM european_sales
   WHERE region = 1151;
```

Example 2: CAST

The following statement creates a table that defines a *euro* column named *sales*:

```
CREATE TABLE swiss_sales
  (location address
  ,sales euro);
```

The following statement uses CAST to return a *euro* UDT expression. Using CAST requires a cast definition that converts the DECIMAL(8,2) predefined type to a *euro* type.

```
INSERT INTO swiss_sales
   SELECT location, CAST (sales AS euro)
   FROM european_sales
   WHERE region = 1038;
```

Example 3: NEW

The following INSERT statement uses NEW to return an *address* UDT expression and insert it into the *european_sales* table.

```
INSERT european_sales (1001, NEW address(), 0);
```

Example 4: Methods and Functions

The following statement uses the built-in constructor function and mutator methods to return a new instance of the *address* UDT and insert it into the *european_sales* table:

```
INSERT INTO european_sales
VALUES (101, address().street('210 Stanton').zip('76543'), 500);
```

Teradata Database executes the UDT expression in the following order:

Step	Invocation	Result
1	*address()* constructor function	Default UDT instance
2	mutator method for *street*	UDT instance with *street* attribute set to '210 Stanton'
3	mutator method for *zip*	UDT instance with *zip* attribute set to '76543'

The final result of the UDT expression is an instance of the *address* UDT with the *street* attribute set to '210 Stanton' and the *zip* attribute set to '76543'.

Related Topics

FOR more information on ...	SEE ...
creating a UDT	CREATE TYPE in *SQL Data Definition Language*.
creating cast definitions for a UDT	CREATE CAST in *SQL Data Definition Language*.
using UDT expressions in DML statements such as SELECT and INSERT	CREATE TYPE in *SQL Data Manipulation Language*.

NEW

Purpose

Constructs a new instance of a structured type and initializes it using the specified constructor method or function.

Syntax

where

Syntax element ...	Specifies ...
SYSUDTLIB.	the database in which the constructor exists. Teradata Database only searches the SYSUDTLIB database for UDT constructors, regardless of whether the database name appears in the NEW expression.
constructor_name	the name of the constructor, which is the same as the name of the structured type.
argument	an argument to pass to the constructor. Parentheses must appear even for constructors that take no arguments.

ANSI Compliance

NEW is partially ANSI SQL:2011 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Usage Notes

You can also construct a new instance of a structured type by calling the constructor method or function. For an example, see "Example 1" on page 1319.

To construct a new instance of a dynamic UDT and define the run time composition of the UDT, you must use the NEW VARIANT_TYPE expression. For details, see "NEW VARIANT_TYPE" on page 1321.

Default Constructor

When a structured UDT is created, Teradata Database automatically generates a constructor function with an empty argument list that you can use to construct a new instance of the structured UDT and initialize the attributes to NULL.

Determining Which Constructor is Invoked

Teradata Database uses the rules in the following table to select a UDT constructor:

IF the NEW expression specifies a constructor with an argument list that is ...	THEN ...
empty	If a constructor method that takes no parameters and has the same name as the UDT: • Exists in the SYSUDTLIB database, Teradata Database selects that constructor method. • Does not exist in the SYSUDTLIB database, Teradata Database selects the constructor function that is automatically generated when the structured UDT is created.
not empty	Teradata Database selects the constructor method in SYSUDTLIB with a parameter list that matches the arguments passed to the constructor in the NEW expression.

Example 1

Consider the following statement that creates a structured UDT named *address*:

```
CREATE TYPE address
AS (street VARCHAR(20)
   ,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE european_sales
  (region INTEGER
  ,location address
  ,sales DECIMAL(8,2));
```

The following statement uses NEW to insert an *address* value into the *european_sales* table:

```
INSERT european_sales (1001, NEW address(), 0);
```

Teradata Database selects the default constructor function that was automatically generated for the *address* UDT because the argument list is empty and the *address* UDT was created with no constructor method. The default *address* constructor function initializes the *street* and *zip* attributes to NULL.

The following statement is equivalent to the preceding INSERT statement but calls the constructor function instead of using NEW:

```
INSERT european_sales (1001, address(), 0);
```

Example 2

To create XML type instances, see *Teradata XML*.

Related Topics

FOR more information on ...	SEE ...
creating constructor methods	CREATE METHOD in *SQL Data Definition Language*.
the constructor function that Teradata Database automatically generates when the structured type is created	CREATE TYPE (Structured Form) in *SQL Data Definition Language*.
constructing a new instance of a dynamic UDT and defining the run time composition of the UDT	"NEW VARIANT_TYPE" on page 1321

NEW VARIANT_TYPE

Purpose

Constructs a new instance of a dynamic or VARIANT_TYPE UDT and defines the run time composition of the UDT.

Syntax

```
NEW VARIANT_TYPE — ( ┌──────────────────────────────────────────────┐ ) —
                     │                                              ,│
                     ├─ expression AS alias_name ──────────────────┤
                     │                                              │
                     └─ table_name.column_name ─┬──────────────────┘
                                                └─ AS alias_name ─┘
```

1101A576

where

Syntax element ...	Specifies ...
expression	any valid SQL expression; however, the following restrictions apply: • *expression* cannot contain a dynamic UDT expression. Nesting of dynamic UDT expressions is not allowed. • the first expression (that is, the first attribute of the dynamic UDT) cannot be a LOB, UDT, or LOB-UDT expression.
alias_name	a name representing the expression or column reference which corresponds to an attribute of the dynamic UDT. When provided, *alias_name* is used as the name of the attribute. You must provide an alias name for any expression that is not a column reference. You cannot assign the same alias name to more than one attribute of the dynamic UDT. Also, you cannot specify an alias name that is the same as a column name if that column name is already used as an attribute name in the dynamic UDT.
table_name	the name of the table in which the column being referenced is stored.
column_name	the name of the column being referenced. If you do not provide an alias name, the column name is used as the name of the corresponding attribute in the dynamic UDT. The same column name cannot be used as an attribute name for more than one attribute of the dynamic UDT. If a column has the same name as an alias name, the column name cannot be used as an attribute name.

ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

Usage Notes

You can use the NEW VARIANT_TYPE expression to define the runtime composition or internal attributes of a dynamic UDT. Each expression you pass into the NEW VARIANT_TYPE constructor corresponds to one attribute of the dynamic UDT. You can assign an alias name to represent each NEW VARIANT_TYPE expression parameter. The name of the attribute will be the alias name provided or the column name associated with the column reference if no alias is provided. This is summarized in the following table:

IF...	THEN the attribute name is...
alias_name is provided	*alias_name*
table_name.column_name is provided, but *alias_name* is not provided	*column_name*
an *expression* is provided that is not a column reference and *alias_name* is not provided	an error is returned.

Note that you must provide an alias name for all expressions that are not column references. In addition, the attribute names must be unique. Therefore, you must provide unique alias names and/or column references.

The data type of the attribute will be the result data type of the expression. The resultant value of the expression will become the value of the corresponding attribute.

Restrictions

- You can use the NEW VARIANT_TYPE expression only to construct dynamic UDTs for use as input parameters to UDFs. To construct a new instance of other structured UDTs, use the NEW expression. For details, see "NEW" on page 1318.
- UDFs support a maximum of 128 parameters. Therefore, you cannot use NEW VARIANT_TYPE to construct a dynamic UDT with more than 128 attributes.
- The sum of the maximum sizes for all the attributes of the dynamic UDT must not exceed the maximum permissible column size as configured for the Teradata Database. Exceeding the maximum column size results in the following SQL error: "ERR_TEQRWOVRFLW _T("Row size or Sort Key size overflow.")".

Example 1

The following NEW VARIANT_TYPE expression creates a dynamic UDT with a single attribute named *weight*:

```
NEW VARIANT_TYPE (Table1.a AS weight)
```

In the next example, the NEW VARIANT_TYPE expression creates a dynamic UDT with a single attribute named *height*. In this example, no alias name is specified; therefore, the column name is used as the attribute name.

```
NEW VARIANT_TYPE (Table1.height)
```

In the next example, the first attribute is named *height* based on the column name. However, the second attribute is also named *height* based on the specified alias name. This is not allowed since attribute names must be unique; therefore, the Teradata Database returns the error, “ERRTEQDUPLATTRNAME - "Duplicate attribute names in the attribute list. %VSTR", being returned to the user.”

```
NEW VARIANT_TYPE (Table1.height, Table1.a AS height)
```

Example 2

This example shows a user-defined aggregate function with an input parameter named *parameter_1* declared as VARIANT_TYPE data type. The SELECT statement calls the new function using the NEW VARIANT_TYPE expression to create a dynamic UDT with two attributes named *a* and *b*.

```
CREATE TYPE INTEGERUDT AS INTEGER FINAL;

CREATE FUNCTION udf_agch002002dynudt (parameter_1  VARIANT_TYPE)
RETURNS INTEGERUDT CLASS AGGREGATE (4) LANGUAGE C  NO SQL
EXTERNAL NAME   'CS!udf_agch002002dynudt!udf_agch002002dynudt.c'
PARAMETER STYLE SQL;

SELECT udf_agch002002dynudt(NEW VARIANT_TYPE (Tbl1.a AS a,
                                              (Tbl1.b + Tbl1.c) AS b))
FROM Tbl1;
```

Related Topics

FOR more information on ...	SEE ...
dynamic UDTs	“VARIANT_TYPE UDT” in *SQL Data Types and Literals*.
constructing a new instance of a structured UDT that is *not* a dynamic UDT	“NEW” on page 1318.
writing UDFs which use input parameters of VARIANT_TYPE data type	*SQL External Routine Programming*

Method Invocation

Purpose

Invokes a method associated with a UDT.

Syntax

where:

Syntax element ...	Specifies ...
database_name	an optional qualifier for the *column_name*.
table_name	an optional qualifier for the *column_name*.
column_name	the name of a distinct or structured UDT column.
udf_name	the name of a UDF that returns a distinct or structured UDT.
argument	an argument to the UDF.
CAST	a CAST expression that converts a source data type to a distinct or structured UDT. Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *SQL Data Definition Language*.

Syntax element ...	Specifies ...
expression	an expression that results in a data type that is compatible as the source type of a cast definition for the target UDT.
udt_name	the name of a distinct or structured UDT.
NEW	an expression that constructs a new instance of a structured type and initializes it using the specified constructor method. For details on NEW, see "NEW" on page 1318.
SYSUDTLIB.	the database in which the constructor exists. Teradata Database only searches the SYSUDTLIB database for UDT constructors, regardless of whether the database name appears in the expression.
constructor_name	the name of a constructor method associated with a UDT. Constructor methods have the same name as the UDT with which they are associated.
argument	an argument to pass to the constructor. Parentheses must appear even though the argument list may be empty.
method_name	the name of an observer, mutator, or user-defined method (UDM). You must precede each method name with a period.
argument	an argument to pass to the method. Parentheses must appear even though the argument list may be empty.

ANSI Compliance

Invocation of UDT methods is partially ANSI SQL:2011 compliant.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Additionally, when a statement specifies an ambiguous expression that can be interpreted as a UDF invocation or a method invocation, Teradata Database gives UDF invocation higher precedence over method invocation. ANSI SQL:2011 gives method invocation higher precedence over UDF invocation.

Observer and Mutator Methods

Teradata Database automatically generates *observer* and *mutator* methods for each attribute of a structured UDT. Observer and mutator methods have the same name as the attribute for which they are generated.

Method	Description	Invocation Example
Observer	Takes no arguments and returns the current value of the attribute.	"Example" on page 1326

Method	Description	Invocation Example
Mutator	Takes one argument and returns a new UDT instance with the specified attribute set to the value of the argument.	"Example 4: Methods and Functions" on page 1317

Usage Notes

When you invoke a UDM on a UDT, Teradata Database searches the SYSUDTLIB database for a UDM that has the UDT as its first parameter followed by the same number of parameters as the method invocation.

If several UDMs have the same name, Teradata Database must determine which UDM to invoke. For details on the steps that Teradata Database uses, see *SQL External Routine Programming*.

Restrictions

To use any of the following functions as the first argument of a method invocation, you must enclose the function in parentheses:

- DATE
- TIME
- VARGRAPHIC

For example, consider a structured UDT called *datetime_record* that has a DATE type attribute called *start_date*. The following statement invokes the *start_date* mutator method, passing in the result of the DATE function:

```
SELECT datetime_record_column.start_date((DATE)) FROM table1;
```

Example

Consider the following statement that creates a structured UDT named *address*:

```
CREATE TYPE address
AS (street VARCHAR(20)
   ,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE european_sales
  (region INTEGER
  ,location address
  ,sales DECIMAL(8,2));
```

The following statement invokes the *zip* observer method to retrieve the value of each *zip* attribute in the *location* column:

```
SELECT location.zip() FROM european_sales;
```

Related Topics

FOR more information on ...	SEE ...
creating methods	CREATE METHOD in *SQL Data Definition Language*.
creating UDTs	CREATE TYPE in *SQL Data Definition Language*.
UDM programming	*SQL External Routine Programming*.

APPENDIX A Notation Conventions

This appendix describes the notation conventions used in this book.

This book uses three conventions to describe the SQL syntax and code:

Convention	Description
Syntax diagrams	Describes SQL syntax form, including options. For details, see "Syntax Diagram Conventions" on page 1329.

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 1334.

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 1336.

Syntax Diagram Conventions

Notation Conventions

Item	Definition / Comments
Letter	An uppercase or lowercase alphabetic character ranging from A through Z.
Number	A digit ranging from 0 through 9. Do not use commas when typing a number with more than 3 digits.

Item	Definition / Comments
Word	Keywords and variables. • UPPERCASE LETTERS represent a keyword. Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase. • lowercase letters represent a keyword that you must type in lowercase, such as a Linux command. • Mixed Case letters represent exceptions to uppercase and lowercase rules. The exceptions are noted in the syntax explanation. • *lowercase italic letters* represent a variable such as a column or table name. Substitute the variable with a proper value. • **lowercase bold letters** represent an excerpt from the diagram. The excerpt is defined immediately following the diagram that contains it. • UNDERLINED LETTERS represent the default value. This applies to both uppercase and lowercase words.
Spaces	Use one space between items such as keywords or variables.
Punctuation	Type all punctuation exactly as it appears in the diagram.

Paths

The main path along the syntax diagram begins at the left with a keyword, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

Continuation Links

Paths that are too long for one line use continuation links. Continuation links are circled letters indicating the beginning and end of a link:

When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue reading.

Required Entries

Required entries appear on the main path:

If you can choose from more than one entry, the choices appear vertically, in a stack. The first entry appears on the main path:

Optional Entries

You may choose to include or disregard optional entries. Optional entries appear below the main path:

If you can optionally choose from more than one entry, all the choices appear below the main path:

Some commands and statements treat one of the optional choices as a default value. This value is UNDERLINED. It is presumed to be selected if you type the command or statement without specifying one of the options.

Strings

String literals appear in apostrophes:

Abbreviations

If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.

In the above syntax, the following formats are valid:

- SHOW CONTROLS
- SHOW CONTROL

Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat:

Read loops from right to left.

The following conventions apply to loops:

IF...	THEN...
there is a maximum number of entries allowed	the number appears in a circle on the return path. In the example, you may type *cname* a maximum of 4 times.
there is a minimum number of entries required	the number appears in a square on the return path. In the example, you must type at least three groups of column names.
a separator character is required between entries	the character appears on the return path. If the diagram does not show a separator character, use one blank space. In the example, the separator character is a comma.

IF...	THEN...
a delimiter character is required around entries	the beginning and end characters appear outside the return path. Generally, a space is not needed between delimiter characters and entries. In the example, the delimiter characters are the left and right parentheses.

Excerpts

Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by (|) terminators on each side of the break. The name for the excerpted piece appears between the terminators in boldface type.

The boldface excerpt name and the excerpted phrase appears immediately after the main diagram. The excerpted phrase starts and ends with a plain horizontal line:

Multiple Legitimate Phrases

In a syntax diagram, it is possible for any number of phrases to be legitimate:

In this example, any of the following phrases are legitimate:

- *dbname*
- DATABASE *dbname*
- *tname*

- TABLE *tname*
- *vname*
- VIEW *vname*

Sample Syntax Diagram

Diagram Identifier

The alphanumeric string that appears in the lower right corner of every diagram is an internal identifier used to catalog the diagram. The text never refers to this string.

Character Shorthand Notation Used In This Book

Introduction

This book uses the Unicode naming convention for characters. For example, the lowercase character ‘a’ is more formally specified as either LATIN CAPITAL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the book, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings.

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in *International Character Set Support*.

Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a–z A–Z 0–9	Any	Any single byte Latin letter or digit.
<u>a</u>–<u>z</u> <u>A</u>–<u>Z</u> <u>0</u>–<u>9</u>	Any	Any fullwidth Latin letter or digit.
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by “ss_3”.
<u>I</u>	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by “ss_2”, forming an individual multibyte character.
<u>Δ</u>	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss_2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss_3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set

```
LMN<TEST>QRS
```

is represented as:

```
D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2
```

Pad Characters

The following table lists the pad characters for the various server character sets.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

Predicate Calculus Notation Used In This Book

Relational databases are based on the theory of relations as developed in set theory. Predicate calculus is often the most unambiguous way to express certain relational concepts.

Occasionally this book uses the following predicate calculus notation to explain concepts.

This symbol ...	Represents this phrase ...
iff	If and only if
∀	For all
∃	There exists

APPENDIX B

Performance Considerations

This Appendix provides suggestions for improving database query performance.

Using CASE Expressions to Optimize Query Design

Effects on Performance

The CASE expression can provide performance improvements:

- For multiple aggregates filtering distinct ranges of values. For example, total sales for several time periods.
- When creating two-dimensional reports directly from Teradata Database. For example, balances in individual accounts held by all bank customers.

CASE expressions help increase performance by returning multiple results in a single pass over the data, instead of needing to make multiple passes over the data and then use the client application to combine the results into a single report.

You can see performance improvements using the CASE expression as the following increase:

- Number of queries against the same source table(s)
- Volume of data in the source table

Valued and Searched CASE Expression

Use one of the following CASE expression forms to return alternate values based on search conditions.

This form...	Tests...	Example

Valued	an expression against possible values.	Create a catalog entitled “Autumn Sale” that shows spring items marked 33% off and summer items marked 25% off.
		`SELECT item_number, item_description, item_price AS 'Current//Price'`
		`, CASE item_season`
		`  WHEN 'spring' THEN item_price *(1-.33)`
		`  WHEN 'summer' THEN item_price *(1-.25)`
		`  ELSE NULL`
		`END AS 'Sale//Price'`
		`FROM inventory_table;`

This form...	Tests...	Example

Searched	arbitrary expression(s).	Repeat the query above, and mark down by 50% summer items with inventories of less than three.
		`SELECT item_number, item_description, item_price AS 'Current//Price'`
		`,CASE`
		` WHEN item_season = 'summer' AND item_count < 3`
		`  THEN item_price *(1-.50)`
		` WHEN item_season = 'summer' AND item_count >= 3`
		`  THEN item_price *(1-.25)`
		`  WHEN item_season = 'spring'`
		`  THEN item_price *(1-.33)`
		`  ELSE NULL`
		`END AS 'Sale//Price'`
		`FROM inventory_table`
		`WHERE item_season IN('spring' OR 'summer');`

The following examples illustrate simple code substitution, virtual denormalization, and single pass examples that use the CASE expression.

Example 1: Simple Code Substitution

For example, instead of joining to a description table, use the CASE expression when the WHERE clause will not contain the case values, that is, the WHERE clause in the example below will not contain region_number.

It is important to not use CASE expression in a view, when the queries that access the view will use the values in a case in the WHERE clause. When this occurs, the Optimizer is unable to evaluate any statistics and a less than optimum query plan could be executed.

```
SELECT CASE region_number
WHEN 1 THEN 'North'
WHEN 2 THEN 'South'
WHEN 3 THEN 'East'
      ELSE 'West' END
,SUM(sales)
FROM sales_table
GROUP BY 1;
```

Example 2: Virtual Denormalization

ABC Telephone Company has a History table with *n* columns, plus call minutes and call type:

- 1 - daytime call
- 2 - night-time call
- 3 - weekend call

You want a summary of call minutes for each call type for each area code on a single line of output.

The standard solution is:

1. Do a GROUP BY on call_type and area code in the History table.
2. Do a self-join to get call_types 1 and 2 into the same row.
3. Do another self-join to get call_type 3 into the same row that contains all three call types.

In the classic denormalization solution, you would physically denormalize the History table by putting all three call types in the same row. However, a denormalized table requires more maintenance.

Instead, you can use the CASE expression to perform a virtual denormalization of the History table:

```
CREATE View DNV
       AS SELECT Col1, ... , Coln
       ,CASE WHEN call_type = 1
         THEN call_minutes END (NAMED Daytime_Minutes)
       ,CASE WHEN call_type = 2
         THEN call_minutes END (NAMED Nighttime_Minutes)
       ,CASE WHEN call_type = 3
         THEN call_minutes END (NAMED Weekend_Minutes)
FROM history;
```

Example 3: Single Pass

In this example, you want a report with five sales columns side by side:

- Current Year, Year to Date (Ytd)
- Current Year, Month to Date (Mtd)
- Last Year, Year to Date (LyYtd)
- Last Year, Month to Date (LyMtd)
- Last Year, Current Month (LyCm)

You currently execute five separate SQL statements and combine the results in an application program.

```
SELECT SUM(sales) ... WHERE sales_date BETWEEN 060101 AND date; [Ytd]
SELECT SUM(sales) ... WHERE sales_date BETWEEN 061001 AND date; [Mtd]
SELECT SUM(sales) ... WHERE sales_date BETWEEN 050101 AND ADD_MONTHS
(date, -12); [LyYtd]
SELECT SUM(sales) ... WHERE sales_date BETWEEN 051001 AND ADD_MONTHS
(date, -12); [LyMtd]
SELECT SUM(sales) ... WHERE sales_date BETWEEN 051001 AND 051031; [LyCm]
```

Instead, you can use the CASE expression to execute one SQL statement that only makes one pass on the Sales_History table.

```
SELECT ...
      SUM(CASE WHEN sales_date BETWEEN 060101 AND date THEN sales ELSE 0
END),  [Ytd]
      SUM(CASE WHEN sales_date BETWEEN 061001 AND date THEN sales ELSE 0
END),  [Mtd]
      SUM(CASE WHEN sales_date BETWEEN 050101 AND ADD_MONTHS (date, -12)
THEN sales ELSE 0 END),[LyYtd]
      SUM(CASE WHEN sales_date BETWEEN 051001 AND ADD_MONTHS (date, -12)
THEN sales ELSE 0 END),[LyMtd]
```

```
      SUM(CASE WHEN sales_date BETWEEN 051001 AND 051031 THEN sales ELSE
0 END), [LyCm]
FROM ...
WHERE sales_date BETWEEN 050101 AND date ...
```

Working with Analytical Functions

Teradata Database support for analytical functions allows you to perform computations at the SQL level rather than through a higher-level calculation engine.

Teradata Database supports ordered analytical syntax.

For complete information, see Chapter 21: "Ordered Analytical / Window Aggregate Functions."

Analytical Functions and Performance

Analytical functions, which are extremely helpful for decision support, speed up order-based analytical queries.

Using analytical functions, you can target the data analysis within the data warehouse itself. This provides several advantages, including:

- Improved processing performance.
- Less data movement across the network.
- Faster analysis than that performed by external tools and sort routines, but full access to ordered analytical functions by external tools such as Teradata Warehouse Miner.

  For example, Teradata Warehouse Miner FREQ function uses CSUM, RANK, and QUALIFY in determining frequencies.
- Support for ANSI version of existing aggregate functions, enabling you to use RANK, SUM, AVG, and COUNT on multiple partitions within a statement select list.
- Simpler SQL programming, particularly because you can use:
  - Nested aggregates with the HAVING clause
  - Window functions
  - QUALIFY, RANK, and ORDER BY clauses

  For example, Teradata Database permits this query structure:

  ```
  SELECT state, city, SUM(sale),
  RANK() OVER
  (PARTITION BY state ORDER BY SUM(sale))
  FROM Tbl1, Tbl2
  WHERE Tbl1.cityid = Tbl2.cityid
  GROUP BY state, city
  HAVING MAX(sale) > 10
  QUALIFY RANK() OVER
  (PARTITION BY state ORDER BY MIN(sale)) > 10;
  ```

Example: Using Teradata RANK

RANK (sort_expression_list) returns the rank (1..n) of all rows in a group by values of sort_expression_list.

For example, assume you enter this query:

```
SELECT ProdId, Month, Sales, RANK(Sales)
FROM SalesHistory
GROUP BY ProdId
QUALIFY RANK(Sales) <=3;
```

The rows of the response table are ranked as follows:

ProdId	Month	Sales	RANK
1234	0607	500	1
1234	0609	300	2
1234	0608	250	3
...			
5678	0609	450	1
5678	0608	150	2
5678	0607	100	3

This opens up possibilities for applying RANK to non-analytical processing that may be cumbersome otherwise.

For example, RANK can:

- Process data sequentially.
- Generate unique sequential numbers on columns that uniquely define the row.
- Process consecutive rows in a predefined order, when you define a self-join on a ranked table.

For example:

1. Create a copy of the table with a new column containing the rank, based on some ordering criteria; for example: term_eff_date or load_event_id.
2. Define a self-join on the table similar to the following:
   - `WHERE A.rankvalue = B.rankvalue - 1`
   - `AND A.policy_id = B.policy_id`
3. Use the self-joined table to process all table rows in a single pass (proceeding from row number *n* to row number *n*+1). This offers significant performance improvement over making multiple passes to process just two rows at a time.

Data in Partition By Column and System Resources

The columns specified in the PARTITION BY clause of a window specification determine the partitions over which the ordered analytical function executes.

For example, the following query specifies the StoreID column in the PARTITION BY clause to compute the group sales sum for each store:

```
SELECT StoreID, SMonth, ProdID, Sales,
SUM(Sales) OVER (PARTITION BY StoreID)
FROM sales_tbl;
```

At execution time, Teradata Database moves all of the rows that fall into a partition to the same AMP. If a very large number of rows fall into the same partition, the AMP can run out of spool space.

For example, if the sales_tbl table in the preceding query has millions or billions of rows, and the StoreID column contains only a few distinct values, an enormous number of rows are going to fall into the same partition, potentially resulting in out-of-spool errors.

To avoid this problem, examine the data in the columns of the PARTITION BY clause. If necessary, rewrite the query to include additional columns in the PARTITION BY clause to create smaller partitions that Teradata Database can distribute more evenly among the AMPs.

For example, the preceding query can be rewritten to compute the group sales sum for each store for each month:

```
SELECT StoreID, SMonth, ProdID, Sales,
SUM(Sales) OVER (PARTITION BY StoreID, SMonth)
FROM sales_tbl;
```

Using the BETWEEN Clause to Reduce Row Redistribution

When retrieving data for a time interval, you can use MIN and MAX dates to define a BETWEEN clause, which results in fewer row redistributions than other methods.

Given a reference calendar that contains 730 rows with:

- calendar_date
- fiscal_week
- fiscal_month
- fiscal_quarter

If you want summary data from a History table for fiscal_quarter, the standard query would be:

```
SELECT H.item_code, SUM(H.items_sold), SUM(H.sales_revenue)
FROM History H , Calendar C
WHERE C.fiscal_quarter = '3Q06'
AND   C.calendar_date = H.sale_date
GROUP BY H.item_code
ORDER BY H.item_code;
```

The Optimizer processes the query as follows:

1. Build a spool table with dates from the reference calendar (90 days).
2. Duplicate the calendar spool. Either:
   - Product join the calendar spool with the History table (90 compares/history table row).
   - Sort both tables to do merge join.

Alternatively, redistribute the entire History table. Product join the large table with the calendar spool (~1 row /AMP).

Another approach is to denormalize the History table. Add fiscal_week, fiscal_month, fiscal_quarter to the History table. Qualify fiscal_month directly in the denormalized table. The penalties for using this approach include:

- Denormalization maintenance costs are higher.
- Extra bytes require more I/Os.

Glossary

AMP Access Module Process

ANSI American National Standards Institute

BLOB Binary Large Object

BTEQ Basic Teradata Query

BYNET Banyan Network

CJK Chinese, Japanese, and Korean

CLIv2 Call Level Interface Version 2

CLOB Character Large Object

CPPI Character Partitioned Primary Index. A partitioned primary index where the partitioning expression involves comparison of CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data types.

cs0, cs1, cs2, cs3 Four code sets (codeset 0, 1, 2, and 3) used in EUC encoding.

distinct type A UDT that is based on a single predefined data type

E2I External-to-Internal

EUC Extended UNIX Code

FK Foreign Key

HI Hash Index

I2E Internal-to-External

JI Join Index

JIS Japanese Industrial Standards

LOB Large Object

LT/ST Large Table/Small Table (join)

NPPI Nonpartitioned Primary Index

NUPI Nonunique Primary Index

NUSI Nonunique Secondary Index

OLAP OnLine Analytical Processing

OLTP OnLine Transaction Processing

PDE Parallel Database Extensions

PE Parsing Engine vproc

PI Primary Index

PK Primary Key

PPI Partitioned Primary Index

predefined type Teradata Database system type such as INTEGER and VARCHAR

RDBMS Relational Database Management System

SDF Specification for Data Formatting

structured type A UDT that is a collection of one or more fields called attributes, each of which is defined as a predefined data type or other UDT (which allows nesting)

Teradata DWM Teradata Dynamic Workload Management

UCS Universal Coded Character Set, specified by International Standard ISO/IEC 10646

UDF User-Defined Function

UDM User-Defined Method

UDT User-Defined Type

UDT expression An expression that returns a distinct or structured UDT data type

UJI Unique Join Index. A non-compressed, single-table join index where the definition includes a unique primary index (UPI), the ROWID keyword in the select list of the SELECT clause, and a WHERE clause that covers a query on the base table (the WHERE clause qualifies a superset of the row set qualified by the WHERE clause of a query on the base table).

UPI Unique Primary Index

USI Unique Secondary Index

vproc Virtual Process

Index

Symbols

||, concatenation operator 1159

A

ABS function 124
ACCOUNT function 350
ACOS inverse trigonometric function 184
ACOSH hyperbolic function 190
ADD_MONTHS function 850
Addition operator 111
Aggregate functions
  AVG 36
  CORR 39
  COUNT 42
  COVAR_POP 46
  COVAR_SAMP 49
  date and 32
  DateTime types and 839
  DISTINCT option and 35
  floating point data and 34
  GROUP BY clause and 32
  GROUPING 52
  HAVING clause and 35
  interval types and 840
  KURTOSIS 55
  literal expressions and 32
  LOB data types and 34
  MAX 57
  MIN 60
  nesting 33
  nulls and 34
  Period data types and 34
  recursive queries and 35
  REGR_AVGX 63
  REGR_AVGY 66
  REGR_COUNT 69
  REGR_INTERCEPT 73
  REGR_R2 77
  REGR_SLOPE 80
  REGR_SXX 84
  REGR_SXY 87
  REGR_SYY 90
  select list containing 31
  SKEW 93
  STDDEV_POP 96
  STDDEV_SAMP 99
  SUM 102
  VAR_POP 105
  VAR_SAMP 108
  when expression evaluates to zero 33
  WHERE clause and 35
Aggregate UDF 1299
ALL predicate quantifier 907
AMP, identify with HASHAMP 884
Analytical functions, performance and 1340
AND logical operator 902
  truth table 904
ANY predicate quantifier 907
Arithmetic functions
  ABS 124
  CEILING 136
  DEGREES 187
  EXP 138
  FLOOR 140
  GREATEST 516
  LEAST 519
  LN 142
  LOG 144
  POWER 149
  RADIANS 187
  RANDOM 151
  ROUND 169
  SIGN 172
  SQRT 174
  TRUNC 176
  WIDTH_BUCKET 178
  ZEROIFNULL 182
Arithmetic operators 1033
  addition operator 111
  division operator 111
  exponentiate 111
  LOB data types and 112
  MOD operator 111
  multiplication 111
  Period data types and 112
  subtraction operator 111
  unary minus operator 111
  unary plus operator 111
ARRAY arithmetic functions 222
ARRAY comparison functions 217
ARRAY concatenation function 214
ARRAY concatenation operator 212
ARRAY Constructor Expression 197

ARRAY Element Reference 194
ARRAY Expressions
  ARRAY Constructor Expression 197
  ARRAY Element Reference 194
ARRAY functions
  Arithmetic functions 222
  ARRAY Scope Reference 200
  ARRAY_AGG 202
  ARRAY_AVG 229
  ARRAY_COMPARE 244
  ARRAY_COUNT_DISTINCT 238
  ARRAY_GET 241
  ARRAY_MAX 232
  ARRAY_MIN 235
  ARRAY_SUM 226, 274
  ARRAY_UPDATE 247
  ARRAY_UPDATE_STRIDE 250
  CARDINALITY 209
  Comparison functions 217
  Concatenation function 214
  UNNEST 206
ARRAY methods
  OCOUNT 256
  ODELETE 282
  OEXISTS 253
  OFIRST 262
  OLAST 265
  OLIMIT 259
  ONEXT 271
  OPRIOR 268
  OTRIM 279
ARRAY operators
  Concatenation operator 212
ARRAY Scope Reference 200
ARRAY_AGG function 202
ARRAY_AVG function 229
ARRAY_COMPARE function 244
ARRAY_COUNT_DISTINCT function 238
ARRAY_GET function 241
ARRAY_MAX function 232
ARRAY_MIN function 235
ARRAY_SUM function 226, 274
ARRAY_UPDATE function 247
ARRAY_UPDATE_STRIDE function 250
ASCII function 1165
ASIN inverse trigonometric function 184
ASINH hyperbolic function 190
ATAN inverse trigonometric function 184
ATAN2 inverse trigonometric function 184
ATANH hyperbolic function 190
Attribute functions 285
  BYTES 286
  CHARACTER_LENGTH 288
  CHARACTERS 291
  DEFAULT 293
  FORMAT 297
  MCHARACTERS 285, 288
  OCTET_LENGTH 298
  TITLE 301
  TYPE 303
AVERAGE aggregate function. See AVG aggregate function
AVG aggregate function
  DateTime types and 839
  described 36
  Interval types and 840

B

BEGIN function 1037
BETWEEN clause, performance and 1342
BETWEEN predicate 912
BITAND function 313
BITNOT function 316
BITOR function 318
BITXOR function 321
Blank, as used in strings 931
BLOB data types
  aggregate functions and 34
  arithmetic operators and 112
  comparison operators and 496
  predicates and 901
Bound functions
  BEGIN function 1037
  END function 1042
  LAST function 1060
Built-in functions 349
  ACCOUNT 350
  CURRENT_DATE 351
  CURRENT_TIME 357
  CURRENT_TIMESTAMP 361
  CURRENT_USER 365
  DATABASE function 366
  DATE function 367
  PROFILE 371
  ROLE 355, 372
  SESSION 374
  TIME 378
  USER 381
Business calendar functions
  DayNumber_Of_Calendar 455
  DayNumber_Of_Month 429
  DayNumber_Of_Week 423
  DayNumber_Of_Year 439
  DayOccurrence_Of_Month 431
  MonthNumber_Of_Calendar 459
  MonthNumber_Of_Quarter 451
  MonthNumber_Of_Year 443
  QuarterNumber_Of_Calendar 461

td_friday 419
td_monday 411
td_month_end 427
td_quarter_begin 445
td_quarter_end 447
td_saturday 421
td_sunday 409
td_thursday 417
td_tuesday 413
td_wednesday 415
td_week_begin 405
td_week_end 407
td_year_begin 435
td_year_end 437
WeekNumber_Of_Quarter 449
YearNumber_Of_Calendar 463
Business calendar functionstd_month_begin 425
Business calendar macros 386
Byte conversion 592
HASHBUCKET function and 891
Byte/bit manipulation functions
BITAND 313
BITNOT 316
BITOR 318
BITXOR 321
COUNTSET 324
GETBIT 326
ROTATELEFT 328
ROTATERIGHT 331
SETBIT 334
SHIFTLEFT 337
SHIFTRIGHT 340
SUBBITSTR 343
TO_BYTE 346
BYTES function 286

C

CALCTMARIX table operator 1270
Calendar functions
td_day_of_calendar 472
td_day_of_month 468
td_day_of_week 466
td_day_of_year 470
td_month_of_calendar 486
td_month_of_quarter 482
td_month_of_year 484
td_quarter_of_calendar 490
td_quarter_of_year 488
td_week_of_calendar 480
td_week_of_month 476
td_week_of_year 478
td_weekday_of_month 474
td_year_of_calendar 492
CALENDAR system view
cumulative sum 985
moving difference 1000
CAMSET function 524
CAMSET_L function 527
CARDINALITY function 209
CASE expression and nulls 575
CASE expression, performance and 1337
CASE operation
COALESCE expression 575
data type of, rules governing 566
defined 555
NULLIF expression 577
searched 561
valued 556
Case sensitivity in comparisons 507
CASE_N function 126
CAST
DECIMAL(18) with a DECIMAL(15) default 672
CAST function 586, 589
ANSI DateTime conversion and 656
DECIMAL data type conversions and 672
CEILING function 136
CHAR function. See CHARACTERS function.
CHAR2HEXINT function 1167
Character
assignability rules for 630
conversion to formatted DATE conversion 603
translation 599
translation (internal to external) 1157
Character sets
restrictions on creation of KANJI1 objects 289, 510, 567, 1157, 1181, 1192, 1230, 1237, 1257
Character string functions. See String functions
CHARACTER_LENGTH function 288
CHARACTERS function 291
ANSI equivalent 288
CHARS function. See CHARACTERS function
CHR function 1170
CLOB data types
aggregate functions and 34
arithmetic operators and 112
comparison operators and 496
predicates and 901
COALESCE expression 575
Comparison evaluations by data type 500
Comparison operators
= 497
> 497
>= 497
GE 497
general rules 498
GT 497
Japanese character sets 510

LE 497
LOB data types and 496
LT 497
NE 497
Period data types 1034
results 498
Comparison rules
floating point data and 499
string 507
Compression functions
CAMSET 524
CAMSET_L 527
LZCOMP 534
LZCOMP_L 536
TD_LZ_COMPRESS 542
TransUnicodeToUTF8 550
TS_COMPRESS 546
TS_DECOMPRESS 548
Concatenation operator 1159
Conditional expressions 902
Conversion
byte 592
byte to INTEGER using HASHBUCKET 891
CAST function and 586
character to character 596
character to DATE 601
character to formatted date 603
character to INTERVAL 606
character to numeric 608
character to Period 614
character to TIME 617
character to TIME WITH TIME ZONE 617
character to TIME, implicit 581, 618, 624
character to TIMESTAMP 623
character to TIMESTAMP, implicit 581, 618, 624
character to UDT 628
data type 579
DATE to character 631
DATE to Period 640
DATE to TIMESTAMP 642
DATE to UDT 648
field mode 591
implicit 579
interval to character 650
INTERVAL to INTERVAL 652
interval to numeric 656
interval to UDT 658
numeric 670
numeric to character 660
numeric to INTERVAL 668
numeric to UDT 674
Period to character 676
Period to DATE 679
Period to Period 681
Period to TIME 686
Period to TIMESTAMP 688
rounding rules 671
signed zone decimal 690
string functions and 1158
table showing supported types 580
Teradata DATE 635
Teradata syntax and 589
TIME to character 694
TIME to Period 697
TIME to TIME 699
TIME to TIMESTAMP 707
TIME to UDT 721
TIMESTAMP to character 723
TIMESTAMP to DATE 727
TIMESTAMP to Period 738
TIMESTAMP to TIMESTAMP 740, 748
TIMESTAMP to UDT 756
truncation rules 671
CORR aggregate function 39
COS trigonometric function 184
COSH hyperbolic function 190
COUNT aggregate function 42
COUNT function
DateTime types and 840
Interval types and 840
COUNTSET function 324
COVAR_POP aggregate function 46
COVAR_SAMP aggregate function 49
CreateBusinessCalendarPattern macro 386
CreateException macro 388
CSUM function 984
CUBE grouping set, GROUPING aggregate function and 52
CUME_DIST 987
Cumulative sum
CALENDAR view 985
computing 984
CURRENT_DATE function 351
CURRENT_TIME function 357
CURRENT_TIMESTAMP function 361
CURRENT_USER function 365

D

Data conversion rules
explicit 589
implicit 579
rounding 671
truncation 671
Data definition 586
byte conversion 592
byte to INTEGER conversion, HASHBUCKET and 891
CAST, data type conversion and 586
character to character conversion 596

character-to-DATE conversion 601
character-to-formatted DATE conversion 603
character-to-INTERVAL conversion 606
character-to-numeric conversion 608
character-to-Period 614
character-to-TIME conversion 617
character-to-TIMESTAMP conversion 623
character-to-UDT conversion 628
data type conversions 579
DATE conversion (Teradata) 635
DATE-to-character conversion 631
DATE-to-Period conversion 640
DATE-to-TIMESTAMP conversion 642
DATE-to-UDT conversion 648
Exact numeric-to-INTERVAL conversion 668
explicit type conversion rules 589
implicit type conversion rules 579
interval-to-character conversion 650
interval-to-exact numeric conversion 656
INTERVAL-to-INTERVAL conversion 652
interval-to-UDT conversion 658
numeric-to-character conversion 660
numeric-to-numeric conversion 670
numeric-to-UDT conversion 674
Period-to-character conversion 676
Period-to-DATE conversion 679
Period-to-Period conversion 681
Period-to-TIME conversion 686
Period-to-TIMESTAMP conversion 688
signed zone decimal conversion 690
TIMESTAMP-to-character conversion 723
TIMESTAMP-to-DATE conversion 727
TIMESTAMP-to-Period conversion 738
TIMESTAMP-to-TIMESTAMP conversion 740, 748
TIMESTAMP-to-UDT conversion 756
TIME-to-character conversion 694
TIME-to-Period conversion 697
TIME-to-TIME conversion 699
TIME-to-TIMESTAMP conversion 707
TIME-to-UDT conversion 721
Database, get default database 366
DATE
as logical predicate 505
scalar operations on 843
Date
get current date (Teradata) 367
get system date 351
Date expressions, Teradata 841
DATE to UDT conversion 648
Date, aggregate operations and 32
DateTime expressions 822
rules for, ANSI 827
DateTime functions, and scalar operations 840
DateTime scalar operations
arithmetic 838
restrictions on 821
DateTime types
aggregate functions and 839
assignment rules 818, 820
DATE-to-Period conversion 640
DATE-to-TIMESTAMP conversion 642
DayNumber_Of_Calendar function 455
DayNumber_Of_Month function 429
DayNumber_Of_Week function 423
DayNumber_Of_Year function 439
DayOccurrence_Of_Month function 431
DECAMSET function 530
DECAMSET_L function 532
DECIMAL/NUMERIC types, arithmetic expression and rounding 121
DECODE 513
Decompression functions
DECAMSET 530
DECAMSET_L 532
LZDECOMP 538
LZDECOMP_L 540
TD_LZ_DECOMPRESS 544
TransUTF8ToUnicode 552
DEFAULT function 293
DEGREES function 187
DeleteAllExceptions macro 389
DeleteException macro 389
DENSE_RANK(ANSI) 990
DISTINCT, SELECT option 35
Division operator 111

E

EDITDISTANCE function 1176
END function 1042
EQUALS function 1044
ESCAPE, with LIKE predicate 931, 936
Exact numeric-to-INTERVAL conversion 668
EXCEPT operator 1144
EXISTS predicate 913
EXP function 138
Exponentiation operator 111
Expressions, defined 28
EXTRACT function 864

F

Fallback AMP, identify with HASHBAKAMP 887
FALSE 904
Field mode, data type conversions and 591
FIRST_VALUE/LAST_VALUE 993
FLOAT data types
aggregate functions and 34
comparison operations and 499

FLOOR function 140
FORMAT phrase 297
Functions
  defined 25
  TEMPORAL_DATE 375
  TEMPORAL_TIMESTAMP 376

G

GETBIT function 326
GetTimeZoneDisplacement function 868
GREATEST 516
GREATEST function 516
GROUP BY clause
  aggregate functions and 32
  rules for aggregate functions and literal expressions 32
Group count, example 975
GROUPING aggregate function
  CUBE and 52
  described 52
  GROUPING SET and 52
  ROLLUP and 52

H

Hadoop data 1276
Hash index, ordered analytical functions and 949
HASHAMP function 884
HASHBAKAMP function 887
HASHBUCKET function 890
Hash-related functions 883
  HASHAMP 884
  HASHBAKAMP 887
  HASHBUCKET 890
  HASHROW 893
HASHROW function 893
Hyperbolic functions 190
  ACOSH 190
  ASINH 190
  ATANH 190
  COSH 190
  SINH 190
  TANH 190

I

IMMEDIATELY PRECEDES function 1051
IMMEDIATELY SUCCEEDS function 1054
Implicit type conversion 579
  byte-to-UDT 594
  character-to-UDT 628
  comparison operators and 502
  DATE-to-UDT 648, 721, 756
  interval-to-UDT 658
  numeric-to-UDT 674
IN predicate 919
INDEX function 1179
  ANSI equivalent 1155
INITCAP function 1185
INSTR function 1187
INTERSECT operator 1141
Interval conversion
  interval-to-character 650
  interval-to-interval 652
  interval-to-UDT 658
Interval expressions 830
  rules for, ANSI 836
INTERVAL function 1057
Interval scalar operations
  arithmetic 838
  restrictions on 821
Interval types
  aggregate functions and 840
  assignment rules 818, 820
Interval-to-character conversion 650
Interval-to-exact numeric conversion 656
INTERVAL-to-INTERVAL conversion 652
Interval-to-UDT conversion 658
Inverse trigonometric functions 184
  ACOS 184
  ASIN 184
  ATAN 184
  ATAN2 184
IS NOT NULL predicate 926
IS NOT UNTIL_CHANGED function 1047
IS NOT UNTIL_CLOSED function 1049
IS NULL predicate 926
IS UNTIL_CHANGED function 1047
IS UNTIL_CLOSED function 1049

J

Japanese character code notation, how to read 1334
Join index, ordered analytical functions and 949

K

KANJI1
  restriction on creation of objects 289, 510, 567, 1157, 1181, 1192, 1230, 1237, 1257
KURTOSIS aggregate function 55

L

LAST function 1060
LDIFF operator 1081
LEAST function 519
Least squares, computing 1003
LENGTH function 1189
LIKE predicate 928

Linear regression, computing 1003
Literal expressions, aggregate functions and 32
LN function 142
LOAD_FROM_HCATALOG table operator 1276
LOG function 144
Logical expressions
  BETWEEN predicate 912
  FALSE result 904
  NOT BETWEEN predicate 912
  TRUE result 904
  UNKNOWN result 904
Logical operators
  AND 902
  defined 902
  NOT 902
  OR 902
  search conditions and 902
Logical predicate
  conditional expression as 899
  DATE as 505
  DEFAULT function and 512, 902
  defined 899
  LOB data types and 901
  order of evaluation 903
  primitives, tabular summary of 899
  SQL use of 899
LOWER function 1191
LPAD function 1193
LTRIM function 1195
LZCOMP function 534
LZCOMP_L function 536
LZDECOMP function 538
LZDECOMP_L function 540

M

Macros
  Business calendar 386
  CreateBusinessCalendarPattern 386
  CreateException 388
  DeleteAllExceptions 389
  DeleteException 389
MAVG function 996
MAX aggregate function
  DateTime types and 840
  described 57
  Interval types and 840
MAXIMUM aggregate function. See MAX aggregate function
MCHARACTERS function 285, 288
  ANSI equivalent 285
MDIFF function 999
MEDIAN 1002
MEDIAN function 1002
MEETS predicate 1062
MIN aggregate function
  DateTime types and 840
  described 60
  Interval types and 840
MINDEX function 1155, 1209
  ANSI equivalent 1155
MINIMUM aggregate function. See MIN aggregate function
MINUS operator 1144
MLINREG function 1003
MOD operator 111
MonthNumber_Of_Calendar function 459
MonthNumber_Of_Quarter function 451
MonthNumber_Of_Year function 443
Moving average, computing 996
Moving difference
  CALENDAR view 1000
  computing 999
Moving sum, computing 1006
MSUBSTR function 1156, 1230
  ANSI equivalent 1156
MSUM function 1006
Multiplication operator 111
Mutator methods 1324

N

Name, get user name 365, 381
NEW expression 1318
NEXT function 1065
NGRAM function 1197
Normalize functions
  TD_NORMALIZE_MEET 1090
  TD_NORMALIZE_OVERLAP 1088
  TD_NORMALIZE_OVERLAP_MEET 1092
  TD_SUM_NORMALIZE_MEET 1096
  TD_SUM_NORMALIZE_OVERLAP 1094
  TD_SUM_NORMALIZE_OVERLAP_MEET 1098
NOT BETWEEN predicate 912
NOT EXISTS predicate 913
NOT IN predicate 919
  NOT EXISTS predicate and 914
  recursive queries and 924
NOT logical operator 902
NULLIF expression 577
NULLIFZERO function 146
Nulls
  aggregate operations and 34
  CASE expression and 575
  searching for/excluding 926
Numeric conversion
  numeric-to-character 660
  numeric-to-date 666
  numeric-to-UDT 674
Numeric-to-character conversion 660

Numeric-to-date conversion 666
Numeric-to-numeric conversion 670
Numeric-to-UDT conversion 674

O

Observer methods 1324
OCOUNT ARRAY method 256
OCTET_LENGTH function 298
ODELETE ARRAY method 282
OEXISTS ARRAY method 253
OFIRST ARRAY method 262
OLAP functions. See Ordered analytical functions.
OLAST ARRAY method 265
OLIMIT ARRAY method 259
ONEXT ARRAY method 271
operators
  arithmetic operators 1033
  defined 27
  LDIFF operator 1081
  P_INTERSECT operator 1072
  P_NORMALIZE operator 1074
  RDIFF operator 1083
OPRIOR ARRAY method 268
OR logical operator 902
  truth table 904
ORDER BY clause
  ordered analytical functions and 949, 955
  window specification and 949
Order of evaluation. See Logical predicate
Ordered analytical functions 945
  aggregates and 970
  common characteristics of 947
  CSUM 984
  CUME_DIST 987
  DENSE_RANK(ANSI) 990
  derived tables and 970
  description 945
  examples 950
  FIRST_VALUE/LAST_VALUE 993
  GROUP BY clause 980
  hash indexes and 949
  HAVING clause 970
  join indexes and 949
  MAVG 996
  MDIFF 999
  MEDIAN 1002
  MLINREG 1003
  MSUM 1006
  ORDER BY clause 949, 955
  PARTITION BY clause 950, 953
  PERCENT_RANK window function 1008
  PERCENTILE_CONT/PERCENTILE_DISC 1012
  Period data types and 948
  QUALIFY clause 947, 970
  QUANTILE 1015
  RANK (Teradata) 1022
  RANK window function 1018
  recursive queries and 949
  result order 949
  ROW_NUMBER window function 1025
  syntax alternatives for 971
  Teradata OLAP functions 971
  Teradata queries, extending 945
  Teradata Warehouse Miner and 946
  views and 970
  window 953
  window functions 971
OREPLACE function 1203
OTRANSLATE function 1206
OTRIM ARRAY method 279
OVERLAPS predicate 1067

P

P_INTERSECT operator 1072
P_NORMALIZE operator 1074
PARTITION BY clause
  affect on spool space 950
  ordered analytical functions and 950, 953
Partitioned primary index. See PPI
PERCENT_RANK window function, described 1008
PERCENTILE_CONT/PERCENTILE_DISC 1012
Period Value Constructor 1030
Period-to-character conversion 676
Period-to-DATE conversion 679
Period-to-Period conversion 681
Period-to-TIME conversion 686
Period-to-TIMESTAMP conversion 688
POSITION function 1155, 1209
POWER function 149
PPI
  defined 127, 158
  maximum partitions when using CASE_N 129
  maximum partitions when using RANGE_N 159
  multilevel 128, 158
  system-derived columns 129, 159
PPI functions
  CASE_N 126
  RANGE_N 155
Precedence
  arithmetic expressions 122
  logical operators 903
  operator 122
  set operators 1128
PRECEDES function 1076
Predicate quantifiers
  ALL 907

ANY 907
SOME 907
Predicates
BETWEEN 912
DEFAULT function and 512, 902
defined 29
EXISTS 913
IN 919
IS NOT NULL 926
IS NULL 926
LIKE 928
logical 899
MEETS 1062
NOT BETWEEN 912
NOT EXISTS 913
NOT IN 919
OVERLAPS 1067
quantifiers 907
PRIOR function 1079
PROFILE function 371
Profiles, getting the current profile 371
Proximity functions
NEXT function 1065
PRIOR function 1079

Q

QUALIFY clause, ordered analytical functions and 947
Quantifiers
ALL 907
ANY 907
SOME 907
QUANTILE function, described 1015
QuarterNumber_Of_Calendar function 461

R

RADIANS function 187
RANDOM function 151
valued CASE and 556
RANGE_N function 155
RANK (Teradata) function 1022
RANK window function 1018
RDIFF operator 1083
REGR_AVGX aggregate function 63
REGR_AVGY aggregate function 66
REGR_COUNT aggregate function 69
REGR_INTERCEPT aggregate function 73
REGR_R2 aggregate function 77
REGR_SLOPE aggregate function 80
REGR_SXX aggregate function 84
REGR_SXY aggregate function 87
REGR_SYY aggregate function 90
Remaining average 949
Remaining count 975
Remaining sum 979
ROLE function 355, 372
Roles, getting the current role 355, 372
ROLLUP grouping set, GROUPING aggregate function and 52
ROTATELEFT function 328
ROTATERIGHT function 331
ROUND function 169
Rounding
arithmetic operators and DECIMAL/NUMERIC data 121
data type conversion rules 671
Row length errors, UNION operator 1147
ROW_NUMBER window function, described 1025
Rowhash, identify with HASHROW function 893
ROWNUM. See ROW_NUMBER window function.
ROWNUMBER. See ROW_NUMBER window function.
RPAD function 1212
RTRIM function 1214

S

Scalar UDF 1295
Scalar, converting scalar value expressions 586
SDF
Currency 611
CurrencyName 611
data type default formats and 613
FORMAT phrase, relationship to 611
GroupingRule 611
GroupSeparator 611
RadixSeparator 611
Search conditions
defined 902
logical operators and 902
Sequenced aggregation functions
TD_SEQUENCED_AVG 1102
TD_SEQUENCED_COUNT 1104
TD_SEQUENCED_SUM 1100
SESSION function 374
Session, get session number 374
Set operators
ALL option 1129
derived tables and 1131
EXCEPT operator 1144
INSERT...SELECT statements containing 1134
INTERSECT operator 1141
MINUS operator 1144
overview 1125
precedence 1128
rules for 1127
rules for connecting queries by 1137
set result, attributes of 1129
subqueries containing 1133
UNION operator 1146

view definitions containing 1136
SETBIT function 334
SHIFTLEFT function 337
SHIFTRIGHT function 340
SIGN function 172
Signed zone decimal conversion 690
SIN trigonometric function 184
SINH hyperbolic function 190
SKEW aggregate function 93
SOME predicate quantifier 907
SOUNDEX function, described 1216
Specification for data formatting, see SDF
SQL and performance
analytical functions 1340
BETWEEN clause 1342
CASE expression 1337
SQL expressions
aggregate functions
AVG 36
CORR 39
COUNT 42
COVAR_POP 46
COVAR_SAMP 49
DISTINCT option 35
GROUPING 52
HAVING clause and 35
KURTOSIS 55
MAX 57
MIN 60
REGR_AVGX 63
REGR_AVGY 66
REGR_COUNT 69
REGR_INTERCEPT 73
REGR_R2 77
REGR_SLOPE 80
REGR_SXX 84
REGR_SXY 87
REGR_SYY 90
SKEW 93
STDDEV_POP 96
STDDEV_SAMP 99
SUM 102
VAR_POP 105
VAR_SAMP 108
WHERE clause and 35
arithmetic functions
ABS 124
CEILING 136
DEGREES 187
EXP 138
FLOOR 140
GREATEST 516
LEAST 519
LN 142
LOG 144
NULLIFZERO 146
POWER 149
RADIANS 187
RANDOM 151
ROUND 169
SIGN 172
SQRT 174
TRUNC 176
ZEROIFNULL 182
arithmetic operators
addition operator 111
division operator 111
exponentiation 111
MOD operator 111
multiplication operator 111
precedence 122
subtraction operator 111
unary minus operator 111
unary plus operator 111
CASE operation 555
COALESCE expression 575
NULLIF expression 577
searched CASE 561
valued CASE 556
comparison operators
= 497
> 497
>= 497
EQ 497
GE 497
GT 497
Japanese character set comparison operators 510
LE 497
LT 497
NE 497
Period data type comparison operators 1034
conditional expressions 902
hyperbolic functions 190
ACOSH 190
ASINH 190
ATANH 190
COSH 190
SINH 190
TANH 190
inverse trigonometric functions 184
ACOS 184
ASIN 184
ATAN 184
ATAN2 184
logical expressions
BETWEEN 912
NOT BETWEEN 912
trigonometric functions 184

COS 184
SIN 184
TAN 184
SQL functions
ARRAY functions
ARRAY_AGG 202
ARRAY_AVG 229
ARRAY_COMPARE 244
ARRAY_COUNT_DISTINCT 238
ARRAY_GET 241
ARRAY_MAX 232
ARRAY_MIN 235
ARRAY_SUM 226, 274
ARRAY_UPDATE 247
ARRAY_UPDATE_STRIDE 250
CARDINALITY 209
UNNEST 206
attribute functions 285
BYTES 286
CHARACTER_LENGTH 288
CHARACTERS 291
DEFAULT 293
FORMAT 297
MCHARACTERS 285, 288
OCTET_LENGTH 298
TITLE 301
TYPE 303
built-in functions
ACCOUNT 350
CURRENT_DATE 351
CURRENT_TIME 357
CURRENT_TIMESTAMP 361
CURRENT_USER 365
DATABASE 366
DATE 367
PROFILE 371
ROLE 355, 372
SESSION 374
TIME 378
USER 381
Business calendar functions
DayNumber_Of_Calendar 455
DayNumber_of_Month 429
DayNumber_Of_Wweek 423
DayNumber_Of_Year 439
DayOccurrence_Of_Month 431
MonthNumber_Of_Calendar 459
MonthNumber_Of_Quarter 451
MonthNumber_Of_Year 443
QuarterNumber_Of_Calendar 461
td_friday 419
td_monday 411
td_month_begin 425
td_month_end 427
td_quarter_begin 445
td_quarter_end 447
td_saturday 421
td_sunday 409
td_thursday 417
td_tuesday 413
td_wednesday 415
td_week_begin 405
td_week_end 407
td_year_begin 435
td_year_end 437
WeekNumber_Of_Quarter 449
YearNumber_Of_Calendar 463
byte strings
BYTES 286
TRIM 1248
byte/bit manipulation functions
BITAND 313
BITNOT 316
BITOR 318
BITXOR 321
COUNTSET 324
GETBIT 326
ROTATELEFT 328
ROTATERIGHT 331
SETBIT 334
SHIFTLEFT 337
SHIFTRIGHT 340
SUBBITSTR 343
TO_BYTE 346
Calendar functions
td_day_of_calendar 472
td_day_of_month 468
td_day_of_week 466
td_day_of_year 470
td_month_of_calendar 486
td_month_of_quarter 482
td_month_of_year 484
td_quarter_of_calendar 490
td_quarter_of_year 488
td_week_of_calendar 480
td_week_of_month 476
td_week_of_year 478
td_weekday_of_month 474
td_year_of_calendar 492
Compression functions
CAMSET 524
CAMSET_L 527
LZCOMP 534
LZCOMP_L 536
TD_LZ_COMPRESS 542
TransUnicodeToUTF8 550
TS_COMPRESS 546
TS_DECOMPRESS 548

Decompression functions
DECAMSET 530
DECAMSET_L 532
LZDECOMP 538
LZDECOMP_L 540
TD_LZ_DECOMPRESS 544
TransUTF8ToUnicode 552
hash-related functions 883
HASHAMP 884
HASHBAKAMP 887
HASHBUCKET 890
HASHROW 893
Ordered analytical functions
CSUM 984
CUME_DIST 987
DENSE_RANK(ANSI) 990
FIRST_VALUE/LAST_VALUE 993
MAVG 996
MDIFF 999
MEDIAN 1002
MLINREG 1003
MSUM 1006
PERCENT_RANK window function 1008
PERCENTILE_CONT/PERCENTILE_DISC 1012
QUANTILE 1015
RANK 1022
RANK window function 1018
ROW_NUMBER window function 1025
partitioning functions
CASE_N 126
RANGE_N 155
string functions 1155
ASCII 1165
CHAR2HEXINT 1167
CHR 1170
concatenation operator 1159
EDITDISTANCE 1176
INDEX 1179
INITCAP 1185
INSTR 1187
LENGTH 1189
LOWER 1191
LPAD 1193
LTRIM 1195
MINDEX 1209
MSUBSTR 1230
NGRAM 1197
OREPLACE 1203
OTRANSLATE 1206
POSITION 1209
RPAD 1212
RTRIM 1214
SOUNDEX 1216
STRING_CS 1220
STRTOK 1223
STRTOK_SPLIT_TO_TABLE 1225
SUBSTR 1228, 1230
SUBSTRING 1228
TRANSLATE 1234
TRANSLATE_CHK 1244
TRIM 1248
TRIM and concatenation 1249
UPPER 1252
VARGRAPHIC 1255
WIDTH_BUCKET function 178
SQL UDF 1290
SQL-H 1276
SQRT function 174
STDDEV_POP aggregate function 96
STDDEV_SAMP aggregate function 99
String functions
ASCII 1165
CHAR2HEXINT 1167
CHR 1170
EDITDISTANCE 1176
implicit character type conversion 1157
INDEX 1179
INITCAP 1185
INSTR 1187
LENGTH 1189
LOWER 1191
LPAD 1193
LTRIM 1195
MINDEX 1155, 1209
MSUBSTR 1156, 1230
NGRAM 1197
OREPLACE 1203
OTRANSLATE 1206
POSITION 1209
RPAD 1212
RTRIM 1214
rules 1157
server character sets and 1157
SOUNDEX 1216
STRING_CS 1220
STRTOK 1223
STRTOK_SPLIT_TO_TABLE 1225
SUBSTR 1228, 1230
SUBSTRING 1228
TRANSLATE 1234
TRANSLATE_CHK 1244
TRIM 1248
UPPER 1252
VARGRAPHIC 1255
STRING_CS function 1220
STRTOK function 1223
STRTOK_SPLIT_TO_TABLE function 1225
SUBBITSTR function 343

Subqueries, comparison operators and 497
SUBSTR function 1228, 1230
  ANSI equivalent 1155
SUBSTRING function 1155, 1228
Subtraction operator 111
SUM aggregate function 102
SUM function, Interval types and 840
Syntax, how to read 1329
SYS_CALENDAR system database 985, 1000

T

Table operators
  CALCMATRIX 1270
  LOAD_FROM_HCATALOG 1276
  TD_UNPIVOT 1285
Table UDF 1310
TAN trigonometric function 184
TANH hyperbolic function 190
td_day_of_calendar function 472
td_day_of_month function 468
td_day_of_week function 466
td_day_of_year function 470
td_friday function 419
TD_LZ_COMPRESS function 542
TD_LZ_DECOMPRESS 544
td_monday function 411
td_month_begin function 425
td_month_end function 427
td_month_of_calendar function 486
td_month_of_quarter function 482
td_month_of_year function 484
TD_NORMALIZE_MEET function 1090
TD_NORMALIZE_OVERLAP function 1088
TD_NORMALIZE_OVERLAP_MEET function 1092
td_quarter_begin function 445
td_quarter_end function 447
td_quarter_of_calendar function 490
td_quarter_of_year function 488
td_saturday function 421
TD_SEQUENCED_AVG function 1102
TD_SEQUENCED_COUNT function 1104
TD_SEQUENCED_SUM function 1100
TD_SUM_NORMALIZE_MEET function 1096
TD_SUM_NORMALIZE_OVERLAP function 1094
TD_SUM_NORMALIZE_OVERLAP_MEET function 1098
td_sunday function 409
td_thursday function 417
td_tuesday function 413
TD_UNPIVOT table operator 1285
td_wednesday function 415
td_week_begin function 405
td_week_end function 407
td_week_of_calendar function 480
td_week_of_month function 476
td_week_of_year function 478
td_weekday_of_month function 474
td_year_begin function 435
td_year_end function 437
td_year_of_calendar function 492
TEMPORAL_DATE, reference 375
TEMPORAL_TIMESTAMP
  reference 376
Teradata conversion syntax 589
Teradata OLAP functions. See Ordered analytical functions
Teradata Warehouse Miner 946
Time
  get current time (Teradata) 378
  get system time 357
Time expressions, Teradata 841
TIME function 378
Time stamp, get system time stamp 361
Time zone comparisons 829
Time zone, get time zone displacement 357
TIME, conversion to character 694
TIMESTAMP
  arithmetic 836
  conversion to character 723
TIMESTAMP-to-DATE conversion 727
TIMESTAMP-to-Period conversion 738
TIMESTAMP-to-TIMESTAMP conversion 740, 748
TIMESTAMP-to-UDT conversion 756
TIME-to-Period conversion 697
TIME-to-TIME conversion 699
TIME-to-TIMESTAMP conversion 707
TIME-to-UDT conversion 721
TITLE function 301
TO_BYTE function 346
TRANSLATE function 1234
TRANSLATE_CHK function 1244
Translation, character 599
TransUnicodeToUTF8 function 550
TransUTF8ToUnicode function 552
Trigonometric functions 184
  COS 184
  SIN 184
  TAN 184
TRIM function 1248
TRIM function, concatenation and 1249
TRUE 904
TRUNC function 176
TS_COMPRESS 546
TS_DECOMPRESS 548
Type conversion, implicit 579
TYPE function 303

U

UDM invocation 1324
UDT data types
  aggregate functions and 37
  arithmetic operators and 112
  CASE expression and 566
  COALESCE expression and 576
  comparison operators and 501
  conversion 758, 761, 765, 768, 771, 774, 777, 780
  hash-related functions and 885, 888
  implicit type conversions and 581
  method invocation 1324
  mutator methods 1324
  NEW expression 1314, 1318
  NULL value 33
  NULLIF expression and 578
  observer methods 1324
  ordered analytical functions and 948
  set operators and 1128
  string functions and 1160
UDT expression 1314
UDT-to-byte type conversion 758
UDT-to-character type conversion 761
UDT-to-DATE type conversion 765
UDT-to-INTERVAL type conversion 768
UDT-to-numeric type conversion 771
UDT-to-TIME type conversion 774
UDT-to-TIMESTAMP type conversion 777
UDT-to-UDT type conversion 780
Unary minus operator 111
Unary plus operator 111
UNION operator 1146
  outer join and 1150
  reason for unexpected row length errors 1147
Universal Coordinated Time, see UTC
UNKNOWN 904
UNNEST function 206
UNTIL_CHANGED value 1030
UNTIL_CLOSED value 1030
UPPER function 1252
USER function 381
User-defined function
  aggregate UDF 1299
  scalar UDF 1295
  SQL UDF 1290
  table UDF 1310
  window aggregate UDF 1302
User-defined types. See UDT data types
Username, get user name 365, 381
UTC
  time conversions and 619, 625
UTF16 client character set
  KANJI1 translation, internal to external 1157
  OCTET_LENGTH and 299
UTF8 client character set
  KANJI1 translation 1157
  OCTET_LENGTH and 299

V

VAR_POP aggregate function 105
VAR_SAMP aggregate functions 108
VARGRAPHIC function 1255
VARGRAPHIC function conversion tables 1258

W

WeekNumber_Of_Quarter function 449
WIDTH_BUCKET function 178
Wildcards, used with LIKE predicate 929
Window aggregate functions
  defined 960
  difference between aggregate functions and 960
Window aggregate UDF 1302
Window functions. See Ordered analytical functions
Window, defined 953

Y

YearNumber_Of_Calendar function 463

Z

ZEROIFNULL function 182